Monte-Carlo Go Reinforcement Learning Experiments

Bruno Bouzy
Université René Descartes
UFR de mathématiques et d'informatique
C.R.I.P.5
45, rue des Saints-Pères
75270 Paris Cedex 06, France
bouzy@math-info.univ-paris5.fr

Guillaume Chaslot
MICC/IKAT University of Maastricht
Faculty of General Sciences
Department of Computer Science
P.O. Box 616, 6200 MD Maastricht
The Netherlands
g.chaslot@cs.unimaas.nl

Abstract— **This paper describes experiments using reinforcement learning techniques to compute pattern urgencies used during simulations performed in a Monte-Carlo Go architecture. Currently, Monte-Carlo is a popular technique for computer Go. In a previous study, Monte-Carlo was associated with domain-dependent knowledge in the Go-playing program Indigo. In 2003, a 3x3 pattern database was built manually. This paper explores the possibility of using reinforcement learning to automatically tune the 3x3 pattern urgencies. On 9x9 boards, within the Monte-Carlo architecture of Indigo, the result obtained by our automatic learning experiments is better than the manual method by a 3-point margin on average, which is satisfactory. Although the current results are promising on 19x19 boards, obtaining strictly positive results with such a large size remains to be done.**

Keywords: Computer Go, Monte-Carlo, Reinforcement Learning

I. Introduction

This paper presents a study using Reinforcement Learning (RL) to automatically compute urgencies of moves played within random games in a Monte-Carlo (MC) Go framework. This study has three important features. First, although based on the RL theory [1], [2], [3], [4], it is mainly empirical: it is made up of three experiments, each of them being performed in the light brought by the previous one. Second, the last experiment presented here still broadened our understanding of the problem. Consequently, this work is not completed: the results achieved are promising but still below our initial ambitions. Third, this work is based on a particular architecture: the MC Go architecture of our Go playing program Indigo [5]: the performed experiments aim at improving the playing level of this program. Nevertheless, based on these three features, the goal of this paper is to show how RL contributes to the improvement of a MC Go playing program.

To this end, setting up the background of this work is necessary: section II briefly presents the state of the art of computer Go, and the point reached by Indigo project, then section III presents the MC Go architecture which can be either pure or extended with domain-dependent knowledge. Then, section IV presents the core of this study: the automatic computing of this domain-dependent knowledge. It underlines the experimental vocabulary used by section V that describes the experiments. Finally, section VI sums up the results and describes the future work.

II. Background

A. Computer games

Computer games have witnessed the enhancements done in AI for the past decade [6], and future improvements are bound to go on in the next decade [7]. For instance, in 1994, Chinook beat Marion Tinsley, the Checkers world champion [8], and Logistello beat the Othello world champion. In 1997, Deep Blue [9] beat Garry Kasparov, the Chess world champion. In 2006, solving Checkers is nearly achieved [10]. In Othello, Logistello's playing level is clearly supra-human [11]. In Chess, the best programs rank on a par with the best human players. Moreover, the combinatorial complexity of a game can be estimated with the game tree size that, in turn, can be estimated by B^L, where B is the average branching factor of the game, and L is the average game length. Table I provides the values of B^L for these games, and for Go.

Game	Checkers	Othello	Chess	Go
B^L	10^{32}	10^{58}	10^{123}	10^{360}

TABLE I
B^L Estimation.

By observing that the best Go programs are ranked medium on the human scale, at least far below the level of the best human players, a correlation between the size of the game tree and the playing level of the best programs on the human scale can be noticed. The game tree search paradigm accounts for this correlation. A classical game-tree-based playing program uses a tree-search and an evaluation function. On current computers, this approach works well for Checkers, Othello, and Chess. In these games, the search is sufficiently deep, and the evaluation function easily computed to yield a good result. On the contrary, the Go tree is too huge to yield a good result. Furthermore, the evaluation function on non-terminal positions is not well-known, and position evaluations are often very slow to compute on nowadays' computers.

