Combining expert, offline, transient and online knowledge in Monte-Carlo exploration

Guillaume Chaslot, Louis Chatriot, C. Fiter, Sylvain Gelly, Jean-Baptiste Hoock, Julien Perez, Arpad Rimmel, Olivier Teytaud

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

Abstract. We combine for Monte-Carlo exploration machine learning at four different time scales:

- online regret, through the use of bandit algorithms and Monte-Carlo estimates;
- transient learning, through the use of rapid action value estimates (RAVE) which are learnt online and used for accelerating the exploration and are thereafter neglected;
- offline learning, by data mining of datasets of games;
- use of expert knowledge coming from the old ages as prior information.

The result algorithm is stronger than each element separately. We finally emphasize the exploration-exploitation dilemna in the Monte-Carlo simulations and show great improvements that can be reached with a fine tuning of related constants.

1 Introduction

Definitions of the different Go terms used in this article can be found on the web site `http://senseis.xmp.net/`.

Monte-Carlo Tree Search (MCTS [6, 8, 12]) is a recent tool for difficult planning tasks. Impressive results have already been produced in the case of the game of Go [8, 11]. In this paper, we present the inclusion of expert knowledge in the specific case of the game of Go, a zero-sum sequential game.

MCTS consists in building a tree, in which nodes are situations of the considered environnement and branches are the actions that can be taken by the agent. The main point in MCTS is that the tree is highly unbalanced, with a strong bias in favor of important parts of the tree. The focus is on the parts of the tree in which the expected gain is the highest. For solving such issues, several algorithms have been proposed: UCT[12] (Upper Confidence Trees), focuses on the number of winning simulation plus an uncertainty measure; AMAF [11] (All Moves As First, also termed RAVE for Rapid Action-Value Estimates), focuses on a compromise between UCT and heuristic information extracted from the simulations and BAST[7] (Bandit Algorithm for Search in Tree), uses UCB-like bounds modified through the overall number of nodes in the tree. Other related

algorithms have been proposed as in [6], essentially using a decreasing impact of a priori bias as the number of simulations increases.

In the context of the game of Go, the nodes are equipped with one Go board configuration, also called a coloration of the board, and with statistics, typically the number of won and lost games in the simulations started from this node. As explained in Algorithm 1, MCTS uses these statistics in order to iteratively expand the tree in the regions where the expect reward is maximum. After each simulation from the current position (the root) until the end of the game, the win and loss statistics are updated in every node concerned by the simulation, and a new node corresponding to the first new situation of the simulation is created.

As a consequence, in order to design simulations from the root of the tree (the current real situation) to a final position (win or loss), we have to take decisions (i) in the tree, until we reach a leaf, and (ii) out of the tree, until the game is over. The aim of these two kinds of choices are different. In the tree, the transition must ensure a trade off between exploration and exploitation. Out of the tree, the simulations, usually termed Monte-Carlo simulations, have to give a good estimate of the probability of winning.

Algorithm 1 summarizes Monte-Carlo planning.

Algorithm 1 Pseudo-code of a MCTS algorithm applied to a zero-sum game (typically Go or chess). T is a tree of positions, with each node equipped with statistics (number of wins and losses in simulations starting at this node). Concerning the decision line at the very end of the pseudo-code, the most simulated decision is known as the most reliable criterion; other solutions such as taking the decision with the highest ratio "number of wins divided by the number of simulations" are unsufficiently robust, due to the possible small number of simulations. Here rewards at the end of each game simulation are binary (win or loss), but arbitrary distributions of rewards can be used.

Initialize T to a single node, representing the current state.
while Time left > 0 **do**
  Simulate one game until a leaf (a position) L of T (thanks to bandit algorithms, see Algorithm 3).
  Choose one son (a successor) L' of L.
  Simulate one game from position L' until the game is over (thanks to Monte-Carlo, see Algorithm 2).
  Grow of the tree: add L' as a son of L in T.
  Update statistics in all the tree. In UCT, each node knows how many winning simulations (from this node) have been performed and how many simulations (from this node) have been performed. In other forms of tree-search, more information is necessary (heuristic information based on permutations of simulations in AMAF [11], total number of nodes in the tree in BAST[7]).
end while
Return the move which has been simulated most often from the root.

The function used for taking decisions out of the tree (i.e. the so-called Monte-Carlo part, MC) is defined in Algorithm 2. An atari occurs when a string (a group of stones) can be captured in one move. Some Go knowledge has been added in this part in the form of 3*3 expert designed patterns in order to play more meaningfull games.

