Consistency Modifications for Automatically Tuned Monte-Carlo Tree Search

Vincent Berthier, Hassen Doghmen, Olivier Teytaud

Consistency Modifications for Automatically Tuned Monte-Carlo Tree Search

Vincent Berthier, Hassen Doghmen, and Olivier Teytaud

TAO (Inria), LRI, UMR 8623(CNRS - Univ. Paris-Sud),
bat 490 Univ. Paris-Sud 91405 Orsay, France, firstname.name@lri.fr

Abstract. Monte-Carlo Tree Search algorithms (MCTS [4, 6]), including upper confidence trees (UCT [9]), are known for their impressive ability in high dimensional control problems. Whilst the main testbed is the game of Go, there are increasingly many applications [13, 12, 7]; these algorithms are now widely accepted as strong candidates for high-dimensional control applications. Unfortunately, it is known that for optimal performance on a given problem, MCTS requires some tuning; this tuning is often handcrafted or automated, with in some cases a loss of consistency, *i.e.* a bad behavior asymptotically in the computational power. This highly undesirable property led to a stupid behavior of our main MCTS program MoGo in a real-world situation described in section 3. This is a big trouble for our several works on automatic parameter tuning [3] and the genetic programming of new features in MoGo. We will see in this paper:

- A theoretical analysis of MCTS consistency;
- Detailed examples of consistent and inconsistent known algorithms;
- How to modify a MCTS implementation in order to ensure consistency, independently of the modifications to the "scoring" module (the module which is automatically tuned and genetically programmed in MoGo);
- As a by product of this work, we'll see the interesting property that some heavily tuned MCTS implementations are better than UCT in the sense that they do not visit the complete tree (whereas UCT asymptotically does), whilst preserving the consistency at least if "consistency" modifications above have been made.

1 Introduction: tuning vs consistency in MCTS

Usually, when working on bandits, theoreticians have various models of problems, for which they propose possibly optimal solutions from the point of view of the rates. Unfortunately, to the best of our knowledge, there's no bandit analysis which can be applied for establishing rates for Monte-Carlo Tree Search (MCTS) algorithms; the rates essentially depend on the quantity of tuning you put in your algorithm for biasing the tree search (this tuning is often performed around the "score" function described below). Unfortunately, manually adding heuristics or automatically tuning MCTS certainly improves rates, but it often destroys the good asymptotic properties (consistency) of MCTS. Our goal, in this work, is

to modify MCTS so that we can apply automatic parameter tuning and genetic programming, without any loss of consistency.

Consistency, in UCT-like algorithms [4, 6, 9], is usually considered as trivial, with arguments like "all the tree is asymptotically explored, and therefore the algorithm is consistent". This is certainly true for the "true" UCT version [9], but not necessarily for the many optimized versions of MCTS proposed in the literature [6, 4, 8, 11] which do not visit the whole tree; moreover, as will be shown later, the fact that a MCTS implementation asymptotically builds a complete tree of possible futures does not necessarily make it consistent. These frugal versions (which save up memory and computational power[1]) are the only ones which provide optimal performance. For e.g. the classical testbed of the game of Go, consistency is non trivial. The goal of this research is to provide a as clear as possible frontier between consistent implementations and non-consistent implementations, with the following properties:

- The frontier should provide sufficient conditions that are easy to satisfy by a few corrections in MCTS implementations; also, correcting an optimized implementation for ensuring consistency should give better results on the cases of bad behavior of the algorithm, without reducing its efficiency on low scale experiments (*e.g.* short time settings).
- It should also be compliant with automatic tuning, *i.e.* the modifications should not forbid free modifications of the scoring function.

Proved (section 2) and efficient solutions will be provided, tested on examples (section 3) and experimented (section 4).

Il all the paper, $\#E$ denotes the cardinal of a set E.

