Intelligent Agents for the Game of Go

Jean-Baptiste Hoock, Chang-Shing Lee, Arpad Rimmel, Fabien Teytaud, Olivier Teytaud, Mei-Hui Wang

Intelligent Agents for the Game of Go

Jean-Baptiste Hoock *, Chang-Shing Lee**, Arpad Rimmel*,
Fabien Teytaud*, Olivier Teytaud*, Mei-Hui Wang**
*TAO (Inria), LRI, UMR 8623 (CNRS - Univ. Paris-Sud),
bat 490 Univ. Paris-Sud 91405 Orsay, France, teytaud@lri.fr,
**Department of Computer Science and Information Engineering,
National University of Tainan, Taiwan, leecs@mail.nutn.edu.tw

teytaud@lri.fr, leecs@mail.nutn.edu.tw

Abstract

Monte-Carlo Tree Search (MCTS) is a very efficient recent technology for games and planning, particularly in the high-dimensional case, when the number of time steps is moderate and when there is no natural evaluation function. Surprisingly, MCTS makes very little use of learning. In this paper, we present four techniques (ontologies, Bernstein races, Contextual Monte-Carlo and poolRave) for learning agents in Monte-Carlo Tree Search, and experiment them in difficult games and in particular, the game of Go.

I Introduction

Monte-Carlo Tree Search (MCTS) was recently proposed [1, 2, 3] for decision taking in discrete time control problems. It was applied very efficiently to games [4, 5, 6, 7, 8] but also to planning problems and fundamental artificial intelligence tasks[9, 10]. It clearly outperformed alpha-beta techniques when there was no human expertise easy to encode in a value function. In this section, we will describe MCTS and how it allowed great improvements for computer Go. Section II shows the strengths and limitations of MCTS, and in particular, the lack of learning. There are, however, a few known techniques for introducing learning: Rapid-Action Value Estimate (RAVE) and learnt patterns (both well-known now, and discussed below); our focus is on more recent and less widely-known learning techniques introduced in MCTS. The next two sections will show these less standard applications of supervised learning within MCTS: Section III will show how to use past games for improving future games, and section IV will show the inclusion of learning inside a given MCTS run. Section V will be the conclusion.

Presentation of Monte-Carlo Tree Search

This part presents MCTS and the main improvements of the initial algorithm. A more detailed presentation can be found in [3] which describes Upper-Confidence-Trees (UCT), the most well known variant of MCTS. The idea is essentially (1) performing many random simulations from the current state (2) biasing these random simulations depending on the results so that a player increases (respectively decreases) the probability of move m in state s when the percentage of games won with m played in s increases (respectively decreases). This idea is illustrated in Algorithm 1.
MCTS is very convenient. One must first implement a simulator (which is necessary for nearly all optimization algorithms), and then one just has to implement:

- a memory of 3-uples (state,action,total rewards), which is quite useful for understanding what happens in the system;

Algorithm 1 The UCT algorithm in short. $nextState(s, m)$ is the implementation of the rules of the game, and the ChooseMove() function is defined in Algorithm 2.

```
d =UCT(situation s_0, time t)
while Time left > 0 do
  s = s_0   // start of a simulation
  while s is not terminal do
    m =ChooseMove(s)
    s =nextState(s, m)
  end while
    // the simulation is over
end while
d = most simulated move from s_0 in simulations above
```

Algorithm 2 The ChooseMove function, which chooses a move in the simulations of UCT. k is an empirically tuned crucial parameter.

ChooseMove(situation s)
if There's no statistics from previous simulations in s **then**
  Return a move randomly according to some default policy
else
  for Each possible move m in s **do**
    compute a score(m) as follows:

$$\text{average reward when choosing } m \text{ in } s + \sqrt{\frac{k \times \log(\text{nb of simulations in } s)}{\text{nb of simulations of } m \text{ in } s}}. \quad (1)$$

  end for
  Return the move with highest score.
end if

- the chooseMove function (Algorithm 2), which is trivial (but requires a careful tuning of the constant k).

Some important improvements are as follows:

- for small-sized boards, automatic building of opening books by MCTS on top of MCTS[11];
- multithreaded implementations[12, 13, 14, 15];
- message-passing implementations on clusters without master/slaver architectures [16, 13]; these authors believe that this approach provides better results than the master/slave version[17] but this issue is still controversial;
- biasing Equation 1 as follows:

$$\text{average reward when choosing } m \text{ in } s + H(m,s)/(C + \text{nb of sims in } s)^{\zeta} + \sqrt{\frac{k \times \log(\text{nb of simulations in } s)}{\text{nb of simulations of } m \text{ in } s}}. \quad (2)$$

  for $\zeta \simeq 1$, and where $H(m, s)$ is a heuristic estimate, either:

  - learnt on databases thanks to patterns [18, 4];
  - modified by human expertise [19].

  We refer to references above for more details on the handcrafted heuristics (which basically reflect human expert knowledge, known as lines of influence, good kogeima, line of death, and other concepts known by go players) or on the automatically learnt pattern values (which are statistics on 3x3, 4x4, ... patterns in professional games).

- biasing Equation 2 by another term, termed the RAVE term[5, 6]:

$$\alpha \times \text{average reward when choosing } m \text{ in } s \\ +\beta \times R(m,s) + \gamma \times H(m,s) + \\ \sqrt{\frac{k \times \log(\text{nb of simulations in } s)}{\text{nb of simulations of } m \text{ in } s}}. \quad (3)$$

  where $R(m,s)$ is the average reward when the player to play is the first who plays in m *after* state s (and not necessarily in s), and where α, β and γ depend on the number of simulations and verify:

$$\begin{array}{lll} \alpha & \rightarrow & 1 \text{ as the number of simulations in } s \text{ goes to infinity} \\ \beta & \rightarrow & 0 \text{ as the number of simulations in } s \text{ goes to infinity} \\ \alpha & \simeq & 0 \text{ when the number of simulations in } s \text{ is small} \\ \beta & \simeq & 0 \text{ when the number of simulations in } s \text{ is small} \\ \beta & >> & \alpha \text{ when the number of simulations in } s \text{ is moderate} \\ \gamma & \simeq & C_1/\log(C_2 + \text{number of simulations in } s). \end{array}$$

  This involves several important constants, and provides very good results when compared to the naive Equation 1.

MCTS performs incredibly well in various problems. Nonetheless, if real parameters can be tuned empirically, some "big" parameters, namely the default policy (see Algorithm 2) and the heuristic $H(.)$, are difficult to choose; both are biases in the agents performing the simulations. This paper is devoted to techniques aimed at automatically or adaptively choose this default agent.

II Strength, Limitations and the Need for Learning

MCTS algorithms are extremely free of expert knowledge and extremely free of guidance from tactical solving. This might be a strength: For example, it has often been said that MCTS algorithms are strong for "aji", i.e. for taking into account the influence of dead stones. If MCTS was strongly based on a life and death solver, it might underestimate dead stones. Also, MCTS is extremely strong for problems like so-called Ishi-No-Shita[20], i.e. problems where captures and recaptures at the same locations make the situation extremely unclear whenever it's a very localized fight. However, these strengths do not compensate some big weaknesses when compared to human players. This will be detailed below.

A *Ishi-No-Shita and Nakade*

To the best of our knowledge, the complexity of "Ishi-No-Shita" (Fig. 6, bottom right), i.e. tsumegos in which captures and recaptures occur inside a first capture, is not known. This is an interesting question as it might be part of a more general question: which complexity classes are easy for humans and which are not ? To the best of our knowledge:

- The reading (solving) of Ishi-No-Shita is very difficult and somehow unnatural for humans;
- Computers are not disturbed by the strange structure of these situations and are particularly strong in this case (according to [21]).