Algorithm 2 Algorithm for choosing a move in MC simulations, for the game of Go.

```
if the last move is an atari, then
  Save the stones which are in atari.
else
  if there is an empty location among the 8 locations around the last move which
  matches a pattern then
    Play randomly uniformly in one of these locations.
  else
    if there is a move which captures stones then
      Capture stones.
    else
      if there is a legal move then
        Play randomly a legal move
      else
        Return pass.
      end if
    end if
  end if
end if
```

The function used for simulating in the tree is presented in Algorithm 3. This function is a main part of the code: it decides in which direction the tree should be expanded. There are various formulas, all of them being based on the idea of a tradeoff between exploitation (further simulating the seemingly good moves) and exploration (further simulating the moves which have not been explored a lot). The bandit algorithm for AMAF has been derived in [11]; roughly, the AMAF simulations are created by permutations of moves in "real" simulations. An important improvement (termed progressive widening[9]) of Algorithm 3 consists in considering only the $K(n)$ "best" moves (the best according to some heuristic), at the n^{th} simulation in a given node, with $K(n)$ a non-decreasing mapping from $\mathbb{N}$ to $\mathbb{N}$.

Unfortunately, some problems appear.

First, the Monte-Carlo part is not yet completly understood. Indeed, many discussions around this subject have been published in the computer-go mailing list, but nobody could find criteria for determining, without extensive experiments of the whole MCTS algorithm, what is a good Monte-Carlo simulator. For example, using a good player, e.g. another Go program, for this purpose, can strongly reduce the efficiency of the overall program. This holds whenever

Algorithm 3 Algorithm for choosing a move in the tree, for a zero-sum game with binary reward (extensions to arbitrary distributions are straightforward). $Sims(s,d)$ is the number of simulations starting at s with first move d. The total number of simulations at situation s is $Sims(s) = \sum_d Sims(s,d)$. We present here various formulas for computing the score (see [13, 2, 11] for UCB1, UCB-Tuned and AMAF respectively); other very important variants (for infinite domains, larger number of arms, of specific assumptions) can be found in [3, 1, 10, 7, 6].

Function $decision = Bandit(situation\ s\ in\ the\ tree)$.
for d in the set of possible decisions **do**
Let $\hat{p}(d) = Wins(s,d)/Sims(s,d)$.
switch (bandit formula):

- **UCB1**: compute $score(d) = \hat{p}(d) + \sqrt{2\log(Sims(s))/Sims(s,d)}$.
- **UCB-Tuned.1**: compute $score(d) = \hat{p}(d) + \sqrt{\hat{V}\log(Sims(s))/Sims(s,d)}$ with $\hat{V} = \max(0.001, \hat{p}(d)(1-\hat{p}(d)))$.
- **UCB-Tuned.2**: compute $score(d) = \hat{p}(d) + \sqrt{\hat{V}\log(Sims(s))/Sims(s,d)} + \log(Sims(s))/Sims(s,d)$ with $\hat{V} = \max(0.001, \hat{p}(d)(1-\hat{p}(d)))$.
- **AMAF-guided exploration**: Compute

$$score(d) = \alpha(d)\hat{p}(d) + (1-\alpha(d))\hat{\hat{p}}(d) \quad (1)$$

with:

- $\hat{\hat{p}}(d)$ the ratio of won simulations out of AMAF-simulations using decision d in situation s (see section 3);
- $\alpha(d)$ a coefficient depending on $Sims(s,d)$ (see [11]).

end switch
end for
Return $\arg\max_d\ score(d)$.

we just test it with a constant fixed number of simulations, independently of the computational cost. An intuitive explanation could be that a too biased Monte-Carlo simulator (bias towards prefering some families of situations) in the MCTS algorithm leads to false estimates of the winning rates.

Second, the bandit is also quite unclear. Some implementations still use the UCB formula [2], but with very small constants for the exploration part — mainly, the move with best empirical results is used, independently of its number of simulations, until its estimated probability of winning becomes smaller than the estimated probability of winning for unseen nodes. We here present (i) improvements for the tree part (function Bandit, see Algorithm 3) and (ii) improvements for the Monte-Carlo part (Algorithm 2).