2 Model and theory

A game is (here) a finite set of nodes, organized as a tree with a root. Each node n is of one of the following types:

- max node (nodes in which the max player chooses the next state among descendants);
- min node (nodes in which the min player chooses the next state among descendants);
- terminal node; then, the node is equipped with a reward $Reward(n) \in [0, 1]$;
- random node; then, the node is equipped with a probability distribution on its descendants.

In all cases, we note $D(n)$ the set of children of node n. We assume, for the sake of simplicity, that the root node is a max node. We will consider algorithms which perform simulations; the first simulation is s_1, the second simulation is s_2, etc. Each simulation is a path in the game. Each node n is equipped with:

[1] We will refer to these algorithms, which do not necessarily visit all the tree, as "frugal" algorithms. We'll see that usual non-frugal algorithms asymptotically visit all the tree infinitely often.

– Possibly, some side information $I(n)$.
– A father $F(n)$, which is the father node of n; this is not defined for the root.
– A value $V(n)$; this value is the Bellman value; it is known since [2] that $V(n)$, equal to the expected value if both players play optimaly, is well defined.
– For each $t \in \{0, 1, 2, 3, 4, \dots\}$,
  • $n_t(n) \in \{0, 1, 2, \dots\}$ is the number of simulations with index in $1, 2, \dots, t-1$ including node n, possibly plus some constant $K_1(n)$ [6, 5, 11]:
  $$n_t(n) = \#\{i < t; n \in s_i\} + K_1(n). \quad (1)$$
  • $w_t(n) \in \mathbb{R}$ is the sum of the rewards of the simulations with index in $1, 2, \dots, t-1$ including node n, possibly plus some constant $K_2(n)$ (taking into account expert knowledge of offline values):
  $$w_t(n) = \sum_{i<t; n \in s_i} reward(s_i) + K_2(n). \quad (2)$$
  • If n is not the root, $score_t(n) = score(w_t(n), n_t(n), n_t(F(n)), I(n), t)$.

We consider algorithms as in Algorithm 1.

Remarks:

– The score might be stochastic, without impact on our results.
– The fact that the game is a tree, instead of a directed acyclic graph (DAG), is not central; it's just more convenient for notations, as it gives sense to the score of a node, independently of its fathers; all results could be extended to the case of DAGs. Thanks to this simplification, also, we can identify a move and the situation that is reached after this move - this simplifies the writing of Algo. 1.

Let

$$Opt(n) = \{s \in D(n); \forall s' \in D(n), V(s') \leq V(s)\} \text{ if } n \text{ is a max node.}$$

(just replace $\leq$ by $\geq$ for min nodes) $Opt(n)$ is the set of optimal children of n. We note $\hat{V}_t(n) = \frac{w_t(n)}{n_t(n)}$. The case $n_t(n) = 0$ might be solved with *e.g.*

$$n_t(n) = 0 \Rightarrow \hat{V}_t(n) = FPU(n) \quad (3)$$

for some First-Play Urgency function FPU. In all the rest of this paper, unless otherwise stated, $o(.)$ and lim refer to $t \to \infty$; also, "infinitely often" means "for infinitely many t". We now state the following

Input: a game.
Possibly initialize w_t and n_t to some arbitrary values.
for $t = 0, 1, 2, 3, \ldots$ (until no time left) **do**
  $s_t \leftarrow ()$ // empty simulation
  $s = root(game)$
  while s is not terminal **do**
    $s_t \leftarrow s_t.s$ // s is added to the simulation
    switch s **do**
      case max *node*
        $s \leftarrow \arg\max_{n \in D(s)} score(n)$
      end
      case min *node*
        $s \leftarrow \arg\min_{n \in D(s)} score(n)$
      end
      case *random node*
        $s \leftarrow$ randomly drawn node according to distribution at s
      end
    end
  end while
  $s_t \leftarrow s_t.s$ // s is added to the simulation
end for
$decision = \arg\max_{n \in D(root)} n_t(n)$ // decision rule
Output: a *decision*, *i.e.* the node in which to move.

Algorithm 1: A framework of Monte-Carlo Tree Search. All usual implementations fall in this schema depending on the *score* function. The decision step is sometimes different, without impact on our results.