Nakade is a particular case in which a player builds a group A inside an opponent group B; A will be killed, but the liberties of B will be reduced by the stones used for killing A so that B will be dead. Nakade situations are not easily handled in MCTS unless they are the only fight (as usual, MCTS has troubles for mixing the solutions of several simultaneous fights - it does not back up the solving of local tactical fights). It can, therefore, also be said that computers are weak for Nakade. Nakade can be

small, therefore, there can be a Nakade in a corner of a 9x9 board whenever the rest of the board is non trivial; they are in fact strong for Nakade as well as Ishi-No-Shita, but only as long as it's the only fight - they don't backup the result for keeping it in mind for all their simulations.

B *Limitations in Openings*

Openings are the result of a long experience. Even professional players can play very weak opening in a format in which they play for the first time, e.g. when they switch to 13x13 or 9x9 instead of the classical 19x19. MCTS methods have no memory of previous games in their most classical formulation, therefore, it is important to make them learn from their past games, or, as humans do, by watching other games. This can be done by the use of ontologies, which will be discussed in Section A.

C *Limitations in Tactical Search*

The use of tactical search is efficient in Monte-Carlo methods[22] (here we mean Monte-Carlo methods and not Monte-Carlo Tree Search methods, i.e. algorithms without the "chooseMove" functions), but not yet in MCTS. As a consequence, MCTS methods are weak in some special situations which require some specific knowledge. There's no current MCTS program able to play both situations correctly like Fig. 1(left), i.e. a semeai which must be played immediately, and situations like Fig. 1(right), i.e. a semeai which must *not* be played immediately.

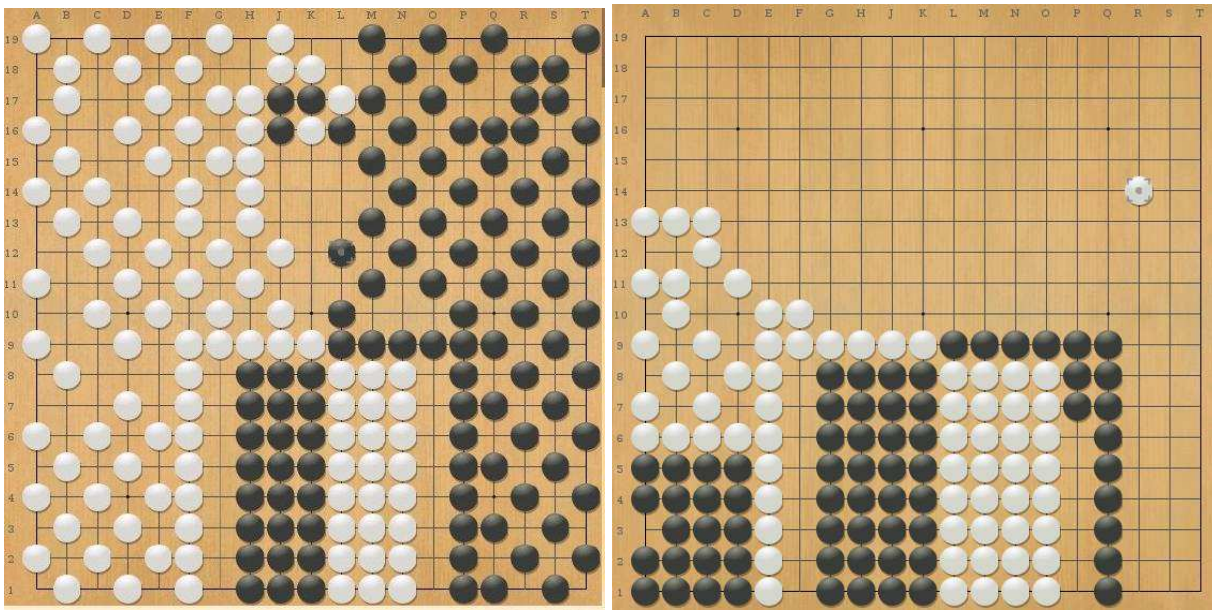


Figure 1: Left: a semeai, white to play. White can kill the big black group J4 (and make its own group M4 live) just by filling its 8 liberties G1-G8. If white does not play in the semeai, black can do it and kill the group M4: therefore, in this example, white must play G1 or G2 or ... or G8 and nothing else. Right: Another semeai. Black can kill the white group, but it's not urgent to play in the semeai: black is in advance of two liberties and should therefore play somewhere else. Playing P1 or P2 or ... or P6 is here a big mistake for black.

Such deep limitations of the method lead to a limited scalability of MCTS, i.e. a limited improvement of performance when increasing the computational power. This contradicts what has often been said

in the early years of MCTS. This is shown in Fig.1. These results show a decrease of scalability as computational power increases. All results are for Chinese rules, komi 7.5. The significance of the results is very clear from the standard deviations. Many people believe that the way for solving

Table 1: Scalability of MCTS for the game of Go.

N =Number of simulations	Success rate of 2N simulations against N simulations in 9x9 Go	Success rate of 2N simulations against N simulations in 19x19 Go
1 000	71.1 ± 0.1 %	90.5 ± 0.3 %
4 000	68.7 ± 0.2 %	84.5 ± 0.3 %
16 000	66.5 ± 0.9 %	80.2 ± 0.4 %
256 000	61.0 ± 0.2 %	58.5 ± 1.7 %

semeai situations is to include supervised learning; learning that, in Fig. 1, all simulations in which white does not play G1-G8 as soon as black attacks the white group lead to the death of the white group. Such a supervised learning of simulations would be a general tool and it would not be restricted to Go. We will see several tools in that direction in Section IV.

III Learning From Past Games: Ontologies and Races

In this Section, we will show two different techniques for learning from past experience.
In Section A, we will use suggestions by strong players, who accepted to comment games lost by MoGo, for building an ontology of Go openings. Then, this ontology will be used for modifying the openings. This is related to teaching by imitation, which is often used by Go players who reproduce professional games.
Second, in Section B, we will select the best patterns for the tree part (the function H proposed in Eq. 2, see Section I) by Bernstein races, thanks to many trials and corrections. This is somehow "brute-force" in the sense that it is based on tedious trial and error of each pattern; it has the advantage of being proved, stable and efficient. It is somehow similar to the process by which human players learn patterns, by a long experience of playing.

A *Ontologies: Improving the Openings by Using Past Experience*

Agent-based systems embedded into the ontology are increasing being applied in a wide range of areas. For example, the Multi-agent Systems Group (GruSMA) team designed and implemented a Healthcare Services multi-agent system to help doctors reduce error at each diagnostic and treatment stage [23]. [24] developed an ontology model to represent the Capability Maturity Model Integration (CMMI) domain knowledge to effectively summarize the evaluation reports for the CMMI assessment. Orgun and Vu [25] proposed an electronic Medical Agent System (eMAGS) with an ontology based on a public health message standard to facilitate the follow of patient information across a whole healthcare organization. Lee, Wang, and Chen [26] also developed an ontology-based intelligent decision support agent to evaluate the performance of each project member to assist in introducing project monitoring and control process area of CMMI. [27] also proposed ontology-based multi-agents to evaluate the diet health status based on the constructed common Taiwanese food ontology and the personal project ontology.
This section presents a developed ontology-based intelligent agent in MCTS for computer Go application. We employ features derived from professional Go players domain knowledge to transform them into the opening-book sequence and represent them by a computer Go ontology. Afterward, the domain experts validate the built ontology. The developed computer Go ontology has been verified through the invited games for computer Go programs playing against human Go players.

1) *Ontologies for Opening Books*

The building of an ontology for the knowledge of opening-books in the game of Go is illustrated in Fig. 2. The first step is to invite Go players to play against computer Go programs via a Go-playing graphic interface such as Kiseido Go Server (KGS) or Go Graphical User Interface (GoGui). Once the game is started, the records of board games are stored by following the Smart-Go Format (SGF). The records of the Go games are stored into the SGF files repository. Then, the linguistic descriptions, including very good (VG) move, good (G) move, uncertain (U) move, bad (B) move, and very bad (VB) move, on each move and alternative branches are given by the invited Go players via MultiGo software or talking to the side assistant. Fig. 2 shows an example of the fuzzy sets for fuzzy variable Move-Score = {VeryBad, Bad, Uncertain, Good, VeryGood}, which indicates that there are five fuzzy sets, including VeryBad, Bad, Uncertain, Good, and VeryGood, to describe the score of the move.

