# Biasing Monte-Carlo Simulations through RAVE Values

Arpad Rimmel, Fabien Teytaud, Olivier Teytaud

# Biasing Monte-Carlo Simulations through RAVE Values

Arpad Rimmel[1], Fabien Teytaud[2], and Olivier Teytaud[2]

[1] Department of Computing Science, University of Alberta, Canada,
rimmel@cs.ualberta.ca
[2] TAO (Inria), LRI, UMR 8623(CNRS - Univ. Paris-Sud),
bat 490 Univ. Paris-Sud 91405 Orsay, France

**Abstract.** The Monte-Carlo Tree Search algorithm has been successfully applied in various domains. However, its performance heavily depends on the Monte-Carlo part. In this paper, we propose a generic way of improving the Monte-Carlo simulations by using RAVE values, which already strongly improved the tree part of the algorithm. We prove the generality and efficiency of our approach by showing improvements on two different applications: the game of Havannah and the game of Go.

## 1 Introduction

Monte-Carlo Tree Search (MCTS) [5, 6, 10] is a recent algorithm for taking decisions in a discrete, observable, uncertain environment with finite horizon. This algorithm is particularly interesting when the number of states is huge. In this case, classical algorithms like Minimax and Alphabeta [9], for two-player games, and Dynamic Programming [13], for one-player games, are too time-consuming or not efficient. MCTS combines an exploration of the tree based on a compromise between exploration and exploitation, and an evaluation based on Monte-Carlo simulations. A classical generic improvement is the use of the RAVE values [8]. This algorithm and this improvement will be described in section 2. It achieved particularly good results in two-player games like computer Go [12] or Havannah [15]. Moreover, it was also successfully applied on one-player problems like the automatic generation of libraries for linear transforms [7], non-linear optimization [2] or active learning [14].

The algorithm can be improved by modifying the Monte-Carlo simulations. For example, in [16], the addition of patterns to the simulations leads to a significant improvement in the case of the game of Go. However, those patterns are domain-specific. In this paper, we propose a generic modification of the simulations based on the RAVE values that we called "poolRave". The principle is to play moves that are considered efficient according to the RAVE values with a higher probability than the other moves. We show significant positive results on two different applications: the game of Go and the game of Havannah.

We first present the principle of the Monte-Carlo Tree Search algorithm and of the RAVE improvement (section 2). Then, we introduce the new Monte-

Carlo simulations (section 3). Finally, we present the experiments (section 4) and conclude.

## 2 Monte-Carlo Tree Search

MCTS is based on the incremental construction of a tree representing the possible future states by using (i) a bandit formula (ii) Monte-Carlo simulations. Section 2.1 presents bandits and section 2.2 then presents their use for planning and games, i.e. MCTS.

### 2.1 Bandits

A $k$-armed bandit problem is defined by the following elements:

- A finite set of arms is given; without loss of generality, the set of arms can be denoted $J = \{1, \dots, k\}$.
- Each arm $j \in J$ is equipped with an unknown random variable $X_j$; the expectation of $X_j$ is denoted $\mu_j$.
- At each time step $t \in \{1, 2, \dots\}$:
  - The algorithm chooses $j_t \in J$ depending on $(j_1, \dots, j_{t-1})$ and $(r_1, \dots, r_{t-1})$.
  - Each time an arm $j$ is selected, the algorithm gets a reward $r_t$, which is an independent realization of $X_{j_t}$.

The goal of the problem is to minimize the so-called *regret*. Let $T_j(n)$ be the number of times an arm has been selected during the first $n$ steps. The *regret* after $n$ steps is defined by

$$\mu^* n - \sum_{j=1}^{k} \mu_j \mathbb{E}[T_j(n)] \text{ where } \mu^* = \max_{1 \leq i \leq k} \mu_i.$$

$\mathbb{E}[T_j(n)]$ represents the esperance of $T_j(n)$.