Theorem (Consistency of Monte-Carlo Tree Search.) *Consider a MCTS Algorithm as in Algo. 1. Assume that for all sons s, s' of a* max *node:*

$$n_t(s) \to \infty, n_t(s') \to \infty, \liminf \hat{V}_t(s) > \limsup \hat{V}_t(s') \Rightarrow n_t(s') = o(n_t(s)) \quad (4)$$

$$n_t(F(s)) \to \infty \text{ and } \liminf \hat{V}_t(F(s)) < V(F(s)) \Rightarrow n_t(s) \to \infty \quad (5)$$

Assume the same equations 4 and 5, with $\liminf$ *replaced by* $\limsup$ *and* $<$ *exchanged with* $>$*, for* min *nodes. Then, almost surely, there exists* t_0 *such that*

$$\forall t \geq t_0, \arg \max_{n \in D(root)} N_t(n) \subset \arg \max_{n \in D(root)} V(n). \quad (6)$$

Eq. 6 is the consistency of MCTS, for algorithms using the decision rule in Alg. 1. There are some other decision rules used in MCTS implementations, but Eq. 6 also immediately shows the optimality of these decision rules (other known decision rules are asymptotically equivalent to the classical decision rule we have proposed).

Remark 1: To the best of our knowledge, Eq. 4 holds for all strong implementations. This is certainly not the case for Eq. 5; we'll see counter-examples in section 3.

Remark 2: Eq. 5 becomes particularly interesting when an upper bound on $V(F(s))$ is known: no use exploring new children when $\hat{V}_t(s) \to V(F(s))$. In deterministic games with binary rewards, Eq. 5 boils down to:

$$\liminf \hat{V}_t(F(s)) < 1 \Rightarrow n(s) \to \infty.$$

We don't request anything if $\liminf \hat{V}_t(F(s)) = 1$. This means that only one move can be simulated at s, as long as we have no refutation for this move. This implies a in-depth analysis of a few moves, which is central in the success of MCTS e.g. in Go and Havannah [14].

Remark 3: We only claim $\subset$ and not $=$ in Eq. 6. We will choose an optimal move, but we will not necessarily find all optimal moves.

Proof of the Theorem:

We will first show that for each node n,

$$\lim_{t \to \infty} n_t(n) = \infty \Rightarrow \hat{V}_t(n) \to V(n). \quad (7)$$

The proof of Eq. 7 is made by induction; we show it for leaf nodes, and then we show that if it holds for the sons of the node n, then it holds for n.

Proof of Eq. 7: If n is a leaf (a terminal node), then Eq. 7 obviously hold.

Let's now assume that Eq. 7 holds for sons of n, and let's show it for n. This is immediate if n is a random node. The case of "min" is symetrical to the case of "max" nodes, and we will therefore only consider the case of max.

First, obviously $\limsup \hat{V}_t(n) \leq V(n)$. This is because $\hat{V}_t(n)$ is a weighted average of the $\hat{V}_t(s)$, for s sons of n, and $\hat{V}_t(s) \to V_t(s)$ for all s simulated infinitely often.

Therefore, we just have to show that

$$\liminf \hat{V}_t(n) \geq V(n). \tag{8}$$

In order to show Eq. 8, we will assume, in order to get a contradiction, that

$$\liminf \hat{V}_t(n) < V(n). \tag{9}$$

If Eq. 9 holds, then, by Eq. 5, each s (son of n) is simulated infinitely often.

Therefore, for each s, son of n, $\hat{V}_t(s) \to V_t(s)$. By Eq. 4, this implies that all suboptimal sons s' verify $n_t(s') = o(n_t(s))$ for any $s \in Opt(n)$. This implies that $\hat{V}_t(n)$ converges to a weighted average of the $\hat{V}_t(s)$, for some (possibly one, possibly several) $s \in Opt(n)$ simulated infinitely often. $\hat{V}_t(s) \to V(s)$ for all these sons, by the induction hypothesis. This implies

$$\hat{V}_t(n) \to \max_{s \in Opt(n)} V_t(s).$$

This is a contradiction with Eq. 9; Eq. 8 is therefore proved. This concludes the proof of Eq. 7, by induction.□

We have shown Eq. 7. We now have to show the conclusion of the theorem, *i.e.* Eq. 6. If all sons of the root are optimal, there is nothing to prove; let's assume that at least one son of the root is not optimal.