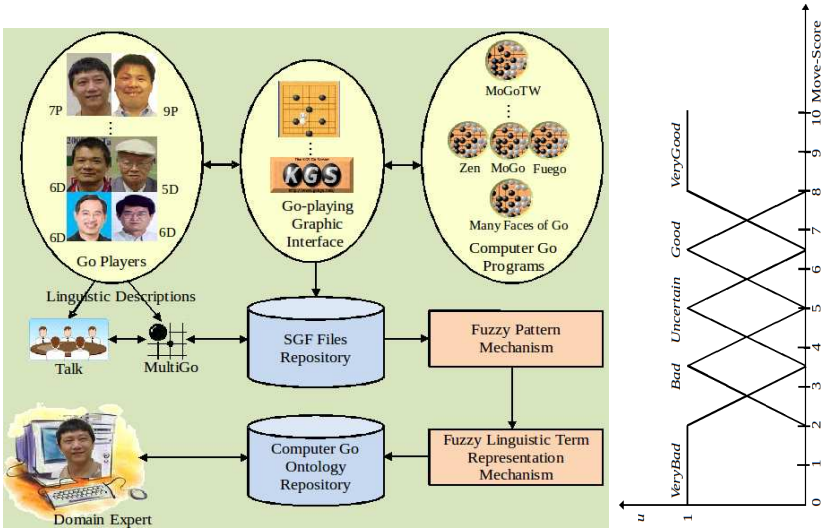


Figure 2: Left: Constructing the ontology to express some knowledge on the game of Go. Right: example of the fuzzy sets for fuzzy variable Move-Score.

The opening-book sequences are extracted based on the SGF files storing in the SGF files repository and obtained through the fuzzy pattern mechanism. The fuzzy pattern is used to present the opening-book sequence for the fuzzy ontology model. The fuzzy pattern template and one example are given in Algorithm 3, where Fuzzy Linguistic Term denotes the linguistic meaning of the fuzzy set described by various human Go players for the same fuzzy pattern or different fuzzy patterns. Algorithm 3 indicates that the 7P Go player (Ming-Chi Cheng) considers B13 is a VeryGood move, but the 6D Go player (Biing-Shiun Luoh) regards B13 as Good move, which means that different Go players have different thinking and linguistic descriptions of the same pattern. The fuzzy linguistic term is used to represent the degree of goodness for each opening-book sequence via the fuzzy linguistic term representation mechanism. Different Go players maybe give different linguistic descriptions for the same opening-book sequence. Additionally, Fig. 3 illustrates part of fuzzy patterns of patterns 1 and 6 given by Cheng. The linguistic descriptions of the moves given by Cheng and Luoh are written in blue and red colors, respectively. Fig. 3 also shows there are different descriptions for different Go players for the same pattern. Finally, the computer Go ontology can be built by integrating fuzzy pattern and fuzzy linguistic term, and the domain experts validate and verify the correctness of the constructed computer Go ontology.

There are eight patterns given by Cheng (patterns 1-4 are for Black, and patterns 5-8 are for White). Each pattern has several branches to reflect parts of countless variations in Go. The details of the eight given patterns, but excluding some of the branches due to length constraints, are shown in Fig. 4.

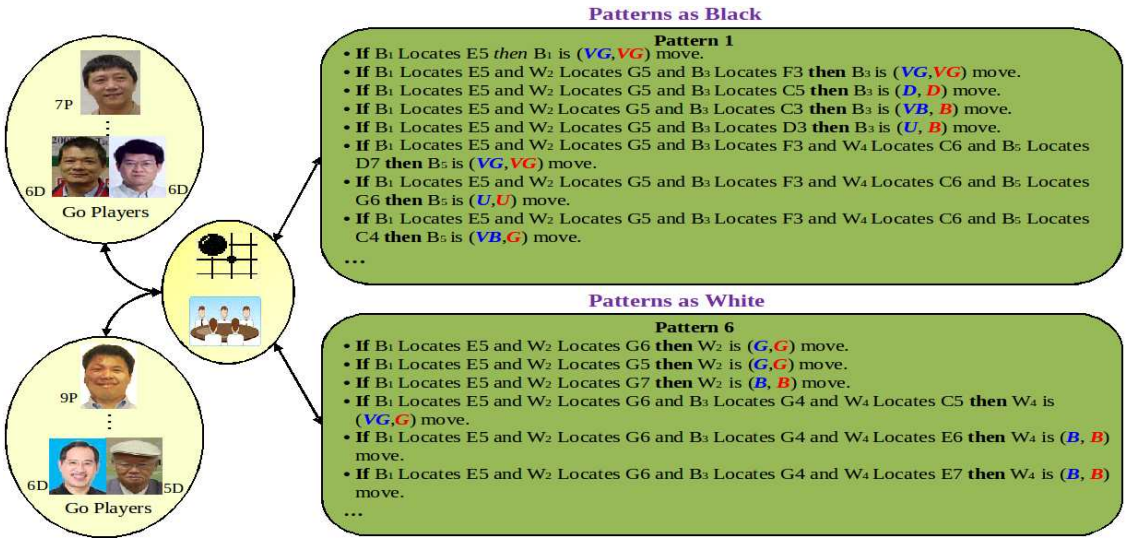


Figure 3: Part of fuzzy patterns for patterns 1 and 6.

Algorithm 3 Fuzzy pattern template and given example.

Fuzzy Pattern:
if B1 Locates L1 and W2 Locates L2 and ... and Bn-1 Locates Ln-1 and Wn Locates Ln, and Bn+1 Locates Ln+1 **then**
Bn+1 is *Fuzzy Linguistic Term* Move
end if
Given Example:
Go Player 1 (Ming-Chin Cheng, 7P)
if B1 Locates E5 and W2 Locates G5 and B3 Locates F3 and W4 Locates C6 and B5 Locates D7 and W6 Locates F7 and B7 Locates E7 and W8 Locates F6 and B9 Locates C4 and W10 Locates F4 and B11 Locates E4 and W12 Locates G3 and B13 Locates G2 **then**
B13 is VeryGood Move.
end if
Go Player 2 (Biing-Shiun Luoh, 6D)
if B1 Locates E5 and W2 Locates G5 and B3 Locates F3 and W4 Locates C6 and B5 Locates D7 and W6 Locates F7 and B7 Locates E7 and W8 Locates F6 and B9 Locates C4 and W10 Locates F4 and B11 Locates E4 and W12 Locates G3 and B13 Locates G2 **then**
B13 is Good Move.
end if

According to Cheng and Luoh, Black 1-15 in Pattern 1 of Fig. 4 are all VeryGood moves for Black, and it is also a typical pattern for Black to definitely win. Another example is the pattern 5 in Fig. 4, according to Cheng, White 16 is a Good move. But, Luoh thinks of White 16 as a VeryGood move. For Pattern 5, White is definitely a sure-win in such a situation. Table 2 lists a more detailed analysis of the given patterns 1-8 from Cheng.

