Learning a Move-Generator for Upper Con dence Trees

Adrien Couetoux, Olivier Teytaud, Hassen Doghmen

Learning a Move-Generator for Upper Confidence Trees

Adrien Couetoux^{1,2}, Olivier Teytaud^{1,2,3}, Hassen Doghmen^{1}

^{1} TAO-INRIA, LRI, CNRS UMR 8623,
Université Paris-Sud, Orsay, France
^{2} OASE Lab, National University of Tainan, Taiwan
^{3} Montefiore Institute, Université de Liège, Belgium

Abstract. We experiment the introduction of machine learning tools to improve Monte-Carlo Tree Search. More precisely, we propose the use of Direct Policy Search, a classical reinforcement learning paradigm, to learn the Monte-Carlo Move Generator. We experiment our algorithm on different forms of unit commitment problems, including experiments on a problem with both macrolevel and microlevel decisions.

1 Introduction

Monte-Carlo Tree Search (MCTS) [5] is a versatile algorithm for Markov Decision Processes (MDP) or games. It is elegant (can be described in a few lines), versatile (can be applied in many settings such as MDP, games, stochastic games), and can work with or without expert information in the playouts (hence a great success in general game playing[14]). It uses the framework of bandits[9] which does not depend on a particular application. It is moderately efficient when the number of time steps is big, but surprisingly stable on high-dimensional problems.

It has been greatly improved by including Progressive Widening and Double Progressive Widening[6, 2], RAVE values[7], Blind Values[4], and handcrafted Monte-Carlo moves[17, 10]. A crucial component is the Monte-Carlo move generator, also known as the playout generator.

In this paper, we focus on the addition of specialized Monte-Carlo moves, i.e. we modify default policy, to help dealing with stochastic planning problems. Finding a default policy that is optimal for all instances of a stochastic problem can be extremely difficult and time consuming. The solution we propose here is to apply a Direct Policy Search to the available default policy. This way, even an initially poor default policy can be improved to fit different instances of one stochastic planning problem.

In Section 2 we describe existing algorithms (Monte-Carlo Tree Search in Section 2.1, and existing algorithms for improving Monte-Carlo move generators in Section 2.2). We also introduce Direct Policy Search for improving a Monte-Carlo move generator in Section 2.3. In Section 3, we experiment our algorithms on three forms of a unit commitment problem (Section 3.1), and on an investment problems (Section 3.2).

2 Algorithms

In this section we will present the vanilla Monte-Carlo Tree Search algorithm (Section 2.1), Direct Policy Search, and Monte-Carlo move generators improvements (Sections 2.2 and 2.3).

2.1 Monte-Carlo Tree Search and Upper Confidence Trees

Monte-Carlo Tree Search (here presented in the framework of a MDP) consists in simulating plenty of series of decisions as long as we have time before choosing an action, and keeping statistics of these games. What follows, is the formal description of the state of the art continuous MCTS, i.e. MCTS with Double Progressive Widening (MCTS-DPW), as seen in [3]. As the reader can see, it mainly requires two things: (i) *a transition function*, capable of simulating what happens when an action a is taken in state s, and returns a new state s' and a reward r. (ii) *a default policy* φ, capable of returning an action a, given a state s. When nothing is specified, it is assumed that this function returns a random action, following a random distribution that covers the entire set of feasible actions in state s.

```
MCTS algorithm with DPW and default policy $\varphi$
Input: a state $S$.
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  and $nbSims(s) > 0$ do
      Apply DPW in state $s$.
      Let $s'$ be the state given by DPW.
      $s = s'$
   end while
   while $s$ is not a terminal state // {happens when a non final and new state $s$ is
   visited} do
      Choose action $a$, according to $\varphi$
      Simulate action $a$; get a new state $s'$
      $s = s'$
   end while
   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 $S$.
```