In [1], the authors achieve a logarithmic regret (it has been proved that this is the best obtainable regret in [11]) independently of the $X_j$ with the following algorithm: first, try one time each arm; then, at each step, select the arm $j$ that maximizes

$$\bar{x_j} + \sqrt{\frac{2ln(n)}{n_j}}. \tag{1}$$

$\bar{x_j}$ is the average reward for the arm $j$ (until now). $n_j$ is the number of times the arm $j$ has been selected so far. $n$ is the overall number of trials so far. This formula consists in choosing at each step the arm that has the highest upper confidence bound (UCB). It is called the UCB formula.

### 2.2 Monte-Carlo Tree Search

The MCTS algorithm constructs in memory a subtree $\hat{T}$ of the global tree $T$ representing all the possible future states of the problem (the so-called extensive form of the problem). The construction of $\hat{T}$ is done by the repetition (while there is some time left) of 3 successive steps: *descent, evaluation, growth*. The algorithm is given in Alg. 1 (Left) and illustrated in Fig. 1.

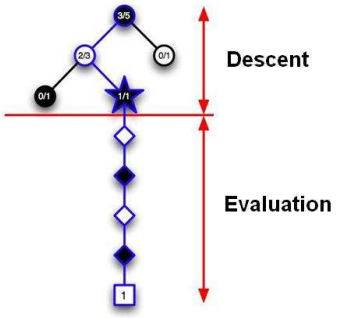


**Fig. 1.** Illustration of the Monte-Carlo Tree Search algorithm from a presentation of the article [8]

**Descent.** The descent in $\hat{T}$ is done by considering that selecting a new node is equivalent to a $k$-armed bandit problem. In each node $s$ of the tree, the following information are stored:

- $n_s$: the total number of times the node $s$ has been selected.
- $\bar{x_s}$: the average reward for the node $s$.

The formula to select a new node $s'$ is based on the UCB formula 1. Let $C^s$ be the set of children of the node $s$:

$$s' \leftarrow \arg\max_{j \in C^s} \bar{x_j} + \sqrt{\frac{2ln(n_s)}{n_j}}$$

Once a new node has been selected, we repeat the same principle until we reach a situation $S$ outside of $\hat{T}$.

**Evaluation.** Now that we have reached a situation $S$ outside of $\hat{T}$. There is no more information available to take a decision; we can't, as in the tree, use the bandit formula. As we are not at a leaf of $T$, we can not directly evaluate $S$. Instead, we use a Monte-Carlo simulation to have a value for $S$. The Monte-Carlo simulation is done by selecting a new node (a child of $s$) using the heuristic function $mc(s)$ until a terminal node is reached. $mc(s)$ returns one element of $C^s$ based on a uniform distribution (in some cases, better distributions than the

uniform distribution are possible; we will consider uniformity here for Havannah, and the distribution in [16] for the game of Go).

**Growth.** In the growth step, we add the node $S$ to $\hat{T}$. In some implementations, the node $S$ is added to the node only after a finite fixed number of simulations instead of just 1; this number is 1 for our implementation for Havannah and 5 in our implementation for Go.

After adding $S$ to $\hat{T}$, we update the information in $S$ and in all the situations encountered during the descent with the value obtained with the Monte-Carlo evaluation (the numbers of wins and the numbers of losses are updated).