Figs. 5(a) and 5(b) show the fuzzy linguistic term representation for fuzzy patterns 1 and 6, respectively. Fig. 5(a) illustrates the sequence of the moves and the comments on each move for pattern 1 as Black. It could be divided into two parts, that is, part 1 records the fuzzy pattern and part 2 stores the fuzzy linguistic terms given by domain experts (DEs). However, each Go player has different comments on the same sequence of moves so that the part 2 will have various fuzzy linguistic terms to represent the comments given by various domain experts like DE_1, DE_2, ..., and DE_N. Moreover, each comment provided by the domain expert could be a fuzzy set, that is, a linguistic term. Hence, Fig. 5(a) shows that the first moves comments given by DE_1, DE_2, ..., and DE_N are "VeryGood", "VeryGood", ... and "FSN_1", respectively. In Fig. 5, the linguistic descriptions marked in blue and red are given by Cheng and Luoh, respectively. This figure, 5, also indicates that "FSN_1", ... and "FSN_{19}", represent the

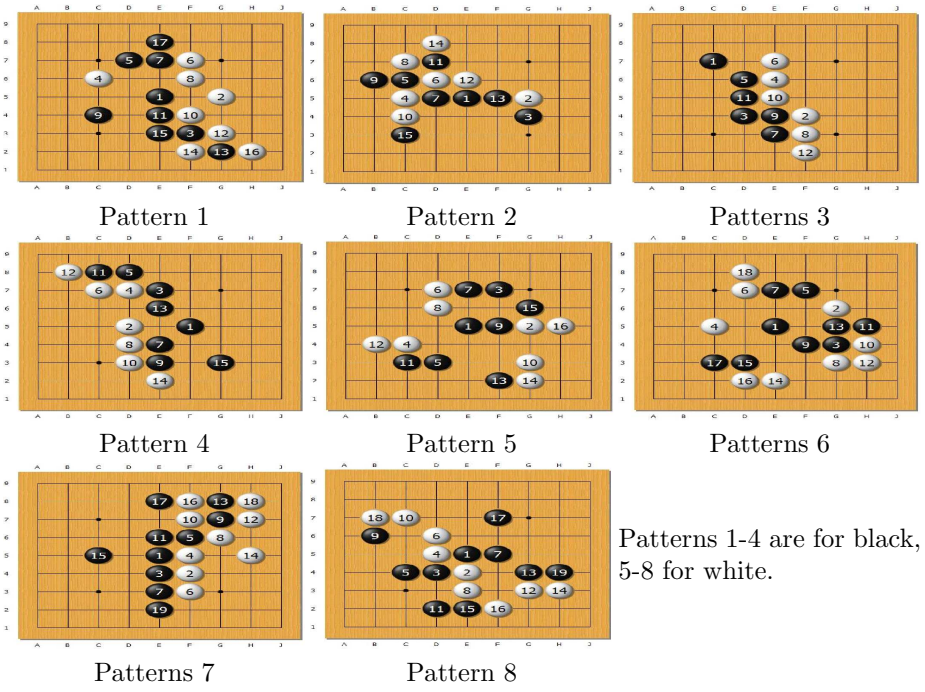


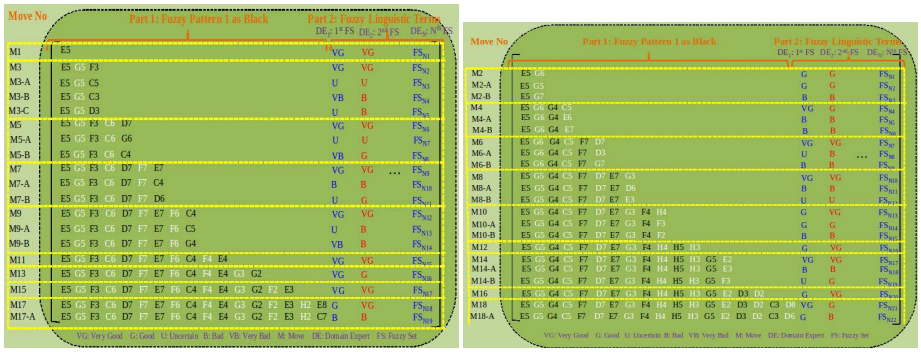
Figure 4: Patterns given by Cheng.

Table 2: Comments made by Cheng for the eight given patterns (see Fig. 4).

As Black:	As White:
•Pattern 1: It is a typical case in which Black definitely wins the game if Black has this pattern. •Pattern 2: Black 15 is a high-level move from the domain knowledge of professional Go players. •Pattern 3: Black definitely loses when the game comes to Move 11. Additionally, it is very hard to win against professional Go players if Black only plays one VG move in the whole game. •Pattern 4: Black definitely wins when Black 13 plays at E6. On the other hand, if Black 15 answers G3 after White 14 (E2), it will turn Black into a loss.	•Pattern 5: White definitely wins the game. •Pattern 6: White definitely wins the game because White 8 establishes the basis of White winning the game. •Pattern 7: Black definitely wins the game, which is caused by three uncertain moves, including White 2 (F4), White 6 (F5), and White 8 (G6). If White 8 answers at E6 instead of G6, White may have the chance to win. •Pattern 8: White plays badly for ko-fight, living, and death. The reason is that White 12 (G3) and White 14 (H3) cause White stone group at the bottom-right corner to die.

given linguistic descriptions of fuzzy sets by DE_N for the M1,M3,..., and M17-A move.

In Go, rank indicates a player's skill in the game. It is divided into two groups, i.e. amateur Go players and professional Go players. The rank for an amateur Go player could be 1st dan, 2nd dan, ..., and 7th dan, where dan could be abbreviated D. The players skill increases from 1st dan (1D) to 7th dan (7D). For a professional dan grade, it is abbreviated P, and the best rank is 9th dan (9P). According to the phases of Go, it is composed three phases, namely an opening game, a middle game, and an end game. If the Go player, no matter a human or a computer Go, is able to do an excellent opening, the chances of winning will be great, especially playing such a small 9x9 board. As a result, it is very important for a computer Go to construct an excellent opening-book, if a computer Go would like to challenge the top human Go player. On the other hand, except for VeryGood moves and Good moves, the other types of moves, namely Uncertain move, Bad move, and VeryBad move, are also necessary to develop a computer Go assessment system in the future. However, the opening sequence varies with the stones color that you hold. For this reason, the computer Go should exist in two opening sequences, that is, one is as Black, and another is as White. If there is such an ontology existing to represent



a: Fuzzy Pattern 1 b: Fuzzy Pattern 2

Figure 5: Fuzzy linguistic term representation for (a) fuzzy pattern 1 and (b) fuzzy pattern 2.

the above-mentioned information, then computer Go will quickly and easily learn and understand the opening sequences that are recommended by domain experts.

2) *Game Results of the 2010 Invited Game MoGoTW vs. Human Go Player*

This study constructed a platform for the Go games held at National University of Tainan (NUTN) and Haifong Weiqi Academy, Taiwan, on Mar. 21 and Apr. 2, 2010, respectively. All games are 9x9 games, Chinese rules, with komi 7.5 and 45 minutes per side. Additionally, all games follow the Chinese rule. On the invited games, MoGo or MoGoTW ran on four types of different machines, including a DELL PowerEdge R900 with 16 cores, NUTN-Mini-Cluster with 24 cores, HP DL785G6 with 16 cores, NUTN-Mini-Cluster with 16 cores, and IBM x3850 with 8 cores. Fig. 8 shows the comments given by Ming-Chi Cheng on some of the games on Mar. 12 and Apr. 2. Ming-Chi Cheng is invited to give comments on the games. Cheng was born in Taiwan in 1965 and went to Japan to learn Go when he got the scholarship of the Ing Chang-Ki Weichi Educational Foundation in 1978. He became a 1P and 7P professional Go player in 1982 and 1995, respectively, and returned to Taiwan to popularize the Go education at Tainan city in 2000. He is currently the president of the Tainan Go association.

B *Selecting Agents by Bernstein Races*

In this section, we present an automatic tool for improving MCTS by adding "good" biases (the function H in section I). This is directly in competition with the learning of patterns on databases as in [4]. MoGo has already "good" biases thanks to both automatically extracted and handcrafted patterns [19]. But it takes time and it is boring for programmers. That's why we have automatized this part by randomly creating a set of patterns. In our context, a pattern is seen as a mutation and an agent a is seen as a program P plus a mutation m. The first part presents the different problems raised by the automatization and which should be solved (for example, the MSHT effect defined below). The second part proposes an algorithm to solve these different problems and the last part shows some results of the algorithm applied on the program MoGo.

