Bandit-Based Genetic Programming with Application to Reinforcement Learning

J.-B Hoock, O Teytaud

Bandit-Based Genetic Programming with Application to Reinforcement Learning

J.-B. Hoock and O. Teytaud

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

Abstract :

When looking for relevant mutations of a learning program, a main trouble is that evaluating a mutation is noisy; we can have a precise estimate of a mutation, if we test it many times, but this is quite expensive; or we can have a rough estimate, which is much faster. This is a load balancing problem: on which mutations should we spend more effort ?

Bandit algorithms have been used for this load balancing: they choose the computational effort spent on various possible mutations, depending on the current estimate of the quality of a mutation and on the precision of this estimate. However, in many cases, we want to validate some possible mutations; when should we stop the bandit mutation, and analyze new mutations ? Racing algorithms are aimed at combining the load balancing and the statistical validation; we here mathematically analyze and experiment racing algorithms in the context of mutations of programs, i.e. genetic programming.

As an application, we consider Monte-Carlo Tree Search. Monte-Carlo Tree Search is a recent very successful algorithm for reinforcement learning, successfully applied in games and Markov decision Processes. We consider the validation of randomly generated patterns in a Monte-Carlo Tree Search program. Our bandit-based genetic programming (BGP) algorithm, with proved mathematical properties, outperformed a highly optimized handcrafted module of a well-known computer-Go program with several world records in the game of Go.

Mots-clés : Reinforcement learning, Monte-Carlo Tree Search, Genetic Programming, Bernstein races.

1 Introduction

Genetic Programming (GP) is the automatic building of programs for solving a given task. In this paper, we investigate a bandit-based approach for selecting fruitful modifications in genetic programming, and we apply the result to our reinforcement learning program MoGo.

When learning patterns in a reinforcement learning algorithms with limited ressources in an uncertain framework, there are two issues:

- which modifications of the policy are to be tested now ?
- when we have no more resources (typically no more time), we must decide which modifications are accepted.

The second issue is often addressed through statistical tests. However, when many modifications are tested, it is a problem of multiple simultaneous hypothesis testing: this is far from being straightforward; historically, this was poorly handled in many old applications. CournotDesrosières (2000) stated that if we consider a significance threshold of 1% for differences between two sub-populations of a population, then, if we handcraft plenty of splittings in two sub-populations, we will after a finite time find a significant difference, whenever the two populations are similar. This was not for genetic programming, but the same thing holds in GP: if we consider 100 random mutations of a program, all of them being worst than the original program, and if we have a 1% risk threshold in the statistical validation of each of them, then with probability $(1-1/100))^{100} \simeq 37\%$ we can have a positive validation of at least one harmful mutation. Cournot concluded, in the 19th century, that this effect was beyond mathematical analysis; nonetheless this effect is clearly understood today, with the theory of multiple hypothesis testing - papers cited below clearly show that mathematics can address this problem.

The first issue is also non trivial, but a wide literature has been devoted to it: so-called bandit algorithms. This is in particular efficient when no prior information on the modifications is available, and we can only evaluate the quality of a modification through statistical results. Whereas bandits handle the first problem, and multiple simultaneous hypotehsis testing handles both cases, races are aimed at handling both problems simultaneously. Races can be based on arbitrary confidence intervals; however, in the general case (i.e. when variances might be small or not), the best tool is usually Bernstein's confidence intervals. However, when variance is never small, Hoeffding's bounds are equivalent and simpler.

Usually the principles of a Bernstein race are as follows:

- decide a risk threshold δ_0;
- then, modify the parameters of all statistical tests so that all confidence intervals are *simultaneously* true with probability $\geq 1 - \delta_0$;
- then, as long as you have computational resources, apply a *bandit* algorithm for choosing which modification to test, depending on statistics; typically, a bandit algorithm will choose to spend computational resources on the modification which has the best statistical upper bound on its average efficiency;
- at the end, select the modifications which are significant.

A main reference, with theoretical justifications, is Mnih *et al.* (2008). A main difference here is that we will not assume that all modifications are cumulative: here, whenever two modifications A and B are statistically good, we can't select both modifications - maybe, the baseline + A + B will be worse than the baseline, whenever both baseline+A and baseline+B are better than the baseline. Also, in Mnih *et al.* (2008), the

authors have no baseline; we are not aware of a framework which exactly matches our case (see however Even-Dar *et al.* (2006)).

In section 2, we present non-asymptotic confidence bounds. In section 3 we present racing algorithms. Then, section 4 presents our algorithm and its theoretical analysis. Section 5 is devoted to experiments.