**Algorithm 1 Left.** MCTS($s$) **Right.** RMCTS($s$), including the poolRave modification.//$s$ a situation.

```
Initialization of T̂, n, x̄
while there is some time left do
  s' ← s
  Initialization of game
  //DESCENT//
  while s' in T̂ and s' not terminal do
    s' ← arg max_{j∈C^{s'}} [x̄_j + sqrt(2ln(n_{s'})/n_j)]
    game ← game + s'
  end while
  S = s'
  //EVALUATION//
  while s' is not terminal do
    s' ← mc(s')
  end while
  r = result(s')
  //GROWTH//
  T̂ ← T̂ + S
  for each s in game do
    n_s ← n_s + 1
    x̄_s ← (x̄_s * (n_s − 1) + r)/n_s
  end for
end while
```

```
Initialization of T̂, n, x̄, n^RAVE, x̄^RAVE
while there is some time left do
  s' ← s
  Initialization of game, simulation
  //DESCENT//
  while s' in T̂ and s' not terminal do
    s' ← arg max_{j∈C^{s'}} [x̄_j + α x̄^RAVE_{s',j} + sqrt(2ln(n_{s'})/n_j)]
    game ← game + s'
  end while
  S = s'
  //EVALUATION//
  //beginning of the poolRave modification //
  s'' ← last visited node in the tree with at
  least 50 simulations
  while s' is not terminal do
    if Random < p then
      s' ←one of the k moves with best RAVE
      value in s''
       /* this move is randomly and uniformly
      selected */
    else
      s' ← mc(s')
    end if
    simulation ← simulation + s'
  end while
  //end of the poolRave modification //
  //without poolRave, just s' ← mc(s')//
  r = result(s')
  //GROWTH//
  T̂ ← T̂ + S
  for each s in game do
    n_s ← n_s + 1
    x̄_s ← (x̄_s * (n_s − 1) + r)/n_s
    for each s' in simulation do
      n^RAVE_{s,s'} ← n^RAVE_{s,s'} + 1
      x̄^RAVE_{s,s'} ← (x̄^RAVE_{s,s'} * (n^RAVE_{s,s'} − 1) + r)/n^RAVE_{s,s'}
    end for
  end for
end while
```

### 2.3 Rapid Action Value Estimates

This section is only here for introducing notations and recalling the principle of rapid action value estimates; people who have never seen these notions are referred to [8] for more information. One generic and efficient improvement of the Monte-Carlo Tree Search algorithm is the RAVE values introduced in [3, 8]. In this section we note $f \rightarrow s$ the move which leads from a node $f$ to a node $s$ ($f$ is the father and $s$ the child node corresponding to move $m = f \rightarrow s$). The principle is to store, for each node $s$ with father $f$,

- the number of wins (won simulations crossing $s$ - this is exactly the number of won simulations playing the move $m$ in $f$);
- the number of losses (lost simulations playing $m$ in $f$);
- the number of AMAF[1] wins, i.e. the number of won simulations such that $f$ has been crossed and $m$ has been played after situation $f$ by the player to play in $f$ (but not necessarily in $f$!). In MCTS, this number is termed RAVE wins (Rapid Action Value Estimates);
- and the number of AMAF losses (defined similarly to AMAF wins).

The percentage of wins established with RAVE values instead of standard wins and losses is noted $\bar{x}_{f,s}^{RAVE}$. The total number of games starting from $f$ and in which $f \rightarrow s$ has been played is noted $n_{f,s}^{RAVE}$.

From the definition, we see that RAVE values are biased; a move might be considered as good (according to $\bar{x}_{f,s}$) just because it is good later in the game; equivalently, it could be considered as bad just because it is bad later in the game, whereas in $f$ it might be a very good move.

Nonetheless, RAVE values are very efficient in guiding the search: each Monte-Carlo simulation updates many RAVE values per crossed node, whereas it updates only one standard win/loss value. Thanks to these bigger statistics, RAVE values are said to be more biased but to have less variance.

Those RAVE values are used to modify the bandit formula 1 used in the *descent* part of the algorithm. The new formula to chose a new node $s'$ from the node $s$ is given below; let $C^s$ be the set of children of the node $s$.

$$s' \leftarrow \arg\max_{j \in C^s} [\bar{x_j} + \alpha \bar{\mathbf{x}}_{\mathbf{s,j}}^{\mathbf{RAVE}} + \sqrt{\frac{2\ln(n_s)}{n_j}}]$$

$\alpha$ is a parameter that tends to 0 with the number of simulations. When the number of simulations is small, the RAVE term has a larger weight in order to benefit from the low variance. When the number of simulations gets high, the RAVE term becomes small in order to avoid the bias. Please note that the right hand term $+\sqrt{\frac{2\ln(n_s)}{n_j}}$ exists in the particular case UCT; in many applications, the constant 2 is replaced by a much smaller constant or even 0; see [12] for more on this.

[1] AMAF=All Moves As First.

The modified MCTS algorithm with RAVE values is given in Alg. 1 (Right); it includes also the poolRave modification described below. The modifications corresponding to the addition of the RAVE values are put in bold and the poolRave modification is delimited by text.