1) *The Multiple Simultaneous Hypothesis Testing (MSHT) Effect*

The automatization of the integration of patterns in the program MoGo is based on mutations of agents. When mutations of an agent are tested, there are two main troubles : (1) the price of the evaluation of an agent and (2) the size of the huge set of possible agents. In many cases, one more trouble comes from the fact that the evaluation is noisy. This happens when the fitness function (function that evaluates the quality of an agent) is stochastic. When evaluating an agent, one spends time (supposed constant)

testing it, and gets a noisy reward $fitness(a) \in \{0,1\}$ (equal to 1 with probability $\mathbb{E}fitness(a)$): we are looking for the optimum in Eq. 4.

$$\arg \min_{a \in \text{possible agents}} \mathbb{E}fitness(a). \tag{4}$$

When mining a huge set of agents in an uncertain framework (noisy optimization), there are two main issues:

- *load balancing:* which agents are to be tested now ?
- *statistical validation:* which agents should be validated ?

The statistical validation is not trivial; whenever each agent is tested rigorously, e.g. with confidence 95 % (i.e. 5% of probability of error), we will have erroneous validations very frequently if millions of agents are tested (with just 100 agents, each of them being equivalent or a bit worse than the baseline, we have a probability 99.4% ($= 1-(1-0.95)^{100}$) of erroneous validation). This is known as the *Multiple Simultaneous Hypothesis Testing (MSHT) effect.*

Moreover, two mutations of an agent cannot be validated simultaneously. The Bernstein race is a typical one, following [28], except that we want to validate one and only one mutation of an agent and not to take the sum of two mutations of the agent, because in our framework it is known[29] that two good mutations do not necessarily cumulate, in the sense that if P is a program (an agent), and if $P+m$ is the agent after applying modification m; in some cases, the situation described by Equations 5-6 can occur:

$$\mathbb{E}f(a1 = P + m_1) > \mathbb{E}f(P), \mathbb{E}f(a2 = P + m_2) > \mathbb{E}f(P) \tag{5}$$

$$\text{and yet } \mathbb{E}f(a3 = P + m_1 + m_2) < \mathbb{E}f(P). \tag{6}$$

At each step, at most one agent is validated. But how to choose the next agent to be tested ?

2) *Principle*

Bandits have been used for a while for the load balancing problem. However, to the best of our knowledge, they have not been used yet to solve load balancing and statistical validation simultaneously. We have investigated the use of racing algorithms ([28]), a particular form of bandit algorithms covering simultaneously the load balancing and the statistical validation.

The statistical validation is the fact to accept or reject a mutation of an agent. This validation is based on bounds, computed at some points in the race. An agent is accepted if and only if its lower

Algorithm 4 Functions for computing the confidence intervals. # denotes the cardinal operator.

Function $computeBounds(a, pop, \delta)$ (variant 1)
// Parameters:
// a a tested agent.
// pop the population of mutations.
// δ a risk level.
Static internal variable: $nbTest(a)$, initialized at 0.
Let $n = n(a)$ be the number of times a has been simulated.
Let r be the total reward over those n simulations.
$nbTest(a) = nbTest(a) + 1$
Let $lb(a) = r/n - deviation_{\text{Bernstein}}\left(\delta/(\#pop \times \pi^2 nbTest(a)^2/6), n\right)$.
Let $ub(a) = r/n + deviation_{\text{Bernstein}}\left(\delta/(\#pop \times 2^{\pi^2 nbTest(a)^2/6}), n\right)$.

bound lb is greater than 0. We compute bounds by taking account of the MSHT effect. In particular, the equation depends on two parameters :

- $nbTest(a)$ the number of times the agent a has been tested (i.e. the number of times $computeBounds(a, \dots)$ has been called); this is incremented at each call to $computeBounds(.)$.
- n the number of times the agent a has been simulated.

Algorithm 5 is the complete algorithm for testing a set of mutations; each call to Algorithm 5 calls Algorithm 4 multiple times.

Algorithm 5 The Bernstein Race algorithm (or RBGP algorithm), which is used in our experiment.

BernsteinRace(pop,δ).
while $pop \neq \emptyset$ **do**
  // Parameters:
  // pop a population of agents, obtained by mutations of an initial agent P.
  // δ a risk level.
  Select agent a belonging to pop
  Let n be the number of simulations of agent a.
  Simulate a n more times (*i.e.* now a has been simulated $2n$ times).
  //this ensures $nbTests(a) = O(\log(n(a)))$
  $computeBounds(a, pop, \delta)$
  if $lb(a) > 0$ **then**
    Return individual corresponding to agent a.
  else if $averagefitness(s) < 0.004$ or ($averagefitness < 0.006$ and $n > 10^5$) **then**
    $pop = pop \setminus \{a\}$ a is discarded.
  end if
end while
Return "there's no good agent in the offspring"

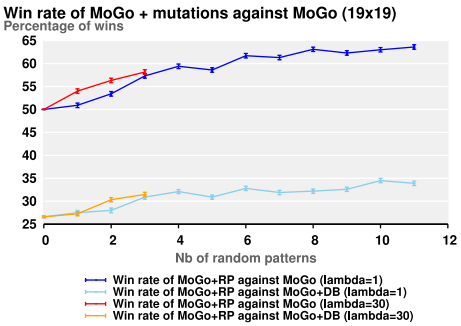
3) *Experiments*

Algorithm 6 calls Algorithm 5 multiple times; it is just a restart on top of Algorithm 6. In the terminology of evolutionary algorithms, we can say that Algorithm 6 is a $(1 + \lambda)$-algorithm.

Algorithm 6 The complete RBGP algorithm.

Let P be the current agent.
Let λ the number of agents.
for a given number of iterations **do**
  Create randomly a population pop of λ agents by mutation of P
  Apply RBGP $BernsteinRace(pop, \delta)$
  if An agent $P' \in pop$ has been accepted **then**
    P' becomes the current baseline: $P \leftarrow P'$.
  else
    The current baseline is not changed.
  end if
end for

We now present the results. First, with the big database of MoGo (MoGo+database), no agent has been validated. Seemingly, the original agent with database is too strong. The big database has then been removed; we now work on a light version, which is less efficient, but with small memory requirements and launching time. Then, Algorithm 5 has found some agents. Figure 6 shows the results. We here compare $\lambda = 30$ and $\lambda = 1$: we compare (in blue and red) MCTS on top of the learnt agent versus the initial MCTS, and (orange and cyan) MCTS on top of the learnt agent against the agent using the full database. We use two benchmarks: the different agents MoGo+mutations versus the agent MoGo, and the different agents MoGo+mutations versus the agent MoGo+database. We see here that increasing λ has little influence on the progress rate *per selected individual*. We see little influence of λ on this criterion. All results are for Chinese rules and komi 7.5. These numbers (right part) are indicative only, as we could not rerun the algorithm many times in order to have clear significance; however, the results on the left part are clearly significant, as they were validated by the statistical tests in the races.



Number of mutations	Cumulated number of evals $\lambda = 1$	Cumulated number of evals $\lambda = 30$
1	219297	63952
2	267483	103261
3	288005	266541

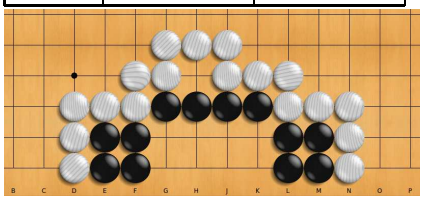


Figure 6: Left: performance improvement depending on the number of accepted mutations. Top right: cumulated number of evaluations for finding a given number of mutations. Bottom right: a beautiful Ishi-No-Shita from Denis Feldmann's web page; this problem published in Kido (a strong Japanese Go magazine) in 1996 ends in a parallel double snapback. Black plays and saves some stones.

One goal is to find the best cardinal λ of set of agents in order to have certainly a good agent. In this experiments, we present preliminary results with λ=1 and λ=30. We cannot conclude which is the best value for λ; seemingly, the results are quite similar. In both cases, results are satisfactory in the sense that performance is increasing; however, the computational power involved is huge.