Consider Δ the minimum difference between $V(s)$ for suboptimal sons, and $V(root)$; *i.e.*:

$$\Delta = V(root) - \sup_{s \in D(root), V(s) < V(root)} V(s).$$

$\Delta > 0$ by assumption.

Eq. 7 in particular holds for the root node. This implies that

$$\hat{V}(root) \to V(root). \tag{10}$$

Assume, in order to get a contradiction, that for some $k > 0$

$$\sum_{s \in D(root), V(s) < V(root)} n_t(s) \geq k \cdot t \quad \text{infinitely often.} \tag{11}$$

Then, $\hat{V}_t(root)$ is infinitely often the weighted average of

- the $\hat{V}_t(s)$, for s optimal nodes, with total weight at most $1 - k$;
- the $\hat{V}_t(s')$, for s' suboptimal nodes, with total weight at least k.

This implies that $\limsup \hat{V}_t(root) \leq V(root) - k\Delta$; this contradicts Eq. 10; therefore the assumption (Eq. 11) leads to a contradiction. We have therefore shown, by contradiction, that Eq. 11 does not hold, for any $k > 0$. This implies Eq. 6.
□

3 Remarks and examples

No consistency with naive ϵ-greedy algorithms. In MCTS algorithms, ϵ-greedy strategies do not provide consistency. For example, even with a finite set of nodes, and for any positive value of ϵ, a rule like "in each node, with probability ϵ, select a node randomly and uniformly instead of using the score for choosing the next node" is not consistent. A counter-example is given in Fig. 1.

Saving up memory with MCTS algorithms. One might remark, however, that this counter-example is limited to ϵ-greedy algorithms with ϵ constant; we recover consistency if we have $\epsilon \to 0$ sufficiently quickly (but sufficiently slowly as well). However, ϵ-greedy algorithms, in all these cases, have the drawback that all the tree is visited infinitely often. In particular, all the tree is constructed in memory. As pointed out above, this is certainly not the case of MCTS algorithms with other rules: in the binary case with reward in $\{0, 1\}$, if a given arm always wins, and at least for good bandit scores, then none of the other arms is never tested.

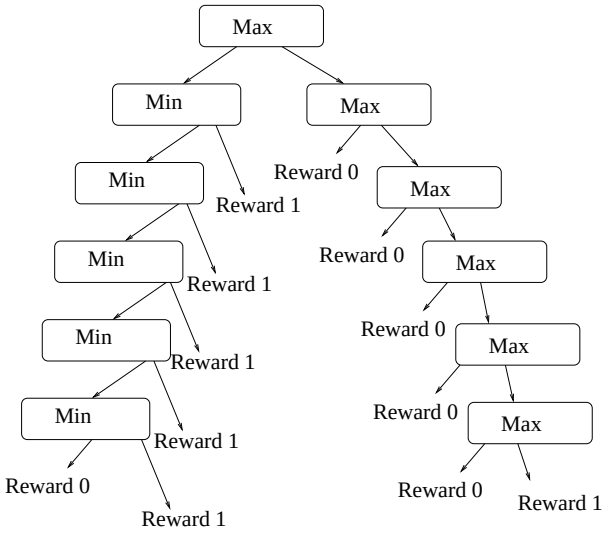

Fig. 1. Counter-example for naive ϵ-greedy exploration in MCTS. In this game, the max player should choose the right side and win a reward 1. However, for any ϵ, with an ϵ-greedy heuristic, and for many bandit rules (even UCB, which would be obviously consistent without the ϵ-greedy modification), MCTS would choose the left side (at least if the number of levels is sufficient), and loose if the min player plays well. Please note however that this counter-example would not hold for ϵ decreasing to zero. This is a counter-example for fixed ϵ only. However (see text), the case of ϵ decreasing to zero can be consistent (depending on the score) but it will certainly not have the frugality property: it will visit all the tree infinitely often.