2 Non-asymptotic confidence bounds

In all the paper, we consider fitness values between 0 and 1 for simplifying the writing. The most classical bound is Hoeffding's bound. Hoeffding's bound states that with probability at least $1-\delta$, the empirical average $\hat{r}$ verifies $|\hat{r} - Er| \leq deviation_{\text{Hoeffding}}(\delta, n)$ where n is the number of simulations and where

$$deviation_{\text{Hoeffding}}(\delta, n) = \sqrt{\log(2/\delta)/(2n)}. \quad (1)$$

Audibert *et al.* (2006); Mnih *et al.* (2008); Heidrich-Meisner & Igel (2009) have shown the efficiency of using Bernstein's bound instead of Hoeffding's bound, in some settings. The bound is then:

$$deviation_{\text{Bernstein}} = \hat{\sigma}\sqrt{2\log(3/\delta)/n} + 3\log(3/\delta)/n \quad (2)$$

where $\hat{\sigma}$ is the empirical standard deviation. Bernstein's version will not be used in our experiments, because the variance is not small in our case; nonetheless, all theoretical results also hold with Bernstein's variant.

3 Racing algorithms

Racing algorithms are typically (and roughly, we'll be more formal below) as follows:

Let S be equal to S_0, some given set of admissible modifications.
while $S \neq \emptyset$ **do**
  Select $s = select() \in S$ with some algorithm
  Perform one Monte-Carlo evaluation of s.
  if s is statistically worse than the baseline **then**
    $S \leftarrow S \setminus \{s\}$ — s is discarded
  else if s is statistically better than the baseline **then**
    Accept s; $S \leftarrow S \setminus \{s\}$ — s is accepted
  end if
end while

With relevant statistical tests, we can ensure that this algorithm will select all "good" modifications (to be formalized later), reject all bad modifications, and stop after a finite time if all modifications have a non-zero effect. We refer to Mnih *et al.* (2008) for more general informations on this, or Koza (1992); Holland (1973) for the GP case; we will here focus on the most relevant (relevant for our purpose) case. In genetic programming,

it's very clear that even if two modifications are, independently, good, the combination of these two modifications is not necessarily good. We will therefore provide a different algorithm in section 4 with a proof of consistency.

4 Theoretical analysis for genetic programming

We will assume here that for a modification s, we can define:

- $e(s)$, the (of course unknown) expected value of the reward when using modification s. This expected value is termed the efficiency of s. We will assume in the sequel that the baseline is 0.5 - an option is good if and only if it performs better than 0.5, and the efficiency is the average result on experiments.
- $n(s)$, the number of simulations of s already performed.
- $r(s)$ the total reward of s, i.e. the sum of the rewards of the $n(s)$ simulations with modification s.
- $ub(s)$, an upper bound on the efficiency of s, to be computed depending on the previous trials ($ub(s)$ will be computed thanks to Bernstein bounds or Hoeffding bounds).
- $lb(s)$, a lower bound on the efficiency of s (idem).

The two following properties will be proved for some specific functions lb and ub; the results around our BGP (bandit-based genetic programming) algorithm below hold whenever lb and ub verify these assumptions.

- **Consistency:** with probability at least $1 - \delta_0$, for all calls to ub and lb, the efficiency of s is between $lb(s)$ and $ub(s)$:

$$e(s) \in [lb(s), ub(s)]. \tag{3}$$

- **Termination:** when the number of simulations of s goes to infinity, then

$$ub(s) - lb(s) \to 0. \tag{4}$$

These properties are exactly what is ensured by Bernstein's bounds or Hoeffding's bounds. They will be proved for some variants of ub and lb defined below (Lemma 4.1, using Hoeffding's bound); they will be assumed in results about the BGP algorithm below. Therefore, our results about BGP (Theorem 4.2) will hold for our variants of lb and ub. Our algorithm and proof do not need a specific function ub or lb, provided that these assumptions are verified. However, we precise below a classical form of ub and lb, in order to point out that there exists such ub and lb; moreover, they are easy to implement. lb and ub are computed by a function with a memory (*i.e.* with static variables):

Function $computeBounds(s)$ (variant 1)
Static internal variable: $nbTest(s)$, initialized at 0.
Let n be the number of times s has been simulated.
Let r be the total reward over those s simulations.
$nbTest(s) = nbTest(s) + 1$
Let $lb(s) = r/n - deviation_{\text{Hoeffding}}\left(\delta_0/(\#S \times 2^{nbTest(s)}), n\right)$.
Let $ub(s) = r/n + deviation_{\text{Hoeffding}}\left(\delta_0/(\#S \times 2^{nbTest(s)}), n\right)$.