The algorithm therefore relies on a Monte-Carlo move generator, also called default policy, φ. The default policy can be a simple random uniform generator (when no expertise is available for making more reasonable simulations), but handcrafted functions can perform better.

```
**Action Selection by Double Progressive Widening (DPW), applied in state $s$ with constants $C > 0$, $\alpha \in ]0,1[$, and $\beta \in ]0,1[$.**
Input: a state $s$.
Output: a state $s'$.
Let $nbVisits(s) \leftarrow nbVisits(s) + 1$
and let $t = nbVisits(s)$
Let $k = \lceil Ct^{\alpha} \rceil$.
Let $(o_i(s))_{i \geq 1}$ be the feasible actions in state $s$.
Choose an action $a_t(s) \in \{o_1(s), \ldots, o_k(s)\}$ maximizing $score_t(s,a)$ defined as follows:
 $totalReward_t(s,a) = \sum_{1 \leq l \leq t-1, a_l(s)=a} r_l(s)$
 $nb_t(s,a) = \sum_{1 \leq l \leq t-1, a_l(s)=a} 1$
  $score_t(s,a) = \frac{totalReward_t(s,a)}{nb_t(s,a)+1} + k_{ucb}\sqrt{\log(t)/(nb_t(s,a)+1)}$  ($+\infty$ if $nb_t(a) = 0$)
Let $k' = \lceil Cnb_t(s, a_t(s))^{\beta} \rceil$
if $k' > \#Children_t(s, a_t(s))$  // {progressive widening on the random part}
then
  Simulate action $a_t(s)$; get a new state $s'$
  if $s' \notin Children_t(s, a_t(s))$ then
    $Children_{t+1}(s, a_t(s)) = Children_t(s, a_t(s)) \cup \{s'\}$
  else
    $Children_{t+1}(s, a_t(s)) = Children_t(s, a_t(s))$
  end if
else
  $Children_{t+1}(s, a_t(s)) = Children_t(s, a_t(s))$
  Choose $s'$ in $Children_t(s, a_t(s))$ // $s'$ is chosen with probability $nb_t(s, a_t(s), s')/nb_t(s, a_t(s))$
end if
```

2.2 Heuristics and Monte-Carlo move generators

Whereas in the 2-player case, it is known that making a Monte-Carlo generator stronger (stronger in the sense: as a stand-alone policy), does not necessarily make the MCTS built on top of it stronger (see [17]), we conjecture that in the one-player case it is usually quite efficient.

The recent improvements in the world of Computer Go basically comes from improvements of the Monte-Carlo move generator, implemented so that the Monte-Carlo simulator does not contradict life&death known results; Zen, CrazyStone, Pachi, are examples of such strong programs, around 2 Dan for short time settings and 4 Dan for long time settings. Other tools have been proposed as generic solutions for learning Monte-Carlo move generators:

– Simulation balancing [15, 8] has been proposed for automatically learning the Monte-Carlo move generator in 2-player games.
– PoolRave[13], in which the Monte-Carlo move is replaced, with a fixed probability $p \in (0, 1)$, by a move uniformly drawn among the c moves with best RAVE score in the last node of the simulation with at least k simulations.

– Contextual Monte-Carlo[12] in which the Monte-Carlo move-generator is improved by online learning a tile-based value function.

These tools are efficient, but the main successes in Monte-Carlo Tree Search nonetheless come from handcrafted Monte-Carlo move generators. In this paper, we used a specialized Monte-Carlo move generator, chosen specifically on our main target problem, as well as a less specialized function. We improve them by Direct Policy Search (Section 2.3).

2.3 Direct Policy Search for generating Monte-Carlo Move Generators

Direct Policy Search (DPS) is an approach very different from Upper Confidence Tree; it is based on selecting a policy among a parametric family of policies by optimization of its parameters. The pseudo-code is as follows:

```
Procedure Simulate($s, MDP, p$):
Inputs: a state $s$, a Markov Decision Process $MDP$, and a policy $p$.
Output: a reward.
Method: simulate $MDP$ from state $s$ with policy $p$ until a terminal state and return the obtained reward.
Procedure Direct Policy Search:
Inputs: (i) a parametric policy $\theta \mapsto \pi(\theta)$, where $\pi(\theta)$ is a mapping from states to actions. (ii) a Markov Decision Process $MDP$. (iii) an initial state $s$.
Output: a parameter $\hat{\theta}$, leading to a policy $\pi(\hat{\theta})$.
Auxiliary method: a noisy optimization algorithm.
Apply the noisy optimization algorithm to the function $\theta \mapsto Simulate(s, MDP, \pi(\theta))$; get $\hat{\theta}$ the approximate optimum.
Return $\hat{\theta}$.
```