Consistency of UCT. UCT is Algorithm 1 with the particular case

$$score_t(s) = \hat{V}_t(s) + \sqrt{\log(n_t(F(s)))/n_t(s)} \quad (12)$$

(constants or other terms are often placed before $\log(i)$; this does not matter for us here) Obviously, the classical UCT (with UCB [10, 1] as bandit algorithm) verifies our assumptions; its variants like "progressive widening" [6], "progressive unpruning" [5] (see also [15] for these modifications of UCB), as well. Our purpose is precisely to consider cases different from UCT, and in particular the frugal versions (those which do not visit all the tree).

Unconsistency of some forms of progressive bias. This is not the case for all "progressive bias" methods [8, 5]. For example, a classical formula for Rave values is

$$score_t(s) = \frac{n_t(s)}{n_t(s)+K}\hat{V}_t(s) + \frac{K}{n_t(s)+K}V'_t(s) \text{ for some constant } K > 0, \quad (13)$$

where $V'_t(s)$ is the Rave value of s [8].

It has been pointed out in the computer-Go mailing-list that in some cases, the Rave heuristic [8] gives value $V'_t(g) = 0$ to the only good move (cf the interesting posts by Brian Sheppard on the computer-go mailing list `http://www.mail-archive.com/computer-go@computer-go.org/msg12202.html`). If the score is

$$score_t(g) = \frac{n_t(g)}{n_t(g)+K}\hat{V}_t(g) + \frac{K}{n(g)+K}V'_t(g),$$

then the score of the good move g is 0 as long as $n_t(g) = 0$; this move is never simulated as long as there is a move with score > 0. The trouble is that a bad move b might have a non-zero Rave value, and therefore its score is

$$score_t(b) \geq \frac{K}{n_t(b)+K}V'_t(b) > 0.$$

This means that only the bad move has a non-zero score, the good move stays at a score 0 and is never simulated.

Unconsistency of negative heuristics. The program MoGo, known for several successes in the game of Go (including the only ever win against a pro with handicap 6 and the only ever win against a top pro 9p (winner of the LG Cup 2007) with handicap 7. MoGo has a complicated bandit involving a compromise between (i) the empirical reward $\hat{V}_t(s)$ (ii) the Rave value $V'_t(s)$ (iii) the pattern-based value (combined with expert rules) $H(I(s))$. A score can in fact, when patterns and expert rules agree against a move, be negative; this means that the score can be negative. The trouble is that there are particular cases in which even very bad patterns might be the only good move; Go experts know some examples in which the famous "empty triangle" (known as a very bad pattern) can be a good move.

We'll see below that this can happen in MoGo; this is not only theoretical: the situation given below occured in a real game against a professional player. MoGo

was convinced that it was going to win, whereas the opponent was convinced that he was in good situation - when black replied E6, the computer (white) simulated this move (which was not yet simulated at all, except during a small transitory regime!) and understood that it had lost the game.

A game lost by MoGo against Fan Hui in 9x9 Go. MoGo had the opportunity to play against Fan Hui, 2nd Dan pro, in Toulouse (August 2009). MoGo won one game, lost two games; if the game presented below had been won by MoGo, this would have been the first ever win of a computer against a pro on a complete match (and not only on a simple game). A very surprising situation happened: whereas usually MoGo, as well as many MCTS implementations, is extremly strong in endgames, it was completly convinced of winning the game whereas the situation was desperate. The game is presented in Fig. 2.