What is important in these formula is that the sum of the $\delta_0/(\#S \times 2^{nbTests(s)})$, for $s \in S$ and $nbTest(s) \in \{1, 2, 3 \ldots\}$, is at most δ_0. By union bound[1], this implies that the overall risk is at most δ_0. The proof of the consistency and of the termination assumptions are therefore immediate consequences of Hoeffding's bounds (we could use Bernstein's bounds if we believed that small standard deviations matter). A (better) variant, based on $\sum_{n \geq 1} 1/n^2 = \pi^2/6$ is

---

Function $computeBounds(s)$ (variant 2)
Static internal variable: $nbTest(s)$, initialized at 0.
Let n be the number of times s has been simulated.
Let r be the average reward over those s simulations.
$nbTest(s) = nbTest(s) + 1$
Let $lb(s) = r/n - deviation_{\text{Hoeffding}}\left(\delta_0/(\#S \times \left(\frac{\pi^2 nbTest(s)^2}{6}\right)), n\right)$.
Let $ub(s) = r/n + deviation_{\text{Hoeffding}}\left(\delta_0/(\#S \times \left(\frac{\pi^2 nbTest(s)^2}{6}\right)), n\right)$.

We show precisely the consistency of $computeBounds$ below.

Lemma 4.1 (Consistency of $computeBounds$.)
For all S finite, for all algorithms calling $computeBounds$ and simulating modifications in arbitrary order, with probability at least $1 - \delta_0$, for all s and after each simulation, $lb(s) \leq e(s) \leq ub(s)$.

Proof: We do the proof for the first variant of the algorithm; the case of the second variant is similar. This is an immediate consequence of Hoeffding's bound. The risk of a confidence interval $[lb(s), ub(s)]$ is $\delta_0/(\#S \times 2^n bTest(s))$ by Hoeffding's bound. By union bound, the risk for all confidence intervals simultaneously is therefore the sum over $\#S$ patterns of $\sum_{nbTests=1}^{\infty} \delta_0/(\#S \times 2^{nbTests})$, i.e. δ_0. □

Our algorithm, BGP (Bandit-based Genetic Programming), based on the $computeBounds$ function above, is as follows:

BGP algorithm.
$S = S_0 =$ some initial set of modifications.

[1] Using something better than the union bound is probably possible as the tests over several patterns are independent; yet, this is not straightforward as independence holds between the patterns, but not for the several tests on a same pattern.

while $S \neq \emptyset$ **do**
  Select $s \in S$ // the selection rule is not specified here
  // (the result is independent of it)
  Let n be the number of simulations of modification s.
  Simulate s n more times (*i.e.* now s has been simulated $2n$ times).
  //this ensures $nbTests(s) = O(\log(n(s)))$
  $computeBounds(s)$
  if $lb(s) > 0.501$ **then**
    Accept s; exit the program.
  else if $ub(s) < 0.504$ **then**
    $S = S \setminus \{s\}$ s is discarded.
  end if
end while

We do not specify the selection rule. The result below is independent of the particular rule.

Theorem 4.2 (Consistency of BGP)
When using variant 1 or variant 2 of $computeBounds$, or any other version ensuring consistency (Eq. 3) and termination (Eq. 4), BGP is consistent in the sense that:

1. *if at least one modification s has efficiency $> .504$, then with probability at least $1 - \delta_0$ a modification with efficiency $> .501$ will be selected (and the algorithm terminates).*
2. *if no modification has efficiency $> .504$, then with probability at least $1 - \delta_0$ the algorithm will*
   (a) *either select a modification with efficiency $> .501$ (and terminate);*
   (b) *or select no modification and terminate.*

Remark: The constants 0.501 and 0.504 are arbitrary provided that the latter is greater or equal to the former. Our results are only for consistency; we have no bounds on rates. This is the main further work.

Proof:
First, the algorithm necessarily terminates. This is proved as follows:

- Assume, in order to get a contradiction, that the algorithm does not terminate.
- Then, at least one modification s is tested infinitely often.
- By the strong law of large numbers, applied to the finitely many modifications of S and in particular to s,

$$r(s)/n(s) \to e(s) \text{ almost surely.} \tag{5}$$

- The following holds for all s which are simulated infinitely often:

$$nbTests(s) = O(\log(n(s))) \tag{6}$$

  (see the pseudocode of BGP and remarks therein for the proof of Eq. 6).

- Eq. 5 and Eq. 6 together imply that (for both variants of $computeBounds$) that $lb(s) - r(s)/n(s) \to 0$ and $ub(s) - r(s)/n(s) \to 0$. As a consequence,

$$lb(s) \to e(s) \text{ and } ub(s) \to e(s). \tag{7}$$

- Necessarily, $e(s) > 0.501$ or $e(s) < 0.504$. This and Eq. 7 imply that one of the halting condition is met or s is discarded; this is a contradiction with the fact that s is simulated infinitely often.