Direct Policy Search is usually applied offline, i.e. a single $\hat{\theta}$ is obtained once and for all. However, optimizing Θ to maximize $\theta \mapsto Simulate(s, MDP, \pi(\theta))$ specifically for the current state s for which we look for a decision is possible. We apply DPS and use the obtained policy $\pi(\hat{\theta})$ as a Monte-Carlo move generator in our MCTS.

The paper in [1] proposes to apply DPS (the terminology in the paper is different, but it is essentially DPS) based on a heuristic function obtained by experts, by smoothing the heuristic and adding parameters in it (the smoothing is here for making the problem easier to optimize). This is our approach in the rest of this paper, except that we do not smooth policies as the randomized nature of our problems make the objective function smooth enough. We use self-adaptation[11] as a noisy optimization algorithm. As a summary, our algorithm is as follows for choosing a move in state s within time t:

```
Procedure OptimisticHeuristics($s, \phi, t, MDP$)
Input: a state $s$, a time $t$, a parametric family of policies $\phi_\theta$.
Output: an action $a$.
Apply DPS with time budget $t/2$ for choosing $\hat{\theta}$ (use warm start if possible)
Apply MCTS with time budget $t/2$ for choosing action $a$.
```

3 Experiments

Here, we compare the performances of different sequential decision making algorithms. Namely, we implemented vanilla MCTS, MCTS with a fixed default policy, MCTS with a default policy improved online by DPS, and DPS alone.

We made experiments on three different forms of the unit commitment problem, and on a more general energy management problem called bilevel.

3.1 Unit commitment problem

We work on a stock management problem, from [16].

The main points in the problem are that: (1) Unit Commitment problems can not be solved efficiently by traditional methods; these problems are usually simplified so that classical methods, like Bellman's stochastic dynamic programming, can be applied. The motivation of our work on Unit Commitment by Monte-Carlo Tree Search methods is that we want to work without simplifying too much the model. (2) Unit Commitment problems exist at many time scales (from milliseconds, up to years for hydroelectric stocks or tenths of years if investments are included) and many dimensionalities (from a few stocks to thousands of state variables), depending on the scope under analysis. We here work on small scale problems for the sake of statistical significance (working on our full problems requires by far too much time for reproducing runs tenths of times).

In this paper, we will consider three variants of the unit commitment problem. The significant difference between these three variants is the way the stocks are connected. In the first one, they are lined up on a one dimension chain (we will call it the one river problem). In the second one, they are linked so that they form a binary tree, with the root being the last stock that the water goes through (we will call it the "binary rivers problem"). Finally, the third one is simply a random arrangement of the stocks, with one single constraint: no cycles are allowed.

Two different heuristics for the Unit commitment problem. The expert parametrized heuristic that we use has been designed using knowledge about the problem, to make it particularly efficient on the one river variant of the unit commitment problem. On the other hand, the naive heuristic uses almost no knowledge about the problem. Given a state s, it requires the current time to go t, and the average demand at the current time step D_{avg}. We provide below pseudo-codes of both heuristics.

Expert heuristic
Input: a state S, of dimension N.
Parameter: a vector θ of dimension 3. Default value is $[1, 0, 1]$.
Information required from the problem:

- $D(t)$: expected electricity demand during time step t.
- $D_{timeToGo}(t)$: expected total demand after time step t.
- $TSA(s, timeToGo)$: total stock available (this assume a 1 river structure).
- $TI(timeToGo, averageInflows)$: expected total usable water from inflows (this assume a 1 river structure).