B. Computer Go

Since 1990, an important effort has been made in computer Go. The main obstacle remains to find out a good evaluation function [12]. Given the distributed nature of this game, it was natural to study the breakdown of a position into sub-parts, and to perform local tree searches using intensive pattern-matching and knowledge bases [13], [14]. The best programs are sold on the market: Many Faces of Go [15], Goemate, Handtalk, Go++ [16], Haruka, KCC Igo. Consequently, the sources of these programs are not available. In 2002, GNU Go [17], an open source program, became almost as strong as these programs. Since then, this program has been used as an example to launch new computer Go projects. Various academic programs exists : Go Intellect, Indigo, NeuroGo [18], Explorer [19], GoLois [20], Magog. Some aspects of these programs are described in scientific papers: [21] for Go Intellect, [22] for NeuroGo, [23] for Explorer, [24] for Golois, and [25] for Magog.

C. Indigo project

The Indigo project was launched in 1991 as a PhD research. Indigo is a Go playing program which has regularly attended international competitions since 1998. Its main results are listed below.

- 9th KGS, 19x19, Dec 2005 (Formal: 3rd/7, Open: 1st/9)
- 8th KGS, 9x9, Formal, Nov 2005 (4th/11)
- 7th KGS, 19x19, Open, Oct 2005 (2th/7)
- 2005 WCGC, Tainan, Taiwan, Sept 2005 (6th/7)
- 10th CO, Taipei, Sept 2005 (19x19: 4th/7, 9x9: 3rd/9)
- 9th CO, Ramat-Gan, Jul 2004 (19x19: 3rd/5, 9x9: 4th/9)
- 8th CO, Graz, Nov 2003 (19x19: 5th/11, 9x9: 4th/10)
- Comp. Go Festival, Guyang, China, Oct 2002 (6th/10)
- 21st Century Cup, 2002, Edmonton, Canada (10th/14)
- Mind Sport Olymp. 2000, London, England (5th/6)
- Ing Cup 1999 Shanghai, China (13th/16)
- Ing Cup 1998 London, England (10th/17)

Participating in these events has allowed Indigo to be assessed against various opponents, which brings about keeping good and efficient methods, and eliminating bad or inefficient ones. Until 2002, Indigo was a classical Go program: it used the breakdown approach, and local tree searches with a large knowledge base. The results improved in 2003, which corresponds to the integration of MC techniques into Indigo. The historical vision of the Indigo development shows the relevance of the MC approach in computer Go. However, the effect of the knowledge approach must not be overlooked. Without knowledge, Indigo would be less strong than it is.

III. Monte-Carlo Go

This section presents the MC technique [26] for computer games, then MC Go as such, without specific knowledge, lastly MC Go associated with specific knowledge.

A. Monte-Carlo games

Monte-Carlo is appropriate for games containing randomness, for example for Backgammon in which the players throw dice. In Backgammon, although the best program used tree search instead simulations during its games, simulations are used after the games or at learning time [27] to find out new policies. MC is also adapted to games including hidden information such as Poker or Scrabble. Poki, one of the best Poker programs [28], and Maven, the best Scrabble program [29], perform simulations during their games in order to represent hidden information. For complete information games, simulations can be appropriate as well. Abramson proposed a Monte-Carlo model for such games [30]. To obtain the evaluation of a given position, the basic idea consists in launching a given number N of random games starting on this position, scoring the terminal positions, and averaging all the scores. To choose a move on a given position, the corresponding idea is the greedy algorithm at depth one. For each move on the given position, launch a given number N of random games starting on this position, score the terminal positions, and average all the scores, and finally play the move with the best mean. The obvious upside of MC is its low complexity when B and L are high: $O(B^L)$ for tree search, and $O(NBL)$ for Monte-Carlo. For complete information games, when the tree is too large for a successful tree search, simulations allow the program to sample tree sequences that reach terminal positions meaningful for evaluating the given position. By averaging the scores, evaluations on non-terminal positions are robust, which is hard to obtain with classical evalution functions based on knowledge extracted from human expertise.