## 3 PoolRave

The contribution of this paper is to propose a generic way to improve the Monte-Carlo simulations. A main weakness of MCTS is that choosing the right Monte-Carlo formula ($mc(.)$ in Alg. 1) is very difficult; the sophisticated version proposed in [16] made a big difference with existing functions, but required a lot of human expertise and work. We aim at reducing the need for such expertise. The modification is as follows: before using $mc(s)$, and with a fixed probability $p$, try to choose one of the $k$ best moves according to RAVE values. The RAVE values are those of the last node with at least 50 simulations. We will demonstrate the generality of this approach by proposing two different successful applications: the classical application to the game of Go, and the interesting case of Havannah in which far less expertise is known.

## 4 Experiments

We will consider (i) Havannah (section 4.1) and then the game of Go (section 4.2).

### 4.1 Havannah

We will briefly present the rules and then our experimental results.

The game of Havannah is a two-player game created by Christian Freeling. The game is played on a hexagonal board with hexagonal locations. It can be considered as a connection game, like the game of Hex or Twixt. The rules are very simple. White starts, and after that each player plays alternatively. To win a game a player has to realize one of these three shapes :

- A ring, which is a loop around one or more cells (empty or not, occupied by black or white stones).
- A fork, which is a continuous string of stones that connects three of the six sides of the board (corner locations are not belonging to the edges).
- A bridge, which is a continuous string of stones that connects one of the six corners to another one.

An example of these three winning positions is given in Fig. 2.

The game of Havannah is specially difficult for computers, for different reasons.

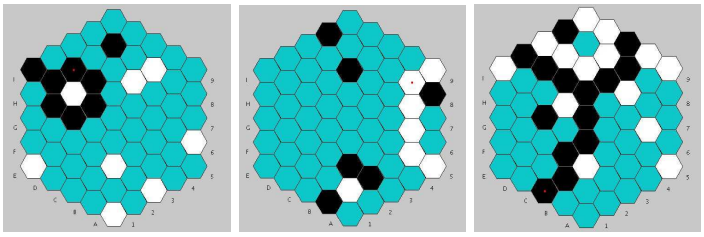

**Fig. 2.** Three finished games: a ring (a loop, by black), a bridge (linking two corners, by white) and a fork (linking three edges, by black).

- First, due to the large action space. For instance, in size 10 (10 locations per edges) there are 271 possible moves for the first player.
- Second, there is no pruning rule for reducing the tree of the possible futures.
- Third, there is no natural evaluation function.
- Finally, the lack of expert knowledge for this game.

The efficiency of the MCTS algorithm on this game has been shown recently in [15]. As far as we know, nowadays, all the robots which play this game use an MCTS algorithm. In their paper, they also have shown the efficiency of the RAVE formula.

The experimental results are presented in Table 1.

| # of simulations | Value of $p$ | Size of the pool | Success rate against the baseline |
|---|---|---|---|
| 1000 | 1/2 | 5 | 52.7±0.62% |
| 1000 | 1/2 | 10 | 54.32±0.46% |
| 1000 | 1 | 10 | 52.42±0.70% |
| 1000 | 1/4 | 10 | 53.19±0.68% |
| 1000 | 3/4 | 10 | 53.34±0.85% |
| 1000 | 1 | 20 | 53.2±0.8% |
| 1000 | 1/2 | 20 | 52.51±0.54% |
| 1000 | 1/4 | 20 | 52.13±0.55% |
| 1000 | 3/4 | 20 | 52.9±0.34% |
| 10000 | 1/2 | 10 | 54.45±0.75% |
| 20000 | 1/2 | 10 | 54.42±0.89% |

**Table 1.** Success rate of the poolRave modification for the game of Havannah. The baseline is the code without the poolRave modification.

We experiment the modification presented in this paper for the game of Havannah. We measure the success rate of our bot with the new modification against the baseline version of our bot. There are two different parameters to tune : (i) $p$ which is the probability of playing a modified move and (ii) the size of the pool. We have experimented with different numbers of simulations in order to see the robustness of our modification. Results are shown in table 1. The best

results are obtained with $p = 1/2$ and a pool size of 10, for which we have a success rate of 54.32% for 1000 simulations and 54.45% for 10000 simulations. With the same set of parameters, for 20000 simulations we have 54.42%, so for the game of Havannah this improvement seems to be independent of the number of simulations.