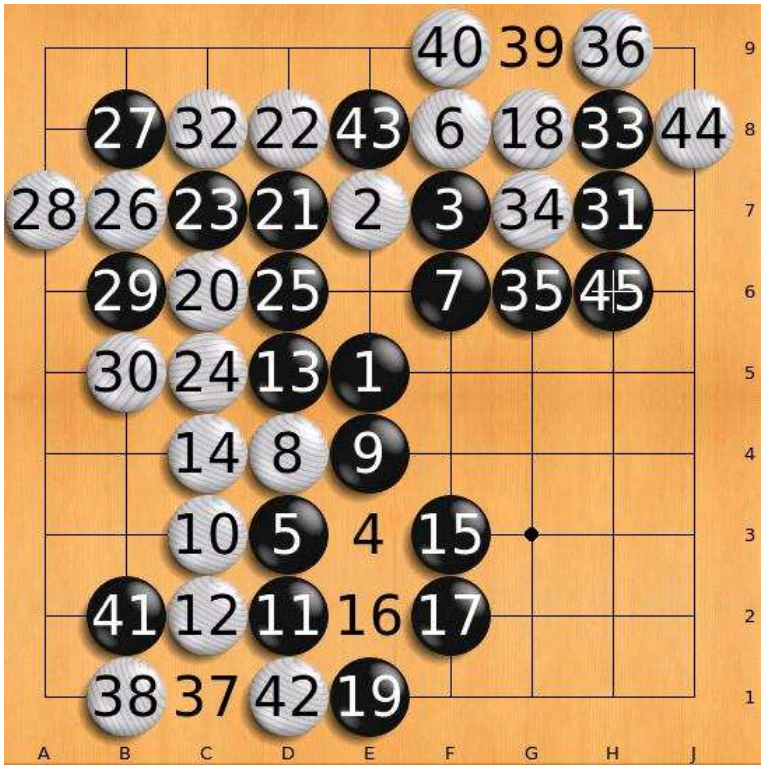

Fig. 2. The game lost by MoGo against the professional player Fan Hui: this is a ko fight. MoGo (white) plays C1 and is very confident; in fact, black replies E6 (which was not simulated by MoGo!) and wins.

Tricks for consistency. We propose below some simple implementation tricks for ensuring that the algorithm is consistent. All "consistency" tricks can

easily be checked using Eq. 4 and 5 in the theorem above; and easy counter-examples can be built for all non-consistent rules. The case in which the reward lies in a finite set is particularly interesting, as we'll see that we can then have consistency whilst preserving the suitable property that we do not simulate the whole tree.

- Using $score_t(s) = \hat{V}_t(s)$ with a FPU as in Eq. 3 (and $K_1(.) = K_2(.) = 0$ in Eqs 1 and 2) is not consistent (if there is only one good move and if the first simulation with this move leads to a loss, we can have a score 0 for this move (for the eternity as it will never be simulated again), and a score > 0 (for the eternity) for a bad move which alway looses except for its very first simulation; consistency is recovered with

  $$score_t(s) = \hat{V}_t(s) + \sqrt{\log(n_t(F(s)))/n_t(s)}$$

  but this is not satisfactory as it implies that all the tree is visited infinitely often.
- For Rave values for the case of deterministic games with reward $\in \{0, 1\}$ (see Eq. 13 and non-consistency discussed there), a simple solution consists in replacing $V'(t)$ by $\max(V'(t), 0.1)$.
- For methods with score closely related to the average reward without any upper confidence bound term, we suggest:

  $$score_t(s) = \frac{w_t(s) + K}{n_t(s) + 2K} \tag{14}$$

  with $K > 0$. This was empirically derived, for the program MoGo, in [11] as a good tool, instead of $\frac{w_t(s)}{n_t(s)}$ (*i.e.* the case $K = 0$) when there's no upper confidence term $+\sqrt{\log(n_t(F(s)))/n_t(s)}$ (and removing this upper confidence term makes MoGo much faster!). Interestingly, we clearly see here an advantage of Eq. 14 instead of Eq. 12: we have consistency in both cases (Eqs 4 and 5 are clearly verified with Eq. 14 and Eq. 12), but the difference is that Eq. 12 implies that all the tree is visited infinitely often, whereas Eq. 14 does not: with Eq. 14 some parts of the tree are never visited.
- Consider now the case in which the rule is modified by genetic programming and automatic tuning, and we don't want to modify the score manually. Consider the case of a deterministic game with discrete reward $\in \{0, 1\}$. Importantly, this is not only for games; adversarial settings are important in verification, and are usual models for unkown random processes. Consider $V_{threshold}$ some constant such that for n a max node:

  $$\exists (s_1, s_2) \in D(n),$$

  $$V(s_1) < V(s_2) \Rightarrow V_{threshold} \in]\sup_{s \in D(n), V(s) < V(n)} V(s), V(root)[.$$

  (just replace:
  - $V_{threshold}$ by $V'_{threshold}$,

- $<$ by $>$,
- and sup by inf

for min nodes)
We see that if Bellman values live in $\{0,1\}$ (games with binary reward and no chance), it is sufficient to choose any $V_{threshold} \in]0,1[$, and to use Alg. 2 instead of Alg. 1 (this is a one-line modification of MCTS!). For the game of Go, $V_{threshold} = 0.3$ is used in our experiments (this was handcrafted, without any tuning). The idea is simply that if $\hat{V}_t(s) < V_{threshold}$ (for a max node) then we simulate randomly one of the sons of s instead of maximizing the score. This is the choice made for our tests below.

Input: a game.
Possibly initialize w_t and n_t to some arbitrary values.
for $t = 0, 1, 2, 3, \ldots$ (until no time left) **do**
  $s_t \leftarrow ()$ // empty simulation
  $s = root(game)$
  while s is not terminal **do**
    $s_t \leftarrow s_t.s$ // s is added to the simulation
    if s is a max (resp. min) node and $\hat{V}_t(s) < V_{threshold}$ (resp. $\hat{V}_t(s) > V'_{threshold}$) **then**
      randomly draw $s \in D(s)$
    else
      switch s **do**
        case max *node*
          $s \leftarrow \arg\max_{n \in D(s)} score(n)$
        end
        case min *node*
          $s \leftarrow \arg\min_{n \in D(s)} score(n)$
        end
        case *random node*
          $s \leftarrow$ randomly drawn node according to distribution at s
        end
      end
    end if
  end while
  $s_t \leftarrow s_t.s$ // s is added to the simulation
end for
$decision = \arg\max_{n \in D(root)} n_t(n)$ // decision rule
Output: a *decision*, *i.e.* the node in which to move.

Algorithm 2: A modification of Algo. 1 which ensures consistency for binary deterministic games. The difference between this algorithm and Algo. 1 is the application of the consistency modification in the Theorem.

4 Experiments

In the game presented in Fig. 2, MoGo poorly estimated the situation. We check now if the new version of MoGo (the version using the consistency modification in Eq. 2) understands more clearly that in this situation, the game is over for white. MoGo provides the following results with 500000 simulations per move:

- the expected success rate is above 90% in 13 over 30 runs in the initial version of MoGo;
- never over 30 runs with the change suggested in the theoretical analysis above.

We could also test the version with the modification against the version without the modification; no decrease of performance could be found. We also did not find significant improvements. Results are presented in Table 1. These experiments are performed with the "score" function of MoGo`http://www.lri.fr/~teytaud/mogo.html`. Usual experiments performed in the field consider nearly 10 000 simulations per move - this is because it's much easier to perform automatic parameter tuning with games played in 1s per move (minutes for complete games) than with 5 000 000 simulations per move (which require half an hour per game on a strong machine - this is closer to real-world cases, but much more expensive).

Number of simulations per move	Success rate of Algo. 2 vs Algo. 1
500 000 (0.3)	52.9 % ± 0.9 %
5 000 000 (0.3)	52.1 % ± 4.9 %

Table 1. Performance of MoGo with the "consistency" modification, vs the current version of MoGo.