If we see results for λ=1, with a few mutations (more precisely 11), we have already 64% of improvement against the light version (e.g. without the big database). The performance against the version with database remains low ($\simeq$ 34%). However, the big database contains more than 100,000 patterns - therefore, results are very encouraging from the point of view of the comparison with an equivalent number of mutations when more mutations will be included. Unfortunately, the number of evaluations before finding a good mutation increases very quickly. For example, the cumulated number of evaluations before finding the 6th mutation is 618,398 whereas for the 11th mutation, it increases to 3,277,383 evaluations.

4) *Conclusion on Bernstein Races for Agent Selection*

We applied Bernstein races for validating randomly mutated agents. We have a successful non-decreasing property of performance. However, the cost is huge, and in 19x19, we could not improve on the initial database of patterns learnt in a supervised manner as in [4]. The main success is in 9x9 (our version is moderately better, than the best version so far), or in terms of sparsity for 19x19 (we have a much better performance than the learnt database for a similar database size).

IV Introducing Learning into MCTS Agents

Over the past decade, there has been a growing interest in utilizing intelligent agents in computer games and virtual environment [30]. For example, Fogel [31] developed an evolutionary entertainment with intelligent agents to control characters in board games and other software entertainment products. Quek et al. [32] presented an evolutionary framework to simulate and study the collective outcome of public goods provisioning in an agent-based model. Lees et al. [30] addressed a central issue for high level architecture (HLA)-based games to develop the HLA-compliant game agents. Acampora, Gaeta, and Loia [33] proposed an exploring e-learning knowledge through ontological memetic agents. According to [27], 1) An agent physically distributes required knowledge to several locations; 2) An agent can model autonomous entities; 3) Agent-based systems can employ security mechanisms; 4) Agent technology has the ability to communicate and coordinate; 5) Agents can automatically discover and compose e-services; 6) Intelligent agents have

deliberate, reactive, and flexible behaviors and can learn; 7) An agents autonomous, reactive, and flexible characteristics make agents ideal for implementing ambient intelligence applications.
In this section, we will present two ways of including learning in Monte-Carlo Tree Search agents; this is about dynamically adapting agents. The previous section was devoted to learning from past experience, here, we adapt agents dynamically during a game. First, we can use RAVE values in the MC part (Section A): whereas classical RAVE values extrapolate knowledge from situations to their ancestors, we here build RAVE values in node s from simulations later than s (as in classical RAVE), but then RAVE values in s influence the Monte-Carlo simulations later than s. This is a highly interesting modification as many people were trying to do so; unfortunately, the results are positive and statistically significant, but moderate.
Second, we will present Contextual Monte-Carlo (Section B); this consists in learning (in a tiled representation) which moves are good in the Monte-Carlo part. This is quite satisfactory for the objective of introducing supervised learning, however, this does not solve the important weaknesses pointed out above (semeais and life-and-death).

A *Using RAVE Values in the MC Part*

We first explain how to use RAVE values in order to improve the Monte Carlo simulations. Then, we present the experimental results for two different applications: the game of Havannah and the game of Go.

1) *Principle*

The idea is to use the RAVE values (see Section I) as follows in the default policy (the Monte-Carlo agent):

- build a pool of the k best moves according to RAVE values.
- choose one move m from the pool.
- with a probability p, play m (otherwise, use the default policy to choose a move).

We call this modification the "poolRave" modification. This improvement is generic, i.e. independent of the application. Furthermore, it is also simple to implement if you already use RAVE values.

Two parameters are used. The first one is k which defines the number of moves contained in the pool. The second is the probability p of playing a move in the pool instead of a regular move. One important problem when trying to bias the Monte-Carlo part is to keep the diversity of the simulations. The parameter p allows us to remain not too deterministic.

2) *Experimenting poolRave on the Game of Havannah*

The rules of the game of Havannah can be found in [7, 34]. For our experiments, we measure the success rate of our bot with the previous modification against the bot without it (in that case the policy in the Monte-Carlo part consists simply in playing randomly). As said in previous section, there are two parameters in this modification. We tested different set of values. Results are presented in Table 3. We also experimented with different numbers of simulations per move in order to see the robustness of our modification.

Best results are obtained with $p = 1/2$ and a pool of size $k = 10$, reaching a success rate of 54.32% for 1,000 simulations and 54.45% for 10,000 simulations. With the same set of parameters, for 20,000 simulations we have 54.42%, so for this first application, this improvement seems to be independent of the number of simulations.

Table 3: Success rate of the poolRave modification for the game of Havannah. The baseline is the code without the poolRave modification. Standard deviations show the statistical significance of these results.

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

3) ***Experiments on the Game of Go***

The game of Go is not only a famous two-player game but also a classical benchmark for Monte-Carlo Tree Search methods. We implemented poolRave in the Go program MoGo. As the "fillboard" modification is already taking care of keeping the diversity of the simulations, we set the parameter p to 1 and experiment for different values of k. The program MoGo with the poolRave modification plays against the same program without the modification.

On 9x9 board, Chinese rules, komi 7.5, with 1,000 simulations per move, we obtain at most $51.7 \pm 0.5\%$ of victory. This result is statistically significant but not really large. One reason could be that the poolRave modification is in competition with the heuristics in MoGo.

In order to test this hypothesis, we conducted the same experiments but we removed the patterns representing the expert knowledge from MoGo. The results are shown in Fig. 7 (right). These results are for Chinese rules, komi 7.5. The results are much more impressive this time. We obtained up to $62.7 \pm 0.9\%$ of victory for a pool of size $k = 20$. We also experimented this modification in the open source Go program Fuego `http://fuego.sourceforge.net`. The experiments are done on 19x19 boards with 1000 simulations per move. We obtained up to $57.4 \pm 1.0\%$ of victory against the version without the poolRave modification, in 19x19 with patterns - as a conclusion, we could also have positive results, but with a slightly different implementation. In MoGo, results remain very moderate when patterns (and other modifications of MoGo) are kept.

4) ***Conclusion on poolRave***

The poolRave modification is generic. We proved it by using it in 3 different implementations of MCTS (with differences in implementations as discussed above). In each case, we achieved some improvements. The improvements are more significant when expert knowledge is small or absent.

While this modification is a clear improvement of Monte Carlo simulations, it does not solve the problem of semeai presented earlier in this article.

B *Contextual Monte-Carlo (CMC)*

In this section, we present another way of improving new Monte-Carlo simulations based on the results from previous simulations. Experimental results are shown for the game of Havannah.

1) *Principle of CMC*

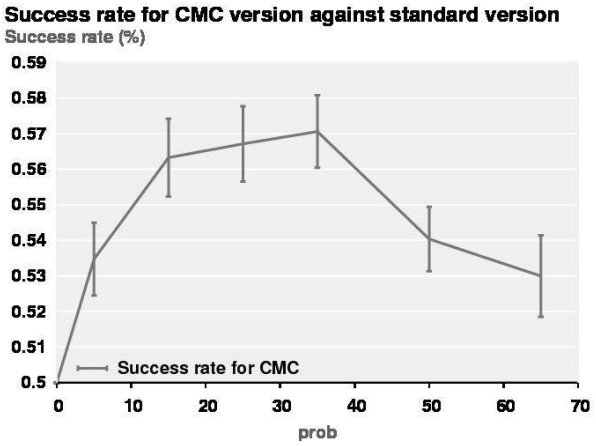
In the poolRave section, we used statistics per move. In this section we want to learn more complex patterns by using statistical information about pairs of moves. Another difference is that in poolRave we were using RAVE values whereas here, we use standard statistics.

In order to do so, we first define $L_c(a_1, a_2)$ as the ensemble of simulations where the moves a_1 and a_2 have been played by the player c. We then put in memory the average empirical reward $\hat{V}_c(a_1, a_2)$ for each $L_c(a_1, a_2)$ based on previous simulations. In a Monte-Carlo simulation, we will then play for the player c the move a that maximizes $\hat{V}_c(a, b)$. b is the last move played by the player c. In order to keep the diversity, we apply this modification with a probability p only. We call this modification "contextual Monte-Carlo".

2) *Experiments*

This modification is applied to our bot playing the game of Havannah. The experiments are realized by making the version with CMC play against the version without CMC. Each player has 1000 simulations per move. We study the impact of the value p.