[8] and [6] have already combined offline learning (statistics from professional games) and online learning (bandit choice of moves). [11] combines online learning (bandit choice) and transient learning (AMAF values) and experimented the use of offline learning, but the offline learning part (using RLGO) was later removed from MoGo as the improvement was minor and even negative after tuning. We here present our algorithm combining:

- Online learning (bandit), in section 2;
- Transient learning (AMAF-values as in [11]), which is somewhat similar to agregate learning in the sense that a simulation on one state influences several states in a transient manner; this part is presented in section 3;
- Expert rules, as explained in section 4;
- Offline learning, thanks to patterns as in [6], as explained in section 5;
- Diversity preservation in the Monte-Carlo simulation as explained in section 6.1;
- Counter-examples around the Nakade as explained in section 6.2.

The combination of these various learning is presented in section 7.

2 Online learning: Monte-Carlo Tree Search

The principle of MCTS consists in building, in an incremental manner, a tree of possible situations; the root is the current situation, an edge is a legal move, and an edge corresponding to a move c links a position to the next position when move c is played. A huge number of simulated games is performed; each simulation updates statistics in the tree, increases the tree by adding some nodes, and is itself influenced by statistics. The way statistics influence simulations is the so-called "bandit" module.

The bandit algorithm is aimed at online learning values of moves; it dynamically chooses moves to be explored. In many programs, this is done by UCT-like algorithms, i.e. choosing the move such that the following quantity is maximal:

$$score(move) = \underbrace{w(move)/n(move)}_{Exploitation} + C \underbrace{\sqrt{\log(\sum_{m \in M} n(m))/n(move)}}_{Exploration} \quad (2)$$

where:

- $n(move)$ is the number of times this move has been explored in this situation;
- $w(move)$ is the number of times this move has been tried with a won game as a consequence;
- M is the set of possible moves at this situation.

A classical improvement consists in replacing Eq. 2 by some improved versions (see Algorithm 3). Typically UCB-Tuned [2] is a general solution, which is not specific to games and can be applied to planning. In the case of games, formula like the AMAF values in Algorithm 3 can be used in order to bias the exploration terms in the direction of moves which are suitable according to "AMAF" statistics [11]. We will introduce (in section 4 and 5) bias on top of this bias by introducing "virtual" AMAF simulations: the number of AMAF simulations for a given node (i.e. a given situation) and a given move is not initialized to 0, but to a number depending on patterns and rules.

3 Transient learning: Rapid Action Value Estimates thanks to All Moves As First heuristic

Since [11], MoGo uses the AMAF-guided exploration; starting from zero knowledge, AMAF statistics are collected during simulations, as well as standard statistics. The principle is as follows:

- For each simulation s with reward $r \in \{0, 1\}$, collect the sequence of moves $m_1, \ldots, m_n$ from the root of the tree, until the simulation is over;
- For each node N of the simulation s with reward r, consider the moves $m_k, m_{k+2}, m_{k+4}, \ldots, m_{k+2g}$ of same color as N (moves at which the same player is going to play), until the simulation is over;
- In node N, for each i in $\{1, 2, \ldots, g\}$, consider the so-called AMAF-simulation $m_i, m_{k+2}, m_{k+4}, \ldots, m_{k+2i-2}, m_k, m_{k+2i+2}, m_{k+2i+4}, \ldots, m_{k+2g}$ with reward r also.

Of course, these simulations are 'false' simulations; we do not know if the simulation is consistent (rules are not necessarily respected), and the reward is uncertain. However, the strength of this heuristic is that it is much faster than the online bandit algorithm: for each simulation, we get several AMAF-simulations; in the case of Go, nearly half of the number of empty locations. Therefore, these AMAF values are biased, but they are averaged on bigger samples; as a consequence, they are intrinsically transient:

- at the beginning of the exploration of a node, there's no AMAF value;
- later, the standard statistics are very weak as only a few simulations are performed, but AMAF values are available; then, the weight of AMAF values is close to 1;
- eventually, the AMAF values become obsolete as standard statistics, more reliable, are collected; the weight of AMAF values goes to 0.

A weakness remains, due to the initial time: at the very beginning, neither AMAF values nor standard statistics are available; we need some prior knowledge. Such prior knowledge is of high importance in e.g. CrazyStone or Mango [8, 6], but in these source codes there's no RAVE values providing transient information. In this work we successfully plug (i) expert knowledge (section 4) (ii) patterns extracted from datasets as in Mango (section 5) into MoGo.

4 Offline learning: adding expert knowledge

The pruning of probably bad moves by expert-rules has been experimented as a more parsimonious exploration of the tree; however, such removals are dangerous as in Go, as exceptions to pruning rules are frequent. All strong programs use bias in the tree and no or almost no hard pruning.