5 Conclusion

MCTS is a recent and very important class of algorithms. We have shown that consistency becomes an issue in MCTS programs which are highly optimized on experiments with limited scales (typically, limited time settings). This becomes particularly important for genetically evolved programs, as well as automatically tuned programs. We have shown that there are real world cases of failures due to this lack of consistency.

We proposed mathematically proved methods for ensuring consistency independently of the parts of MCTS which are automatically tuned. Interestingly, we also point out that some heavily optimized algorithms are better than UCT in some settings in the sense that they do not simulate the whole tree, whereas UCT does (UCT even simulates the whole tree infinitely often!); and they nonetheless

preserve consistency, at least if tricks for consistency as presented in this paper are applied.

Interestingly, the results are particularly visible on a real-world case which would be difficult to reproduce in artificial experiments: big ko-fights as in Fig. 2 are based on cultural knowledge of strong players and are rare in games between computers. It's very difficult to derive a good behavior on such cases just on the basis of artificial experiments, whereas a mathematical consistency analysis has succeeded. This shows the soundness and the generality of the approach.

The main limitation of this work is that it is limited, at least for the detailed mathematical analysis, to the tuning of MCTS algorithms. However, MCTS is a very important class of algorithms, for control and games. Moreover, the principle of exploring infinitely often *only* children of nodes which have empirical reward below the expected value in case of success control makes sense in all simulation-based control algorithms.

Acknowledgements

We thank the various people who inspired the development of MoGo: Bruno Bouzy, Tristan Cazenave, Albert Cohen, Thomas Hérault, Rémi Coulom, Rémi Munos, the computer-go mailing list, the KGS community, the Cgos server, the IAGO challenge, the Recitsproque company, the National University of Tainan (Taiwan). We thank Grid5000 for providing computational resources for experiments presented in this paper. We thank Jean-Yves Papazoglou and Eric Saves for organizing the games with Fan Hui and for the opportunity of meeting plenty of great Go player. We also thank the many direct contributors of MoGo, who can't all be cited here `http://www.lri.fr/~teytaud/mogo.html`.

References

1. P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2/3):235–256, 2002.
2. R. Bellman. *Dynamic Programming*. Princeton Univ. Press, 1957.
3. G. Chaslot, J.-B. Hoock, F. Teytaud, and O. Teytaud. On the huge benefit of quasi-random mutations for multimodal optimization with application to grid-based tuning of neurocontrollers. In *ESANN*, Bruges Belgium, 2009.
4. G. Chaslot, J.-T. Saito, B. Bouzy, J. W. H. M. Uiterwijk, and H. J. van den Herik. Monte-Carlo Strategies for Computer Go. In P.-Y. Schobbens, W. Vanhoof, and G. Schwanen, editors, *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91, 2006.
5. 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.
6. 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*, 2006.

7. F. De Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel. Bandit-Based Optimization on Graphs with Application to Library Performance Tuning. In *ICML*, Montréal Canada, 2009.
8. 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.
9. L. Kocsis and C. Szepesvari. Bandit-based monte-carlo planning. In *ECML'06*, pages 282–293, 2006.
10. T. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6:4–22, 1985.
11. 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 (accepted).
12. 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.
13. P. Rolet, M. Sebag, and O. Teytaud. Optimal robust expensive optimization is tractable. In *Gecco 2009*, page 8 pages, Montréal Canada, 2009. ACM. G.: Mathematics of Computing/G.1: NUMERICAL ANALYSIS/G.1.6: Optimization, I.: Computing Methodologies/I.2: ARTIFICIAL INTELLIGENCE/I.2.8: Problem Solving, Control Methods, and Search.
14. F. Teytaud and O. Teytaud. Creating an Upper-Confidence-Tree program for Havannah. In *ACG 12*, Pamplona Espagne, 2009.
15. Y. Wang, J.-Y. Audibert, and R. Munos. Algorithms for infinitely many-armed bandits. In *Advances in Neural Information Processing Systems*, volume 21, 2008.