Second, thanks to Lemma 4.1, we can claim that with probability $1 - \delta_0$,

$$\forall s, e(s) \in [lb(s), ub(s)]. \tag{8}$$

In the rest of this proof, we consider only what happens in this case (the result is only claimed with probability $1 - \delta_0$, and therefore we do not have to consider the other case which occurs with probability $\leq \delta_0$). Thanks to Eq. 8, the proof of the remaining part (i.e. the properties 1 and 2, given that termination is established) is easy:

- By construction of the algorithm, a modification s can't be discarded if its upper bound is $> .504$; as $e(s) < ub(s)$, it can't be discarded if $e(s) > .504$.

- By construction of the algorithm, a modification can't be accepted it its lower bound is $< .501$; as $lb(s) < e(s)$, it can't be accepted if $e(s) < .501$. □

We have only considered $|S| < \infty$. The extension to $S = \{s_1, s_2, s_3, \ldots\}$ countable is straightforward with the following variant of $computeBounds$:

Function $computeBounds(s)$ (variant 3, for countable S)
Static internal variable: $nbTest(s)$, initialized at 0.
Let n be the number of times s has been simulated.
Let r be the average reward over those s simulations.
$nbTest(s) = nbTest(s) + 1$
Let i be such that $s_i = s$ and $\delta_i = 6\delta_0/(\pi^2 i^2)$.
Let $lb(s) = r/n - deviation_{\text{Hoeffding}}\left(\delta_i/(\left(\frac{\pi^2 nbTest(s)^2}{6}\right)), n\right)$.
Let $ub(s) = r/n + deviation_{\text{Hoeffding}}\left(\delta_i/(\left(\frac{\pi^2 nbTest(s)^2}{6}\right)), n\right)$.

The proof of Lemma 4.1 still holds, with this adapted lb and ub, with S countable. For Theorem 4.2, we have to ensure the termination criterion (Eq. 4). For this, we can use the following $select$ algorithm: $select = s_{i_0}$ with i_0 minimum such that $s_i \in S$. This means that we validate or invalidate patterns one at a time[2]. This solution, which ensures that all possible modifications are tested iteratively and that the performance is non-decreasing, will not be further discussed in this paper.

[2]Other solutions are possible, provided that, if the algorithm does not stop, then $\log(nbTests(s_i)i) = o(n(i))$ for all i.

5 Experiments

We will experiment our algorithm on MoGo, a program for the game of Go. We will consider the optimization of its module dedicated to bias as a function of patterns. Life is a Game of Go in which rules have been made unnecessarily complex, according to an old proverb. As a matter of fact, Go has very simple rules, is very difficult for computers, is central in education in many Asian countries (part of school activities in some countries) and has NP-completeness properties for some families of situationsCrasmaru (1999), and PSPACE-hardness for othersLichtenstein & Sipser (1980); Crasmaru & Tromp (2000), and EXPTIME-completeness for some versions Robson (1983). It has also been chosen as a testbed for artificial intelligence by many researchers. The main tools, for the game of Go, are currently MCTS/UCT (Monte-Carlo Tree Search, Upper Confidence Trees); these tools are also central in many difficult games and in high-dimensional planning. An example of nice Go game, won by MoGo as white in 2008 in the GPW Cup, is given in Fig. 1. Since

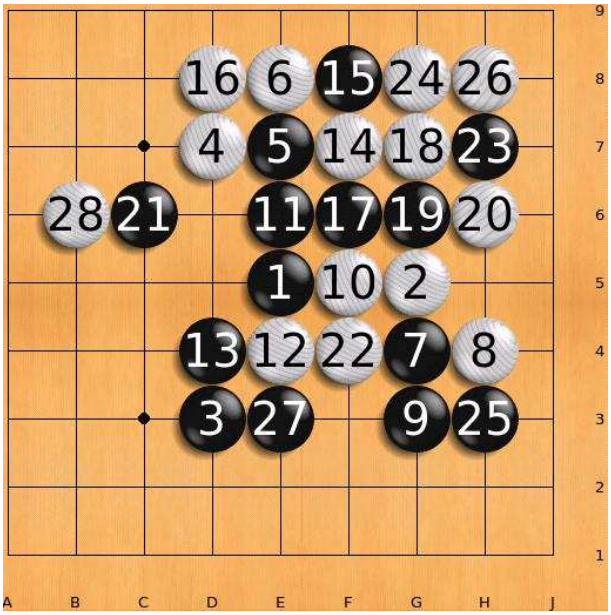

Figure 1: A decisive move (number 28) played by MoGo as white, in the GPW Cup 2008.