Some published solutions for including offline knowledge in the tree are as follows:

- Progressive widening [9] or progressive unpruning [6]: the move which maximizes the score in Eq. 2 *among the n moves preferred by the heuristic* is chosen; n depends on
- Progressive bias [6]: formula 2 is modified by a term $+heuristic\ value(move)/n(move)$;
- First play urgency [15]: the score depicted in Formula 2 is used for moves m such that $n(m) > 0$, and an a priori value, depending on expert knowledge, is used for other moves;
- MoGo incorporates a blend of progressive bias and progressive widening by adding a bonus, decreasing with the number of simulations, for moves located close to the last move. The distance is designed through Go-expertise: it is a topological distance which assumes distance 1 between all stones in one string, whatever may be the size of this string.

In all cases, the influence of expert knowledge decreases in the exploration step: the online values, coming from the bandit exploration, replace the expert knowledge. This is therefore by nature different from the transient learning presented in section 3, which is initially of null influence, increases, and then decays for moves which are sufficiently strongly explored.

Various elements for specifying bias values through expert knowledge already published (e.g. [9, 5]) include:

- shapes: in [9, 5], shapes are automatically generated from datasets. We use this approach in section 5;
- capture moves (in particular, string contiguous to a new string in atari), extension (in particular out of a ladder), avoid self-atari, atari (in particular when there is a ko), distance to border (optimum distance = 3 in 19x19 Go), short distance to previous moves, short distance to the move before the previous move, and also the feature reflecting the fact that the probability (according to Monte-Carlo simulations) of a location to be of one's own color at the end of the game is $\simeq 1/3$.

The following tools are used in our implementations in 19x19, and improve the results from the AMAF heuristic:

- Territory line (i.e. line number 3): 6.67 won AMAF-simulations are added;
- Line of death (i.e. first line): 6.67 lost AMAF-simulations are added;
- Peep-connect (ie. connect two strings when the opponent threatens to cut): 5 won AMAF-simulation is added;
- Hane (a move which "reaches around" one or more of the opponent's stones): 5 won AMAF-simulation is added;
- Threat: 5 won AMAF-simulation are added;
- Connect: 5 won AMAF-simulation are added;
- Wall: 3.33 won AMAF-simulation are added;
- Bad Kogeima(same pattern as a knight's move in chess): 2.5 lost AMAF-simulation are added;
- Empty triangle (three stones making a triangle without any surrounding opponent's stone): 5 lost AMAF-simulations are added.

These shapes are illustrated on Figure 1. With a naive hand tuning of parameters, they provide 63.9±0.5 % of winning rate against the version without these improvements. We are optimistic on the fact that tuning the parameters will strongly improve the results. Moreover, since the early developments of MoGo, some "cut" bonuses are included (i.e., advantages for playing at locations which match "cut" patterns, i.e. patterns for which a location prevents the opponent from connecting two groups).

We have experimented the following features without success: line of influence (i.e. line number four); line of defeat (i.e. line number two); Kogeima; Kosumi (a diagonal move); Kata(a diagonal move to an opponent's stone); Bad tobi; Reduced liberties (i.e. if an enemy string has 3 liberties, then a bonus is applied for simulating one of these liberties).

5 Offline learning: data mining of patterns

Following [5], we built a model for evaluating the probability that a move is played, conditionally to the fact that it matches some pattern. When a node is created, the pattern matching is called, and the probability it proposes is used as explained later (Eq. 3). The pattern matching is computationnally expensive and the first experiments were negative; however, after tuning of the parameters, we could have positive results.

The following parameters had to be modified:

- time scales for the convergence of the weight of online statistics to 1 (see Eq. 3) are increased;
- the number of simulations of a move at a given node before the subsequent nodes is created is increased (because the computational cost of a creation is higher).
- the optimal coefficients of section 4 (expert rules) are modified.

Results are presented in figure 2.