The results are presented on figure 7 (left). This modification is efficient as it leads to 57% of victory



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

Figure 7: Left: Winning percentage for the CMC version against the baseline depending on the p parameter with size 5 in the Havannah game. Right: Success rate of the poolRave modification for the game of 9x9 Go, Chinese rules and komi 7.5.

against the version without modification. Those experiments also show the importance of diversity in Monte-Carlo simulations as the results become worse when the value of p increases above 40%.

3) *Conclusion on Contextual Monte-Carlo*

As the poolRave modification, contextual Monte-Carlo is generic and can be applied to any MCTS implementation. It deals with more complicated concepts as it uses pairs of moves instead of single moves. The results are also slightly better as we reach 57% of victory against 54% with poolRave for the game of Havannah. However, this modification is more difficult to implement. Even if pairs of moves contain more information than single move, it is still far from enough to solve semeai situations where sequences of dozen of moves can be required. An interesting question is to know if it is possible to solve this problem in a generic way by using statistical information from previous simulations or if a specific solver is needed; or maybe a completely different solution is required.

V Conclusion

We presented MCTS (Section I) and some important limitations (Section II). MCTS is now a widely accepted technique for discrete time decision taking in uncertain environments. Surprisingly enough, it is based on the idea of nearly no generalization: whereas many past works in reinforcement learning were based on the use of sophisticated extrapolation algorithms, e.g. using tilings or neural networks, MCTS uses essentially one value per visited state. Learning is essentially limited to averaging in UCT, and RAVE values introduce a very limited form of extrapolation. Also, a MCTS algorithm is exactly the same at the first run and after thousands of runs. We can conclude that it does not learn with experience. We presented two ways of learning from past experience:

- Ontologies of past games (using expert advices) for improving the openings (Section A);
- Improvements of patterns by intensive trials and errors (Section B).

We also proposed two ways of introducing supervised learning within a MCTS run:

- The poolRave modification, which modifies the Monte-Carlo simulations depending on the RAVE values (Section A);
- The Contextual Monte-Carlo, consisting in adapting Monte-Carlo simulations to the current board (Section B).

These two modifications somehow generalize RAVE values.
Interestingly, with these modifications, many of the human techniques for playing Go are incorporated: concepts of permuting sequences (in RAVE values), learning openings (by ontologies) thanks to advices; patterns learnt through long term experience. The poolRave and Contextual Monte-Carlo are somehow related to specialization of sequences to the current board, as when humans solve life and death problems in a game in order to simplify their further analysis. However, computers are far from the abilities of humans for solving life and death problems and incorporating such solutions within a complete strategy. Our results are statistically significant, as shown by standard deviations of our experiments. Nonetheless, they are moderate and none of these learning techniques is a revolution.
We can summarize the numerical results as follows. We have shown that the success rate of $2N$ simulations against N simulations decreases from 71% to 61% in 9x9 Go, and from 90% to 58% in 19x19 Go, when switching from $N = 1000$ to $N = 256000$. This shows a decreasing scalability of the method. Using Bernstein races for selecting patterns improved a MCTS algorithm with no learnt patterns by 64% in 19x19, and some significant improvement in top implementations in 9x9 (see [29]), but not at all a MCTS algorithm with learnt patterns in 19x19. PoolRave provided 54% improvement in Havannah, 62% on a MCTS for Go without patterns in the Monte-Carlo, and 57% on the MCTS implementation for Go Fuego, but only a disappointing 52% on MoGo with patterns. Contextual Monte-Carlo was successfully applied to the game of Havannah, in a less optimized implementation than in Go. As a summary, the progress is mainly for implementations which are not too strongly optimized; they have the advantage of partially replacing expert knowledge or tuning, but, they don't provide huge improvements.
The advances in computational intelligence have contributed to the improvement in computer Go in the past years. It allowed MoGoTW to obtain the 1D, 2D, and 3D certificates awarded by the Taiwanese Go association by winning 23 out of 24 games against 1-3D amateur Go players on Mar. 21, 2010. It is difficult to compare various versions of MCTS against other techniques than MCTS, as there's currently no technique which is comparable, in terms of Go level, to MCTS algorithms. However, the game results indicate that MoGoTW still has troubles with ko-fights and life-and-death problems. Using supervised learning might be the key for this.

Acknowledgements

The authors would like to thank the National Science Council of Taiwan for financially supporting this international cooperation research project under the grant NSC97-2221-E-024-011-MY2 and NSC 99-2923-E-024-003-MY3. Additionally, this work was supported by the French National Research Agency (ANR) through COSINUS program (project EXPLO-RA ANR-08-COSI-004).

References

[1] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," *In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy*, 2006.

[2] G. Chaslot, J.-T. Saito, B. Bouzy, J. W. H. M. Uiterwijk, and H. J. van den Herik, "Monte-Carlo Strategies for Computer Go," in *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, P.-Y. Schobbens, W. Vanhoof, and G. Schwanen, Eds., 2006, pp. 83–91. [Online]. Available: http://www.cs.unimaas.nl/g.chaslot/papers/mcscg.pdf

[3] L. Kocsis and C. Szepesvari, "Bandit based Monte-Carlo planning," in *15th European Conference on Machine Learning (ECML)*, 2006, pp. 282–293.

[4] R. Coulom, "Computing elo ratings of move patterns in the game of go," in *Computer Games Workshop, Amsterdam, The Netherlands*, 2007.

[5] S. Gelly and D. Silver, "Combining online and offline knowledge in UCT," in *ICML '07: Proceedings of the 24th international conference on Machine learning*. New York, NY, USA: ACM Press, 2007, pp. 273–280.

[6] 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. [Online]. Available: http://hal.inria.fr/inria-00369786/en/

[7] F. Teytaud and O. Teytaud, "Creating an Upper-Confidence-Tree program for Havannah," in *ACG 12*, Pamplona Espagne, 2009, pp. 73–89. [Online]. Available: http://hal.inria.fr/inria-00380539/en/

[8] 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*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 49–55.

[9] P. Rolet, M. Sebag, and O. Teytaud, "Optimal active learning through billiards and upper confidence trees in continous domains," in *Proceedings of the ECML conference*, 2009.

[10] F. De Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel, "Bandit-Based Optimization on Graphs with Application to Library Performance Tuning," in *ICML*, Montréal Canada, 2009. [Online]. Available: http://hal.inria.fr/inria-00379523/en/

[11] P. Audouard, G. Chaslot, J.-B. Hoock, J. Perez, A. Rimmel, and O. Teytaud, "Grid coevolution for adaptive simulations; application to the building of opening books in the game of Go," in *Proceedings of EvoGames*. Springer, 2009, pp. 323–332.

[12] 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*, 2007, pp. 175–182.

[13] S. Gelly, J. B. Hoock, A. Rimmel, O. Teytaud, and Y. Kalemkarian, "The parallelization of Monte-Carlo planning," in *Proceedings of the International Conference on Informatics in Control, Automation and Robotics (ICINCO 2008)*, 2008, pp. 198–203.

[14] M. Enzenberger and M. Müller, "A lock-free multithreaded Monte-Carlo tree search algorithm," in *Proceedings of Advances in Computer Games 12*, 2009.

[15] R. Coulom, "Lockless hash table and other parallel search ideas," 2008, post on the computer-go mailing list.

[16] T. Cazenave and N. Jouandeau, "On the parallelization of UCT," in *Proceedings of CGW07*, 2007, pp. 93–101.