Output: an action a.

```
1. initialize:
     – total water available = $TWA = (\theta_0 + \theta_1 \times timeToGo) \times (TSA + TI)$
     – $x = production\ by\ hydroelectricity = 0$
     – $increaseWater \leftarrow true$
     – $S_{available} = \sum_{0 \leq i \leq N-1} s_i$
2. while ($increaseWater$ and $x < S_{available}$)  $s_i$ being the current level of stock $i$ do
     define the marginal cost $mc$ of increasing water, approximated as
         $mc = IC(x, s, t, D(t)) + \theta_2 \times LTC(x, TWA, t, D_{timeToGo}(t))$
     where:
     –  $IC(x, s, t, D(t))$ is negative; it is the marginal benefit associated to the reduction of thermal production.
     –  $LTC(\dots)$ is the sum of thermal production cost, if expected total demand $D_{timeToGo}$, decreased by the total production from the water stocks if equally distributed on the time steps to go, is produced thermally.
     if marginal cost $mc > 0$ then
         then $increaseWater \leftarrow false$
     else
         $x \leftarrow x + 1$.
     end if
3. end while
4. Compute $q$, the ratio $\min(0, \frac{x}{S_{available}})$
5. Return the action vector $a$ defined as follows: $\forall 0 \leq i \leq N-1, a_i = q.L_i$
```

Naive heuristic, polynomial with degree m
Input: a state S, of dimension N.
Parameter: a vector θ of dimension $m + 1$. Default values are $[1, 0, \dots, 0]$
Information required from the problem: t the remaining time steps, and D_{avg} the average demand after the current time step
Output: an action a.

```
1. Compute total amount of water to use $W_{use} = \max(0, D_{avg}.(\theta_0 + \theta_1 t + \cdots + \theta_{m+1} t^m))$
2. Given $S$ and the current level $L_i$ of each stock $i$, $W_{available} = \sum_{0 \leq i \leq N-1} L_i$
3. Compute $q$, the ratio $\min(0, \frac{W_{use}}{W_{available}})$
4. Return the action vector $a$ defined as follows: $\forall 0 \leq i \leq N-1, a_i = q.L_i$
```

Experimental results on the Unit Commitment problem. We present here the results obtained on all three variants of the unit commitment problem. Each time, we compared the following algorithms: (i) vanilla MCTS, as presented in Section 2.1, (ii) MCTS-naive, a MCTS using the naive heuristic as a default policy,(iii) MCTS-expert, a MCTS using the expert heuristic as a default policy,

(iv) MCTS-naive-DPS, a MCTS using the naive heuristic improved by DPS, (v) MCTS-expert-DPS, a MCTS using the expert heuristic improved by DPS and when relevant, (vi) the non tuned naive and expert heuristics.

The x axis shows the time budget allocated per decision made, in logarithmic scale, and went from 0.01 second to 2.56 second. The y axis shows the average reward. Each average reward was computed using 1000 runs. Error bars show the 95% confidence intervals. The higher the reward, the better the algorithm performed. It should be noted that the rewards cannot be compared between different variants of the unit commitment problem. Indeed, only the connections between the stocks change, and changing this changes the amount of water effectively available. Our results for the 1-river problem and for the binary rivers problem are shown on the left side and the right side of Fig. 1, respectively. We did not plot the results of the naive heuristic, that scored -150000 and -380000 respectively (far below other methods), for the sake of readability. In both experiment, MCTS-naive-DPS and MCTS-expert-DPS outperform by a factor of at least 100 the third placed algorithm, MCTS-expert. MCTS-naive and MCTS vanilla share the fourth and fifth places.

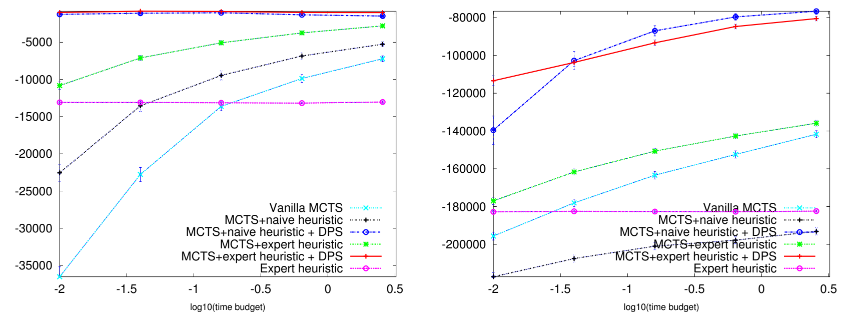

Fig. 1. Performances of different variants of MCTS on the 1 river unit commitment problem (left) and the binary rivers (right), with 7 stocks, 24 time steps. Y axis shows the reward (the higher the better).