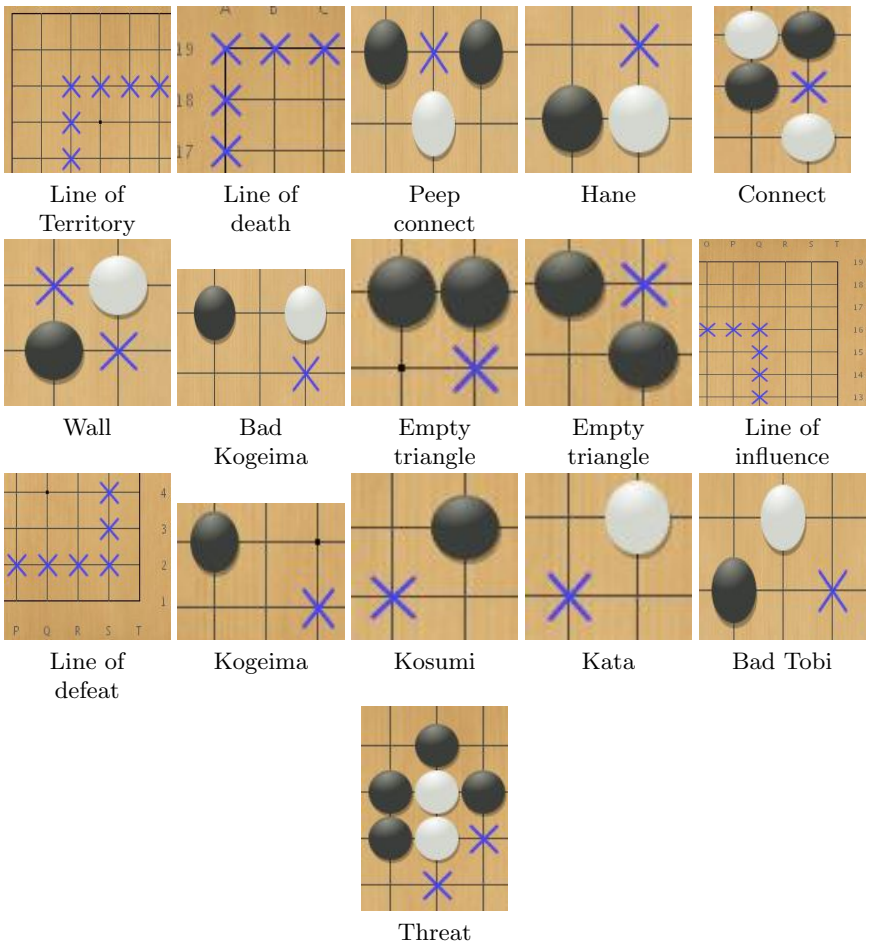


Fig. 1. We here present shapes for which exact matches are required for applying the bonus/malus. In all cases, the shapes are presented for the black player: the feature applies for a black move at one of the crosses. The reverse pattern of course applies for white. In the case of bad Kogeima, there must be no black stone around the cross for the bonus to apply. In the case of Kosumi and Kogeima, there must not be white stones between the black stone and the cross. The "alone on the first line" pattern only applies if all 5 locations around the putative move are empty. In the case of the bad tobi, there must be no black stone among the 8 locations around the cross. In the case of the Hane, the white stone must not have any white stone in its 4 neighbours. Threat is not an exact shape to be matched but just an example: in general, black has a bonus for simulating one of the liberties of an enemy string with exactly two liberties, i.e. to generate an atari.

Tested version	Against	Conditions of games	Success rate
MoGo + patterns	MoGo without patterns	Constant number of simulations = 3000/move	56 % ± 1%
MoGo + patterns	MoGo without patterns	Constant time per move 2s	50.9 % ± 1.5 %
MoGo + patterns + tuning of coefficients	MoGo + pattenrs	Constant time per move 1s	55.2 % ± 0.8 %

Fig. 2. Ajout de patterns extraites de parties professionnelles dans MoGo. Le premier recalage des paramtres est le recalage (i) des coefficients de compromis entre les diffrents apprentissages (ii) des paramtres de l'expertise Go (prsente en section 4; ces coefficients interfrent clairement avec les motifs et sont donc naturellement rquilibrer) (iii) du nombre de simulations dans un coup avant de crer le noeud subsquent. We consider the tuning of coefficients as preliminary and expect strong improvements on this.

6 Improving Monte-Carlo (MC) simulations

It is not known how to design a good Monte-Carlo simulator for MCTS in the case of Go or more generally two players games. Many people have tried to improve the MC engine by increasing its level (the strength of the Monte-Carlo simulator as a stand-alone player), but it is shown clearly in [11] that this is not the good criterion: a MC engine MC_1 which plays incredibly better than another MC_2 can lead to very poor results as a module in MCTS, whenever the computational cost is the same. Some MC engines have been learnt on datasets [9], but the results are strongly improved by changing the constants manually. In that sense, designing and calibrating a MC engine remains a dark art: one has to intensively experiment a modification in order to validate it.