these approaches have been defined Chaslot *et al.* (2006); Coulom (2006); Kocsis & Szepesvari (2006), several improvements have appeared like First-Play Urgency Wang & Gelly (2007), Rave-values Bruegmann (1993); Gelly & Silver (2007) (see `ftp://ftp.cgl.ucsf.edu/pub/pett/go/ladder/mcgo.ps` for B. Bruegman's unpublished paper), patterns and progressive widening Coulom (2007); Chaslot *et al.* (2007), better than UCB-like (Upper Confidence Bounds) exploration terms Lee *et al.* (2009), large-scale parallelization Gelly *et al.* (2008); Chaslot *et al.* (2008); Cazenave &

Jouandeau (2007); Kato & Takeuchi (2008), automatic building of huge opening books Audouard *et al.* (2009). Thanks to all these improvements, our implementation MoGo already won even games against a professional player in 9x9 (Amsterdam, 2007; Paris, 2008; Taiwan 2009), and recently won with handicap 6 against a professional player (Tainan, 2009), and with handicap 7 against a top professional player, Zhou Junxun, winner of the LG-Cup 2007 (Tainan, 2009). Besides impressive results for the game of Go, MCTS/UCT have been applied to non-linear optimization Auger & Teytaud (2010), optimal sailing Kocsis & Szepesvari (2006), active learning Rolet *et al.* (2009). The formula used in the bandit is incredibly complicated, and it is now very hard to improve the current best formula Lee *et al.* (2009).

Here we will consider only mutations consisting in adding patterns in our program MoGo. Therefore, accepting a mutation is equivalent to accepting a pattern. We experiment random patterns for biasing UCT. The reader interested in the details of this is referred to Lee *et al*. (2009). Our patterns contain jokers, black stones, empty locations, white stones, locations out of the goban, and are used as masks over all the board: this means that for a given location, we consider patterns like "there is a black stone at coordinate +2,+1, a stone (of any color) at coordinate +3,0, and the location at coordinate -1,-1 is empty". This is a very particular form of genetic programming.

We consider here the automatic generation of patterns for biasing the simulations in 9x9 and 19x19 Go. Please note that:

- When we speak of good or bad shapes here, it is in the sense of "shapes that should be more simulated by a UCT-like algorithm", or "shapes that should be less simulated by a UCT-like algorithm". This is not necessarily equivalent to "good" or "bad" shapes for human players (yet, there are correlations).
- In 19x19 Go, MoGoCVS is based on tenths of thousands of patterns as in Chaslot *et al.* (2007). Therefore, we do not start from scratch. A possible goal would be to have similar results, with less patterns, so that the algorithm is faster (the big database of patterns provides good biases but it is very slow).
- In 9x9 Go, there are no big library of shapes available; yet, human expertise has been encoded in MoGo, and we are far from starting from scratch. Engineers have spent hundreds of hours manually optimizing patterns. The goals are both (i) finding shapes that should be more simulated (ii) finding shapes that should be less simulated.

Section 5.1 presents our experiments for finding good shapes in 9x9 Go. Section 5.2 presents our experiments for finding bad shapes in 9x9 Go. Section 5.3 presents our unsuccessful experiments for finding both good and bad shapes in 19x19, from MoGoCVS and its database of patterns as in Chaslot *et al.* (2007). Section 5.4 presents results on MoGoCVS with patterns removed, in order to improve the version of MoGoCVS without the big database of pattern.

5.1 Finding good shapes for simulations in 9x9 Go

Here the baseline is MoGo CVS. All programs are run on one core, with 10 000 simulations per move. All experiments are performed on Grid5000. The selection rule,

not specified in BGP, is the upper bound as in UCBLai & Robbins (1985); Auer *et al.* (2002): we simulate s such that $ub(s)$ is maximal. We here test modifications which give a positive bias to some patterns, *i.e.* we look for shapes that should be simulated more often.

For each iteration, we randomly generate some individuals, and test them with the BGP algorithm. For the three first iterations, 10 patterns were randomly generated; the two first times, one of these 10 patterns was validated; the third time, no pattern was validated. Therefore, we have three version of MoGo: MoGoCVS, MoGoCVS+P1, and MoGoCVS+P1+P2, where P1 is the pattern validated at the first iteration and P2 is the pattern validated at the second iteration. We then tested the relative efficiency of these MoGos as follows:

Tested code	Opponent	Success rate
MoGoCVS + P1	MoGoCVS	$50.78\% \pm 0.10\%$
MoGoCVS + P1 + P2	MoGoCVS +P1	$51.2\% \pm 0.20\%$
MoGoCVS + P1 + P2	MoGoCVS	$51.9\% \pm 0.16\%$

We also checked that this modification is also efficient for 100 000 simulations per move, with success rate $52.1 \pm 0.6\%$ for MoGoCVS+P1+P2 against MoGoCVS.