Our results on the random rivers problem are shown in Fig. 2. In this experiment, the most significant difference in the results is that MCTS-naive-DPS is about 10 times faster than MCTS-expert-DPS.

Over all three versions of the unit commitment problem, the most efficient and robust version has been MCTS-naive-DPS. Even on the one river problem, that the expert heuristic was particularly well tuned for, we could not see huge benefits from using it as a parametric function for DPS.

3.2 Experiments on the investment problem

In this section we experiment our algorithm on a problem (simplified from [16]) as follows:

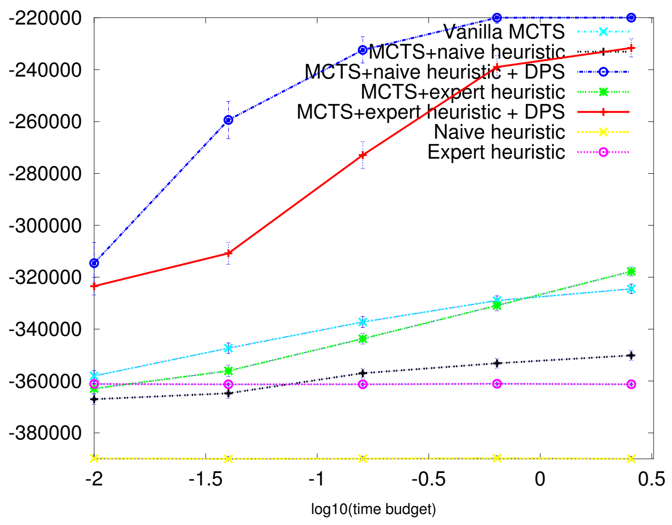

Fig. 2. Performances of different variants of MCTS on the randomly connected unit commitment problem (7 stocks, 24 time steps).Y axis shows the reward (the higher the better).

– At each time step, we decide investments; there is a limited amount of money to invest, and investments must be distributed over 7 different possible infrastructures. There are therefore 7 decision variables for each time step.
– At each time step, a lower level problem (the management of the energy production system) is built and solved, and its cost is the cost of the current transition of the investment problems.
– There are 10 time steps, the last one has a strong influence because it reflects the long-term.

Our results compare the following strategies:

– a heuristic which gives a constant ratio of the investment on each possible infrastructure (the parameters of the heuristic are this proportions); the default parametrization is the same ratio for all infrastructures;
– DPS on a "sum of Gaussians" policy (parameters: positions of the Gaussians, widths, associated decisions; see [16];
– DPS on a "neural network" policy (parameters: weights, theresholds; see [16]);
– DPS on a "sum of Gaussians" policy, added to the heuristic with default parametrization;
– DPS on a "neural network" policy, added to the heuristic with default parametrization;
– MCTS, on top of each of the above.

Results are presented in Fig. 3.