Various shapes are defined in [4, 15, 14]. [15] uses patterns and expertise as explained in Algorithm 2. We present below two new improvements, both of them centered on an increased diversity when the computational power increases; in both cases, the improvement is negative or negligible for small computational power and becomes highly significant when the computational power increases.

6.1 Fill the board: random perturbations of the Monte-Carlo simulations

The principle of this modification is to play first on locations of the board where there is no stone around yet. The idea is increase the number of locations at which Monte-Carlo simulations can find pattern-matching in order to diversify the Monte-Carlo simulations.

Trying every position on the board would take too much time, so the following procedure is used instead. A location on the board is chosen randomly; if the 8 surrounding positions are empty, the move is played, else the following $N - 1$ positions on the board are tested; N is a parameter of the algorithm. This modification introduces more diversity in the simulations: this is due to the fact that the Monte-Carlo player uses a lot of patterns. When patterns match, one of them is played. So the simulations have only a few ways of beginning when a small number of patterns match around the last move, in particular at the beginning

of the game, when there are only a few stones on the goban. As this modification is played before the patterns, it leads to more diversified simulations (see Figure 3 (left)). The detailed algorithm is presented in Algorithm 4. Experiments are

Algorithm 4 Algorithm for choosing a move in MC simulations, including the "fill board" improvement. We experimented also a constraint of four empty locations instead of 8, but results were disappointing.

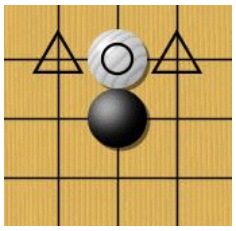
if the last move is an atari, **then**
  Save the stones which are in atari.
else
  "Fill board" part.
  for $i \in \{1, 2, 3, 4, \ldots, N\}$ **do**
    Randomly draw a location x on the goban.
    IF x is an empty location and the eight locations around x are empty, play x (exit).
  end for
  End of "fill board" part.
  if there is an empty location among the 8 locations around the last move which matches a pattern **then**
    Play randomly uniformly in one of these locations (exit).
  else
    if there is a move which captures stones **then**
      Capture stones (exit).
    else
      if there is a legal move **then**
        Play randomly a legal move (exit)
      else
        Return pass.
      end if
    end if
  end if
end if

presented in Figure 3 (right).

6.2 The "Nakade" problem

. A known weakness of MoGo, as well as many MCTS programs, is that *nakade* is not correctly handled. We will use the term *nakade* to design a situation in which a surrounded group has a single large internal, enclosed space in which the player won't be able to establish two eyes if the opponent plays correctly.

The group is therefore dead, but the baseline Monte-Carlo simulator (Algorithm 2) sometimes estimates that it lives with a high probability, i.e. the MC simulation does not necessarily lead to the death of this group. Therefore, the tree will not grow in the direction of moves preventing difficult situations with *nakade* — MoGo just considers that this is not a dangerous situation.



9x9 board		19x19 board	
Nb of playouts per move or time/move	Success rate	Nb of playouts per move or time /move	Success rate
10 000	52.9 % ± 0.5%	10000	49.3 ± 1.2 %
5s/move, 8 cores	54.3 % ± 1.2 %	5s/move, 8 cores	77.0 % ± 3.3 %
100 000	55.2 % ± 1.4 %	100 000	73.7 % ± 2.9%
200 000	55.0 % ± 1.1 %	200 000	78.4 % ± 2.9 %

Fig. 3. Left: diversity loss when the "fillboard" option was not applied: the white stone is the last move, and the black player, starting a Monte-Carlo simulation, can only play at one of the locations marked by triangles. Right: results associated to the "fillboard" modification. As the modification leads to a computational overhead, results are better for a fixed number of simulations per move; however, the improvement is clearly significant. The computational overhead is reduced when a multi-core machine is used: the concurrency for memory access is reduced when more expensive simulations are used, and therefore the difference between expensive and cheap simulations decays as the number of cores increases. By the way, this element shows the easier parallelization of heavier playouts.

This will lead to a false estimate of the probability of winning. As a consequence, the MC part (i.e. the module choosing moves for situations which are not in the tree) must be modified so that the winning probability reflects the effect of a *nakade* .