There was no pattern validated during the third iteration, which was quite expensive (one week on a cluster). We therefore switched to another variant; we tested the case $|S_0| = 1$, *i.e.* we test one individual at a time.We launched 153 iterations with this new version. There were therefore 153 tested patterns, and none of them was validated.

5.2 Finding bad shapes for simulations in 9x9 Go

We now switched to the research of negative shapes, *i.e.* patterns with a negative influence of the probability, for a move, to be simulated. We kept $|S_0| = 1$, i.e. only one pattern tested at each iteration. There were 173 iterations, and two patterns P3 and P4 were validated. We verified the quality of these negative patterns as follows, with mogoCVS the version obtained in the section above:

Tested code	Opponent	Success rate
MoGoCVS + P1 + P2 + P3	MoGoCVS + P1 + P2	$50.9\% \pm 0.2\%$
MoGoCVS + P1 + P2 + P3	MoGoCVS	$52.6\% \pm 0.16\%$
MoGoCVS + P1 + P2 + P3 + P4	MoGoCVS + P1 + P2 + P3	$50.6\% \pm 0.13\%$
MoGoCVS + P1 + P2 + P3 + P4	MoGoCVS	$53.5\% \pm 0.16\%$

This leads to an overall success of 53.5% against MoGoCVS, obtained by BGP.

5.3 Improving 19x19 Go with database of patterns

In 19x19 Go, all tests are performed with 3500 simulations per move. Here also, we tested the case $|S_0| = 1$, *i.e.* we test one individual at a time. We tested only positive biases. The algorithm was launched for 62 iterations. Unfortunately, none of these 62 iterations was accepted. Therefore, we concluded that improving these highly optimized version was too difficult. We switched to another goal: having the same efficiency with

faster simulations and less memory (the big database of patterns strongly slowers the simulations and takes a lot of simulations), as discussed below.

5.4 Improving 19x19 Go without database of patterns

We therefore removed all the database of patterns; the simulations of MoGo are much faster in this case, but the resulting program is nonetheless weaker because simulations are far less efficient (see *e.g.* Lee *et al.* (2009)). Fig. 2 presents a known (from Senseis `http://senseis.xmp.net/?GoodEmptyTriangle#toc1`) difficult case for patterns: move 2 is a good move in spite of the fact that locally (move 2 and locations at the east, north, and north east) form a known very bad pattern (termed empty triangle), termed empty triangle, and is nonetheless a good move due to the surroundings.

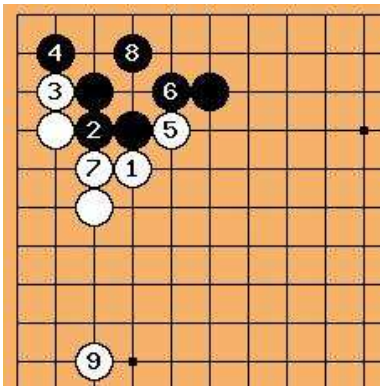

Figure 2: An example from Senseis of good large pattern in spite of a very bad small pattern. The move 2 is a good move.

We keep $|S_0| = 1$, and we have 443 iterations. There were ten patterns validated, validated at iterations 16, 22, 31, 57, 100, 127, 136, 260, 285 and 331. We could validate these patterns Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9 and Q10 as follows. MoGoCVS+AE means MoGoCVS equipped with the big database of patterns extracted from games between humans.