[17] H. Kato and I. Takeuchi, "Parallel monte-carlo tree search with simulation servers," in *13th Game Programming Workshop (GPW-08)*, November 2008. [Online]. Available: http://www.gggo.jp/publications/gpw08-private.pdf

[18] G. Chaslot, M. Winands, J. Uiterwijk, H. van den Herik, and B. Bouzy, "Progressive Strategies for Monte-Carlo Tree Search," in *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, P. Wang *et al.*, Eds. World Scientific Publishing Co. Pte. Ltd., 2007, pp. 655–661. [Online]. Available: papers\pMCTS.pdf

[19] G. Chaslot, C. Fiter, J.-B. Hoock, A. Rimmel, and O. Teytaud, "Adding expert knowledge and exploration in Monte-Carlo Tree Search," in *Advances in Computer Games*. Pamplona Espagne: Springer, 2009. [Online]. Available: http://hal.inria.fr/inria-00386477/en/

[20] R. Hunter, "Nakade & ishi-no-shita," *British Go journal*, vol. 128, pp. 8–12, 2002.

[21] Intelligent Go, "http://intelligentgo.org," 2010.

[22] T. Cazenave and B. Helmstetter, "Combining tactical search and monte-carlo in the game of go," *IEEE CIG 2005*, pp. 171–175, 2005.

[23] A. Moreno, A. Valls, D. Isern, and D. Sanchez, "Applying agent technology to healthcare: The crusma experience," *IEEE Intelligent Systems*, vol. 21, no. 6, pp. 63–67, 2006.

[24] C. S. Lee and M. H. Wang, "Ontology-based computational intelligent multi-agent and its application to cmmi assessment," *Applied Intelligence*, vol. 30, no. 3, pp. 203–219, 2009.

[25] B. Orgun and J. Vu, "Hl7 ontology and mobile agents for interoperability in heterogeneous medical information systems," *Computers in Biology and Medicine*, vol. 36, no. 7-8, pp. 817–836, 2006.

[26] C. S. Lee, M. H. Wang, and J. J. Chen, "Ontology-based intelligent decision support agent for cmmi project monitoring and control," *International Journal of Approximate Reasoning*, vol. 48, pp. 62–76, 2008.

[27] M. H. Wang, C. S. Lee, K. L. Hsieh, C. Y. Hsu, G. Acampora, and C. C. Chang, "Ontology-based multi-agents for intelligent healthcare applications," *Journal of Ambient Intelligence and Humanized Computing*, vol. 1, no. 2, pp. 111–131, 2010.

[28] V. Mnih, C. Szepesvári, and J.-Y. Audibert, "Empirical Bernstein stopping," in *ICML '08: Proceedings of the 25th international conference on Machine learning*. New York, NY, USA: ACM, 2008, pp. 672–679.

[29] J.-B. Hoock and O. Teytaud, "Bandit-based genetic programming," in *Proceedings of EuroGP 2010*, 2010, p. Accepted.

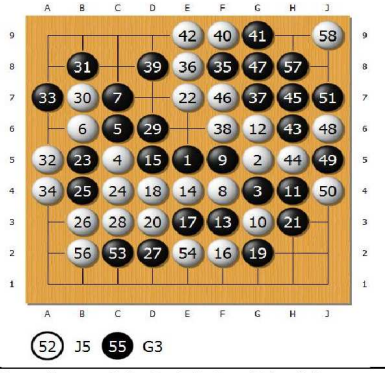
[30] M. Lees, B. Logan, and G. K. Theodoropoulos, "Agents, games, and hla," *Simulation Modelling Practice and Theory*, vol. 14, no. 6, pp. 752–767, 2006.

[31] D. B. Fogel, "Let the games begin," *IEEE Computational Intelligence Magazine*, vol. 3, no. 3, pp. 3–3, 2008.

[32] J. Mendel, L. Zadeh, E. Trillas, R. Yager, L. Lawry, H. Hagras, and S. Guadarrama, "What computing with words means to me," *IEEE Computational Intelligence Magazine*, vol. 5, no. 1, pp. 20–26, 2010.

[33] G. Acampora, M. Gaeta, and V. Loia, "Exploring e-learning knowledge through ontological memetic agents," *IEEE Computational Intelligence Magazine*, vol. 5, no. 2, pp. 66–77, 2010.

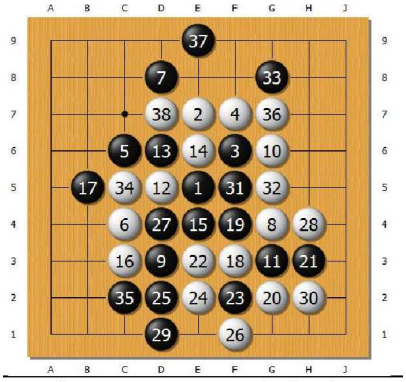
[34] Wikipedia, "Havannah," 2009. [Online]. Available: http://en.wikipedia.org/wiki/Havannah



Results: MoGoTW (Black) vs. Chun-Hsu Chou (9P). White won by resignation.

Comments:

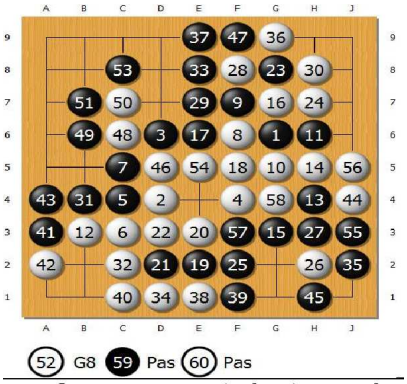
- White made a thorough counterattack with White 8 for Black 3, 5, and 7, which causes the game to be hopeless for Black.
- The opening sequence for Black 3, Black 5, and Black 7 should be improved in the future.



Results: MoGoTW (White) vs. Shih-Min Chin (7D). White won by resignation.

Comments:

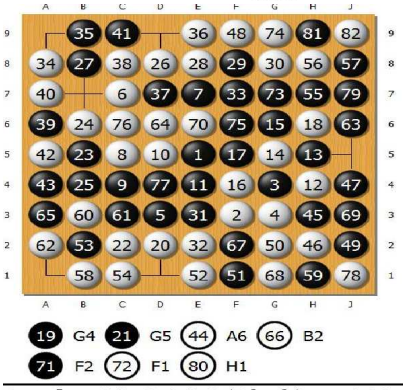
- Black 11 turned the game into a loss. If Black 11 played at 18, Black would simply win.
- White 6 is an uncertain move, and it would better if White 6 played at 22.
- White 8 is a good move.
- White 12, 14, 16, and 18 are the basis for White to win the game.
- After move 12, White plays very well.



Results: MoGoTW (White) vs. Yeh-Yang Liu (6D). White won by 0.5 point.

Comments:

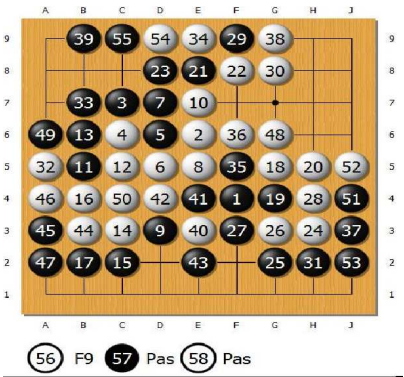
- It is surprising that Black lost by 0.5 points because Black played so perfect from Black 13 to Black 21 at this game. The game result proves that the importance of the opening-book and komi 7.5 is a big burden to Black.
- White 8 and White 10 are vital points to cause White to win.



Results: MoGoTW (Black) vs. Tai-Hsiung Yang (7D). White won by resignation.

Comments:

- Black 9 should play at 12.
- This game proves Black has a problem with the ko-fight and life-and-death. Black should respond to the ko threat on the upper-left corner using 22 instead of 21. Additionally, answering Black 37 at D7 is a significant factor for Black to lose because Black 37 let the group of White stones at the upper-left corner unconditionally alive. The recommended tactic for Black 37 is to play at 38.



Results: MoGoTW (White) vs. Tai-Hsiung Yang (7D). White won by 0.5 point.

Comments:

- Black 1 and Black 3 are seldom seen, which caused White to jump out its opening-book. This game proves that if White can avoid from entering ko-fight and life-and-death problem, it will advantage White to win.
- Black 15 didn't play well. If Black 15 played at 48, it would be hard to decide who the winner is.
- In this game, White performs well and has no serious mistakes.

Figure 8: Comments by Cheng on some of the games.