Interestingly, as most MC tools have the same weakness, and also as MoGo is mainly developed by self-play, the weakness w.r.t *nakade* almost never appeared before humans found the weakness (see post from D. Fotland called "UCT and solving life and death" on the computer-Go mailing list). It would be theoretically possible to encode in MC simulations a large set of known *nakade* behaviors, but this approach has two weaknesses: (i) it is expensive and MC simulations must be very fast (ii) abruptly changing the MC engine very often leads to unexpected disappointing effects. Therefore we designed the following modification: if a contiguous set of exactly 3 free locations is surrounded by stones from the opponent, then we play at the center (the vital point) of this "hole". The new algorithm is presented in Algorithm 5.

We validate the approach with two different experiments: (i) known positions in which old MoGo does not choose the right move (Figure 4) (ii) games confronting the new MoGo vs the old MoGo (Table 1).

7 Combining offline, transient and online learning

In this section we present how we combine online learning (bandit module, section 2), transient learning (section 3), expert knowledge (section 4) and offline pattern-information (section 5). We point out that this combination is far from

Algorithm 5 New MC simulator, avoiding the *nakade* problem.

if the last move is an atari, **then**
  Save the stones which are in atari.
else
  Beginning of the *nakade* modification
  for x in one of the 4 empty locations around the last move **do**
    if x is in a hole of 3 contiguous locations surrounded by enemy stones or the sides of the goban **then**
      Play the center of this hole (exit).
    end if
  end for
  End of the *nakade* modification
  "Fill board" part.
  for $i \in \{1, 2, 3, 4, \ldots, N\}$ **do**
    Randomly draw a location x on the goban.
    IF x is an empty location and the eight locations around x are empty, play x (exit).
  end for
  End of "fill board" part.
  if there is an empty location among the 8 locations around the last move which matches a pattern **then**
    Play randomly uniformly in one of these locations (exit).
  else
    if there is a move which captures stones **then**
      Capture stones (exit).
    else
      if there is a legal move **then**
        Play randomly a legal move (exit).
      else
        Return pass.
      end if
    end if
  end if
end if

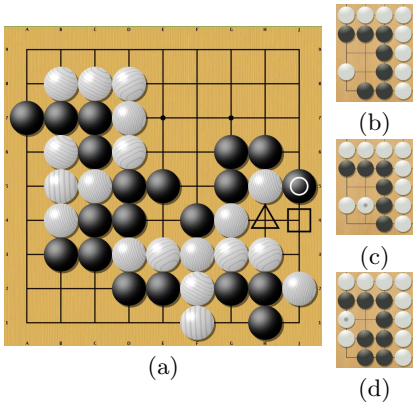


Fig. 4. In Figure (a) (a real game played and lost by MoGo), MoGo (white) without specific modification for the *nakade* chooses H4; black plays J4 and the group F1 is dead (MoGo looses). The right move is J4; this move is chosen by MoGo after the modification presented in this section. Examples (b), (c) and (d) are other similar examples in which MoGo (as black) evaluates the situation poorly and doesn't realize that his group is dead. The modification solves the problem.

being straightforward: due to the subtle equilibrium between online learning (section 2) transient learning (section 3), the first experiments were highly negative, and become clearly conclusive only after careful tuning of parameters.

The score extends the AMAF-guided exploration (Eq. 1) as follows:

$$Score(d) = \alpha \underbrace{\hat{p}(d)}_{Online\ value} + \beta \underbrace{\hat{\hat{p}}(move)}_{Transient\ value} + \gamma \underbrace{H(d)}_{Offline\ value} + \delta P(d,n) \quad (3)$$

where the coefficients α, β, γ and δ are empirically tuned coefficients depending on $n(d)$ (number of simulations of the decision d) and n (number of simulations of the current board) with the following constraints:

Number of simulations per move	Success rate	Number of simulations per move	Success rate
9x9 board		19x19 board	
10000	52.8 % ± 0.5%		
100000	55.6 % ± 0.6 %	100 000	53.2 % ± 1.1%
300000	56.2 % ± 0.9 %		
5s/move, 8 cores	55.8 % ± 1.4 %		
15s/move, 8 cores	60.5 % ± 1.9 %		
45s/move, 8 cores	66.2 % ± 1.4 %		

Table 1. Experimental validation of the nakade modification: modified MoGo versus baseline MoGo. Seemingly, the higher the number of simulations (which is directly related to the level), the higher the impact.