Tested code	Opponent	Success rate
MoGoCVS + Q1	MoGoCVS	$50.9\% \pm 0.13\%$
MoGoCVS + Q1 + Q2	MoGoCVS + Q1	$51.2\% \pm 0.28\%$
MoGoCVS + Q1 + Q2 + Q3	MoGoCVS + Q1 + Q2	$56.7\% \pm 1.50\%$
MoGoCVS + Q1 + ... + Q4	MoGoCVS + Q1 + Q2 + Q3	$52.1\% \pm 0.39\%$
MoGoCVS + Q1 + ... + Q5	MoGoCVS + Q1 + ... + Q4	$51.1\% \pm 0.20\%$
MoGoCVS + Q1 + ... + Q6	MoGoCVS + Q1 + ... + Q5	$54.1\% \pm 0.78\%$
MoGoCVS + Q1 + ... + Q7	MoGoCVS + Q1 + ... + Q6	$50.9\% \pm 0.20\%$
MoGoCVS + Q1 + ... + Q8	MoGoCVS + Q1 + ... + Q7	$51.2\% \pm 0.28\%$
MoGoCVS + Q1 + ... + Q9	MoGoCVS + Q1 + ... + Q8	$50.4\% \pm 0.10\%$
MoGoCVS + Q1 + ... + Q10	MoGoCVS + Q1 + ... + Q9	$52.3\% \pm 0.55\%$
MoGoCVS + Q1 + Q2	MoGoCVS	$53.4\% \pm 0.50\%$
MoGoCVS + Q1 + Q2 + Q3	MoGoCVS	$57.3\% \pm 0.49\%$
MoGoCVS + Q1 + ... + Q4	MoGoCVS	$59.4\% \pm 0.49\%$
MoGoCVS + Q1 + ... + Q5	MoGoCVS	$58.6\% \pm 0.49\%$
MoGoCVS + Q1 + ... + Q6	MoGoCVS	$61.7\% \pm 0.49\%$
MoGoCVS + Q1 + ... + Q7	MoGoCVS	$61.3\% \pm 0.49\%$
MoGoCVS + Q1 + ... + Q8	MoGoCVS	$63.1\% \pm 0.48\%$
MoGoCVS + Q1 + ... + Q9	MoGoCVS	$62.3\% \pm 0.48\%$
MoGoCVS + Q1 + ... + Q10	MoGoCVS	$63.0\% \pm 0.48\%$
MoGoCVS	MoGoCVS + AE	$26.6\% \pm 0.20\%$
MoGoCVS + Q1	MoGoCVS + AE	$27.5\% \pm 0.49\%$
MoGoCVS + Q1 + Q2	MoGoCVS + AE	$28.0\% \pm 0.51\%$
MoGoCVS + Q1 + Q2 + Q3	MoGoCVS + AE	$30.9\% \pm 0.46\%$
MoGoCVS + Q1 + ... + Q4	MoGoCVS + AE	$32.1\% \pm 0.43\%$
MoGoCVS + Q1 + ... + Q5	MoGoCVS + AE	$30.9\% \pm 0.46\%$
MoGoCVS + Q1 + ... + Q6	MoGoCVS + AE	$32.8\% \pm 0.47\%$
MoGoCVS + Q1 + ... + Q7	MoGoCVS + AE	$31.9\% \pm 0.47\%$
MoGoCVS + Q1 + ... + Q8	MoGoCVS + AE	$32.2\% \pm 0.47\%$
MoGoCVS + Q1 + ... + Q9	MoGoCVS + AE	$32.6\% \pm 0.47\%$
MoGoCVS + Q1 + ... + Q10	MoGoCVS + AE	$34.5\% \pm 0.48\%$

An important property of BGP is that all validated patterns are confirmed by these independent experiments. We see however that in 19x19, we could reach roughly 30% of success rate against the big database built on human games (therefore our BGP version uses far less memory than the other version); we will keep this experiment running, so that maybe we can go beyond 50 %. Nonetheless, we point out that we already have 60 % against the version without the database, and the performance is still increasing (improvements were found at iterations 16,22,57,100,122,127, with regular improvements - we have no plateau yet) - therefore we successfully improved the version without patterns, which is lighter (90% of the size of MoGoCVS is in the database).

6 Conclusions

We proposed an original tool for genetic programming and applied it in a reinforcement learning problem. This tool is quite conservative: the learning is based on a set of admissible modifications, and has strong theoretical guarantees. Interestingly, the application of this theory to GP was successful in experiments, with in particular the nice property that all patterns selected during the GP run could be validated in independent experiments. We point out that when humans test modifications of MoGo, they usually test their algorithms based on simple confidence intervals, without taking into account the fact that, as they test multiple variants, one of these variants might succeed just by chance - it happened quite often that modifications accepted in the CVS were later removed, causing big delays and many non-regression tests. This is in particular true for this kind of applications, because the big noise in the results, the big computational costs of the experiments, imply that people can't use p-values like 10^{-10} - with BGP, the confidence intervals can be computed at a reasonnable confidence level, and the algorithm takes care by itself of the risk due to the multiple simultaneous hypothesis testing.

In 9x9 Go, BGP outperformed human development, and the current CVS of MoGo is the version developped by BGP. In 19x19 Go, we have an improvement over the default version of MoGo, using a big database learnt offline in supervised learning, but not against the version enabling the use of big databases - we nonetheless keep running the experiments as the success rate is still increasing and we had a big improvement for light versions.

Further work: The main further work is the analysis of the number of iterations before finding a good modification when such a good modification exists, depending on the number of patterns tested. This should in particular clarify the differences between the different versions of "computeBounds". We are very grateful to reviewer # 2 for pointing out interesting remarks around that.

Acknowledgements

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

AUDIBERT J.-Y., MUNOS R. & SZEPESVARI C. (2006). Use of variance estimation in the multi-armed bandit problem. In *NIPS 2006 Workshop on On-line Trading of Exploration and Exploitation*.