B. Basic Monte-Carlo Go

In the early 1990's, the general MC model by Abramson was used on games with low complexity such as 6x6 Othello. However, in 1993, Bernd Brügmann succeeded in developping the first 9x9 MC Go program, Gobble [31]. More precisely, Gobble was based on simulated annealing [32]. In addition, to make the program work on the computers available at that time, Brügmann used a heuristic later called the all-moves-as-first heuristic [33]. Theoretically, this heuristic enables the process to divide the response time by the size of the board. In practice on 9x9, it enables the program to divide the response time by a few dozens, which is a huge speed-up, and worth considering. After a random game with a score, instead of updating the mean of the first move of the random game, the all-moves-as-first heuristic updates with the score the means of all moves played first on their intersections with the same color as the first move. Symmetrically, the all-moves-as-first heuristic updates with the opposite score the means of all moves played first on their intersections with a different color from the first move. All in all, this heuristic updates the mean of almost all the moves as if they were played first in the random game. Unfortunately, this heuristic is not completely correct because it may update with the same score two moves that have different effects depending on when they are played: before or after a capture (capture being the basic concept in Go). However, this Go-specific heuristic had to be mentioned.

Since 2002, the MC approach has gained popularity in the computer Go community, which can be explained bythe speed of current computers. The standard deviation of random games played on 9x9 boards is roughly 35. If we look for a one-point precision evaluation, 1,000 games give 68% of statistical confidence, and 4,000 games 95%. Given that

10,000 9x9 random games are possible to complete on a 2 GHz computer, then from 2 up to 5 MC evaluations per second with a sufficient statistical confidence are possible, and the method actually works in a reasonable time.

Several strategies exist to speed up the MC process. One of them is progressive pruning [28], [33]. For each move, the process updates not only the mean of a move but also the confidence interval around the mean. As soon as the superior value of the confidence interval of a move is situated below the inferior value of the confidence interval of the current best move, the move is pruned. This reduces the response time significantly. However, this technique is not optimal. Figure 1 shows how progressive pruning works while time is running. Another simple strategy to select the first move of a game consists in choosing the move that has the highest confidence interval superior value [34]. This move is the most promising. By updating its mean and its confidence interval, the confidence interval superior value is generally lowered. This move can either be confirmed as the best move or replaced by another promising move. Hence, the best moves are often updated, and moves are not updated as soon as they are estimated as not promising. Moreover, the bad moves are never definitely eliminated from the process.

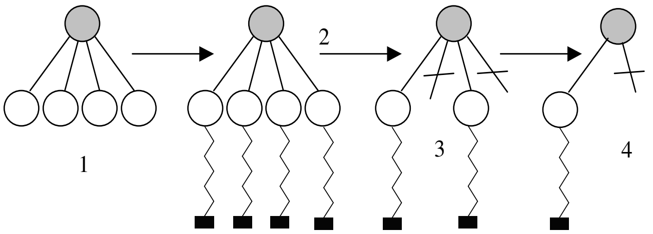

Fig. 1. Progressive pruning: the root is expanded (1). Random games start on children (2). After several random games, some moves are pruned (3). After other random games, one move is left, and the process stops (4).

In 2002, our experiments carried out with Bernard Helmstetter, a doctoral student under Tristan Cazenave's supervision at Paris 8 University, showed that, on 9x9 boards, pure MC programs ranked on a par with heavily knowledge based programs such as Indigo2002 [35]. Given the architectural difference between these programs, that result was amazing. In fact, MC programs share many good properties. The first good property is the increasing playing level in the time used. The more random games, the better the precision on the means. Nowadays, MC method starts to work for a quantitative reason mentioned above. In the near future, with ten times faster computers, the playing strength of MC programs will increase significantly. For knowledge based programs, the knowledge either exists or not whatever the time calculations. For tree search based programs, the timescale is of importance. Considering the ratio by which the speed of computers is multiplied, a ratio of ten only enables tree search programs to look ahead one ply further, which will not improve their playing level significantly in the next few years.

The second good property of MC approach is its robustness of evaluation. Whatever the position, the MC evaluation, far from being totally correct, provides a "good" value. This property is not shared with human-expertise-extracted-knowledge-based programs that can give wrong results on positions where knowledge is erroneous or missing. Furthermore, the variation between the MC evaluation of a position and the MC evaluation of one of the child positions is smooth, which is different in human-expertize-extracted-knowledge-based evaluations.