- $\alpha + \beta + \gamma = 1$;
- $\alpha \simeq 0$ and $\beta \simeq 0$, i.e. $\gamma \simeq 1$, for $n(d) = 0$;
- $\beta >> \alpha$ for $n(d)$ small;
- $\alpha \simeq 1$ for $n(d) \to \infty$;
- δ only depends on n and decreases to 0 as $n \to \infty$;
- $d \mapsto P(d, n)$ converges to the constant function as $n \to \infty$; $\delta P(d, n)$ is analogous to the progressive unpruning term[6].

These rules imply that:

- initially, the most important part is the offline learning;
- later, the most important part is the transient learning (RAVE values);
- eventually, only the "real" statistics matter.

The coefficients are modified at each of the frequent improvements of the offline values and the corresponding part of the source code can be requested to the authors by email.

8 Conclusion

Our conclusions are as follows:

- As well as for humans, all time scales of learning are important:
  - offline knowledge (strategic rules and patterns) as in [8, 6];
  - online information (i.e. analysis of a sequence by mental simulations) [11];
  - transient information (extrapolation as a guide for exploration).
- Reducing diversity has been a good idea in Monte-Carlo; [15] has shown that introducing several patterns and rule greatly improve the efficiency of Monte-Carlo Tree-Search. However, plenty of experiments around increasing the level of the Monte-Carlo simulator as a stand-alone player have given negative results - diversity and playing strength are too conflicting objectives. It is even not clear for us that the goal is a compromise between these two criteria. We can only clearly claim that increasing the diversity becomes more and more important as the computational power increases, as shown in section 6.
- Whereas exploration is important when learning is unsufficiently efficient, the optimal constants in the exploration term (Eqs 2 and 3) become 0 when learning is improved. In MoGo, the constant in front of the exploration term was > 0 before the introduction of RAVE values in [11]; it is now 0.

Acknowledgements. We thank Bruno Bouzy, Rémi Munos, Yizao Wang, Rémi Coulom, Tristan Cazenave, Jean-Yves Audibert, David Silver, Martin Mueller, KGS, Cgos, and the computer-go mailing list for plenty of interesting discussions. Many thanks to the french federation of Go and to Recitsproque for having organized an official game against a high level human; many thanks also to the

several players from the French Federation of Go who accepted to play and discuss test games against MoGo, and also Catalin Taranu, 5th Dan Pro, for his comments after the IAGO challenge. We thank Antoine Cornuejols for his valuable comments on a preliminary version of this paper.

References

1. Rajeev Agrawal. The continuum-armed bandit problem. *SIAM J. Control Optim.*, 33(6):1926–1951, 1995.
2. P. Auer, N. Cesa-Bianchi, and C. Gentile. Adaptive and self-confident on-line learning algorithms. *Machine Learning Journal*, 2001.
3. Jeffrey S Banks and Rangarajan K Sundaram. Denumerable-armed bandits. *Econometrica*, 60(5):1071–96, September 1992. available at http://ideas.repec.org/a/ecm/emetrp/v60y1992i5p1071-96.html.
4. Bruno Bouzy. Associating domain-dependent knowledge and monte carlo approaches within a go program. In K. Chen, editor, *Information Sciences, Heuristic Search and Computer Game Playing IV*, volume 175, pages 247–257, 2005.
5. Bruno Bouzy and Guillaume Chaslot. Bayesian generation and integration of k-nearest-neighbor patterns for 19x19 go. *In G. Kendall and Simon Lucas, editors, IEEE 2005 Symposium on Computational Intelligence in Games, Colchester, UK*, pages 176–181, 2005.
6. G.M.J.B. Chaslot, M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. 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.
7. Pierre-Arnaud Coquelin and Rmi Munos. Bandit algorithms for tree search. In *Proceedings of UAI'07*, 2007.
8. Rémi 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.
9. Rémi Coulom. Computing elo ratings of move patterns in the game of go. In *Computer Games Workshop, Amsterdam, The Netherlands*, 2007.
10. Varsha Dani and Thomas P. Hayes. Robbing the bandit: less regret in online geometric optimization against an adaptive adversary. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 937–943, New York, NY, USA, 2006. ACM Press.
11. Sylvain Gelly and David 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.
12. L. Kocsis and C. Szepesvari. Bandit-based monte-carlo planning. *ECML'06*, 2006.
13. T. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6:4–22, 1985.
14. Liva Ralaivola, Lin Wu, and Pierre Baldi. Svm and pattern-enriched common fate graphs for the game of go. *ESANN 2005*, pages 485–490, 2005.
15. Yizao Wang and Sylvain 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.