### 4.2 Go

The game of Go is a classical benchmark for MCTS; this Asian game is probably the main challenge in games and a major testbed for artificial intelligence. The rules can be found on `http://senseis.xmp.net`; roughly, each player puts a stone of his color in turn, groups are maximum sets of connected stones for 4-connectivity, groups that do not touch any empty location are "surrounded" and removed from the board; the player who surround the bigger space with his stones has won. Computers are far from the level of professional players in Go, and the best MCTS implementations for the game of Go use sophisticated Monte-Carlo Tree Search.

The modification proposed in this article is implemented in the Go program MoGo. The probability of using the modification $p$ is useful in order to preserve the diversity of the simulations. As, in MoGo, this role is already played by the "fillboard" modification [4], the probability $p$ is set to 1. The experiments are done by making the original version of MoGo play against the version with the modification on 9x9 games with 1000 simulations per move.

We obtain up to $51.7 \pm 0.5\%$ of victory. The improvement is mathematically significant but not very important. The reason is that Monte Carlo simulations in the program MoGo possess extensive domain knowledge in the form of patterns. In order to measure the effect of our modification in applications where no knowledge is available, we run more experiments with a version of MoGo without patterns. The results are presented in table 2.

| Size of the pool | Success rate against the baseline |
|---|---|
| 5 | 54.2±1.7% |
| 10 | 58.7±0.6% |
| 20 | 62.7±0.9% |
| 30 | 62.7±1.4% |
| 60 | 59.1±1.8% |

**Table 2.** Success rate of the poolRave modification for the game of Go. The baseline is the code without the poolRave modification. This is in the case of no patterns in the Monte-Carlo part.

When the size of the pool is too large or not large enough, the modification is not as efficient. When using a good compromise for the size (20 in the case of MoGo for 9x9 go), we obtain $62.7 \pm 0.9\%$ of victory.

It is also interesting to note that when we increase the number of simulations per move, we obtain slightly better results. For example, with 10000 simulations per move, we obtain $64.4 \pm 0.4\%$ of victory.

## 5 Conclusion

We presented a generic way of improving the Monte-Carlo simulations in the Monte-Carlo Tree Search algorithm. This method is based on already existing values (the RAVE values) and is easy to implement.

We show two different applications where this improvement was successful: the game of Havannah and the game of Go. For the game of Havannah, we achieve 54.3% of victory against the version without the modification. For the game of Go, we achieve only 51.7% of victory against the version without modification. However, without the domain-specific knowledge, we obtain up to 62.7% of victory.

In the near future, we intend to use an evolution algorithm in order to tune the different parameters. We will also try different ways of using these values in order to improve the Monte-Carlo simulations. We strongly believe that the next step in improving the MCTS algorithm will be reached by finding an efficient way of modifying the Monte-Carlo simulations depending on the context.

## 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. A. Auger and O. Teytaud. Continuous lunches are free plus the design of optimal optimization algorithms. *Algorithmica*, Accepted.
3. B. Bruegmann. Monte-carlo Go (unpublished draft http://www.althofer.de/bruegmann-montecarlogo.pdf). 1993.
4. G. Chaslot, C. Fiter, J.-B. Hoock, A. Rimmel, and O. Teytaud. Adding expert knowledge and exploration in Monte-Carlo Tree Search. In *Advances in Computer Games*, Pamplona Espagne, 2009. Springer.
5. 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.
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*, pages 72–83, 2006.
7. 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.
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. D. Knuth and R. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.
10. L. Kocsis and C. Szepesvari. Bandit based Monte-Carlo planning. In *15th European Conference on Machine Learning (ECML)*, pages 282–293, 2006.
11. T. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6:4–22, 1985.
12. 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.
13. W.-B. Powell. *Approximate Dynamic Programming*. Wiley, 2007.
14. 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.
15. F. Teytaud and O. Teytaud. Creating an Upper-Confidence-Tree program for Havannah. In *ACG 12*, Pamplona Espagne, 2009.
16. 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.