AUDOUARD P., CHASLOT G., HOOCK J.-B., PEREZ J., RIMMEL A. & TEYTAUD O. (2009). Grid coevolution for adaptive simulations; application to the building of opening books in the game of go. In *Proceedings of EvoGames*.

AUER P., CESA-BIANCHI N. & FISCHER P. (2002). Finite time analysis of the multi-armed bandit problem. *Machine Learning*, **47**(2/3), 235–256.

AUGER A. & TEYTAUD O. (2010). Continuous lunches are free plus the design of optimal optimization algorithms. *Algorithmica*, p. 2009.

BRUEGMANN B. (1993). Monte carlo go.

CAZENAVE T. & JOUANDEAU N. (2007). On the parallelization of UCT. In *Proceedings of CGW07*, p. 93–101.

CHASLOT G., SAITO J.-T., BOUZY B., UITERWIJK J. W. H. M. & VAN DEN HERIK H. J. (2006). Monte-Carlo Strategies for Computer Go. In P.-Y. SCHOBBENS, W. VANHOOF & G. SCHWANEN, Eds., *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, p. 83–91.

CHASLOT G., WINANDS M., UITERWIJK J., VAN DEN HERIK H. & BOUZY B. (2007). Progressive strategies for monte-carlo tree search. In P. WANG & OTHERS, Eds., *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, p. 655–661: World Scientific Publishing Co. Pte. Ltd.

CHASLOT G., WINANDS M. & VAN DEN HERIK H. (2008). Parallel Monte-Carlo Tree Search. In *Proceedings of the Conference on Computers and Games 2008 (CG 2008)*.

COULOM R. (2006). 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*.

COULOM R. (2007). Computing elo ratings of move patterns in the game of go. In *Computer Games Workshop, Amsterdam, The Netherlands*.

CRASMARU M. (1999). On the complexity of Tsume-Go. **1558**, 222–231.

CRASMARU M. & TROMP J. (2000). Ladders are PSPACE-complete. In *Computers and Games*, p. 241–249.

DESROSIÈRES A. (2000). *La politique des grands nombres : histoire de la raison statistique*. La Dcouverte.

EVEN-DAR E., MANNOR S. & MANSOUR Y. (2006). Action elimination and stopping conditions for the multi-armed bandit and reinforcement learning problems. *Journal of Machine Learning Research*, **7**, 1079–1105.

GELLY S., HOOCK J. B., RIMMEL A., TEYTAUD O. & KALEMKARIAN Y. (2008). The parallelization of monte-carlo planning. In *Proceedings of the International Conference on Informatics in Control, Automation and Robotics (ICINCO 2008)*, p. 198–203. To appear.

GELLY S. & SILVER D. (2007). Combining online and offline knowledge in UCT. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, p. 273–280, New York, NY, USA: ACM Press.

HEIDRICH-MEISNER V. & IGEL C. (2009). Hoeffding and bernstein races for selecting policies in evolutionary direct policy search. In *ICML '09: Proceedings of the 26th Annual International Conference on Machine Learning*, p. 401–408, New York, NY, USA: ACM.

HOLLAND J. H. (1973). Genetic algorithms and the optimal allocation of trials. *SIAM J. Comput.*, **2**(2), 88–105.

KATO H. & TAKEUCHI I. (2008). Parallel monte-carlo tree search with simulation servers. In *13th Game Programming Workshop (GPW-08)*.

KOCSIS L. & SZEPESVARI C. (2006). Bandit-based monte-carlo planning. In *ECML'06*, p. 282–293.

KOZA J. R. (1992). *Genetic Programming: On the Programming of Computers by means of Natural Evolution*. Massachusetts: MIT Press.

LAI T. & ROBBINS H. (1985). Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, **6**, 4–22.

LEE C.-S., WANG M.-H., CHASLOT G., HOOCK J.-B., RIMMEL A., TEYTAUD O., TSAI S.-R., HSU S.-C. & HONG T.-P. (2009). The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transactions on Computational Intelligence and AI in games*, p. 73–89.

LICHTENSTEIN D. & SIPSER M. (1980). Go is polynomial-space hard. *J. ACM*, **27**(2), 393–401.

MNIH V., SZEPESVÁRI C. & AUDIBERT J.-Y. (2008). Empirical Bernstein stopping. In *ICML '08: Proceedings of the 25th international conference on Machine learning*, p. 672–679, New York, NY, USA: ACM.

ROBSON J. M. (1983). The complexity of go. In *IFIP Congress*, p. 413–417.

ROLET P., SEBAG M. & TEYTAUD O. (2009). Optimal active learning through billiards and upper confidence trees in continous domains. In *Proceedings of the ECML conference*.

WANG Y. & GELLY S. (2007). Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii*, p. 175–182.