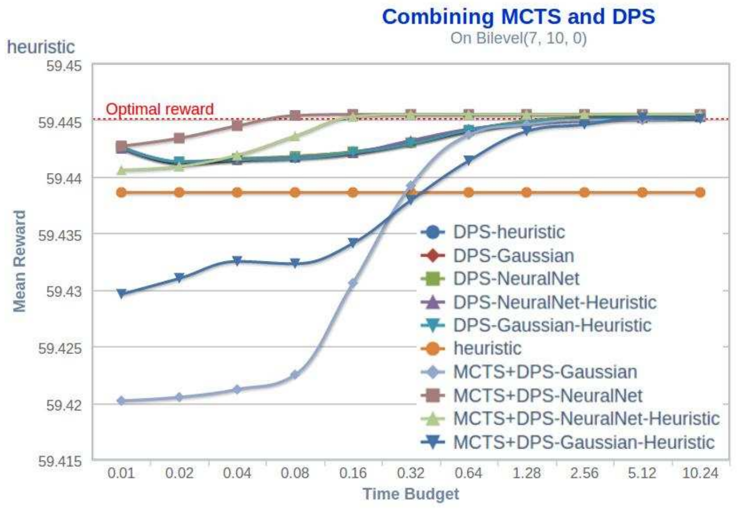

Fig. 3. Results on the energy investment problem. The five DPS curves (curves 1 to 5) are very close to each other; results are better than for the heuristic alone, and versions without the heuristic are almost the same as versions with the heuristic. The MCTS+DPS+neural network was the most efficient strategy, outperforming MCTS+DPS+neural network+heuristic. The sums of Gaussians require more time for learning, hence the poor results for moderate budgets.

4 Conclusion

We combine DPS and MCTS. The DPS provides the Monte-Carlo simulator of the MCTS. The resulting algorithm, has no free parameter and outperforms by far the vanilla MCTS. We use human expertise at two levels: (i) For partial observation handling, i.e. the belief state estimation was handcrafted, so that the problem is essentially a MDP rather than a partially observable MDP. The details of this are beyond the scope of this paper. (ii) In the Monte-Carlo move generator, because in spite of nice and interesting efforts in the literature, no generic algorithm, in the current state of the art, can define a Monte-Carlo move generator as efficiently as a human expert (in the case of Go, but also in the case of unit commitment problems). Nonetheless our DPS could strongly improve the heuristic by optimizing its parameters. We agree with the traditional statement that MCTS is surprisingly efficient when no human expertise is available, but we clearly see that human expertise was an easy key for a speed-up 100, as well as human expertise is the key of recent progress in MCTS for the classical challenge of the game of Go.

Importantly, the need for human expertise is considerably reduced by the use of DPS for optimizing the heuristics, so that our results are a step towards generic MCTS tools.

References

1. Y. Bengio. Using a financial training criterion rather than a prediction criterion. CIRANO Working Papers 98s-21, CIRANO, 1998.
2. 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.
3. A. Couetoux, J.-B. Hoock, N. Sokolovska, O. Teytaud, and N. Bonnard. Continuous Upper Confidence Trees. In *LION'11: Proceedings of the 5th International Conference on Learning and Intelligent OptimizatioN*, page TBA, Italie, Jan. 2011.
4. A. Couetoux, O. Teytaud, and H. Doghmen. Improving the exploration in upper confidence trees. *LION 6, LNCS 7219 proceedings*, pages 366–371, 2012.
5. 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.
6. R. Coulom. Computing elo ratings of move patterns in the game of go. In *Computer Games Workshop, Amsterdam, The Netherlands*, 2007.
7. 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.
8. S.-C. Huang, R. Coulom, and S.-S. Lin. Monte-carlo simulation balancing in practice. In H. 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.
9. L. Kocsis and C. Szepesvari. Bandit based Monte-Carlo planning. In *15th European Conference on Machine Learning (ECML)*, pages 282–293, 2006.
10. 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.
11. S. Meyer-Nieberg and H.-G. Beyer. Self-adaptation in evolutionary algorithms. In F. G. Lobo, C. F. Lima, and Z. Michalewicz, editors, *Parameter Setting in Evolutionary Algorithms*. Springer, Berlin, 2007.
12. A. Rimmel and F. Teytaud. Multiple Overlapping Tiles for Contextual Monte Carlo Tree Search. In *Evostar*, Istanbul Turquie.
13. A. Rimmel, F. Teytaud, and O. Teytaud. Biasing Monte-Carlo Simulations through RAVE Values. In *The International Conference on Computers and Games 2010*, Kanazawa Japon, 05 2010.
14. 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.
15. D. Silver and G. Tesauro. Monte-carlo simulation balancing. In A. P. Danyluk, L. Bottou, and M. L. Littman, editors, *ICML*, volume 382 of *ACM International Conference Proceeding Series*, page 119. ACM, 2009.
16. O. Teytaud. Including Ontologies in Monte-Carlo Tree Search and Applications - an Open Source Platform, 2008.
17. 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.