The third good property of MC Go is its global view. The MC approach does not break down the whole position into sub-positions, which is a risky approach used in classical Go programs. When breaking down a position into sub-positions, the risk is to destroy the problem, and perform local tree searches on irrelevant sub-problems. In such an approach, even if the local tree searches are perfect, the global result is bad as soon as the decomposition is badly performed. MC avoids such risk because it does not break down the position into parts. The move selected by MC is globally good in most cases. Unfortunately, MC programs are tactically bad because they generally perform global tree search at a very shallow depth, even on small boards [36].

Lastly, a MC Go program is easy to develop. This feature may appear unsignificant but it actually brought about the birth of numerous MC programs over the last three years: Vegos [37], DumbGo [38], Crazy Stone [39], Go81 [40], and other programs.

C. Monte-Carlo Go with specific knowledge

In 2003, with both a pure MC program and a knowledge-based program, the association between MC and knowledge provided a tempting perspective. We associated Go knowledge with MC in two different ways: the easy one, and the hard one. The easy one consisted in pre-selecting moves with knowledge, and the hard one consisted in inserting little knowledge into the random games [41]. Indigo2002 was the perfect candidate to become the pre-selector: instead of generating the best move, it was specified to generate the N_{select} best moves, that in turn were input of the MC module as shown in Figure 2.

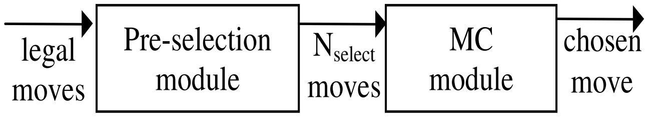

Fig. 2. The two modules of Indigo2003: the pre-selection module selects N_{select} moves by the mean of lot of knowledge, and local tree searches, additionnally yielding a conceptual evaluation of the position. Then, among the N_{select} moves, the MC module selects the move to play by the mean of random simulations.

This simple addition shortened the response time and enabled a MC program to play on 19x19. Moreover, the move pre-selector performing local tree searches could prune tactically bad moves.

The second way to associate specific knowledge and MC is, by far, much more interesting because it introduces the RL experiments described in this paper. Instead of using the uniform probability, it consists in using a non-uniform probability for (pseudo-)random game move generation. This approach results from the use of domain-dependent knowledge. At this point, a few words have to be defined. While the term pseudo-random refers to numbers actually generated by computers, and while the term random refers to the mathematical property of random variables, we use these two terms, pseudo-random and random, in a slightly different meaning: we call random the moves, or the numbers, generated by the rand() function of the computer (intended to be generated with a probability as uniform as possible), and we call pseudo-random, the moves generated by our domain-dependent approach which uses a non-uniform probability.

The MC idea lies in performing a huge number of times a simple random episode to deduce a complex behaviour. In pure MC, the episode was a move sequence respecting the rules of the game, and the complex behaviour, to some extent, was a program playing on a par with Indigo2002. What is the complex behaviour brought about by the episode composed by a sequence of moves respecting the rules and following some basic Go principles such as string capture-escape and cut-connect ?

Concerning the string capture-escape concept, the urgency of the move filling the last liberty of the one-liberty string is linear in the string size. Concerning the cut-connect concept, a pattern representation is adapted. In practice, the question is to determine the adequate pattern size: large enough to contain most concept instances, and small enough not to slow down the random games. The cut-connect concept is not well described by 2x2 patterns nor by the cross patterns (one intersection plus its four neighbours), but it is described quite well by 3x3 patterns (one intersection plus its 8 neighbours). Larger patterns would give better results, but, concerning the cut-connect concept, the most urgent patterns are the smallest ones. Therefore, 3x3 is the proper size to enclose the cut-connect concept. A 3x3 pattern has an empty intersection in its center, and the 8 neighbouring intersections are arbitrary. The urgency of a pattern corresponds to the urgency of playing in its center when this pattern matches the position.

To decide stochastically which move to play during a random game, each matched pattern and each one-liberty string bring their urgency to a given intersection. For each intersection, the urgency to play on it amounts to the sum of the urgencies brought by patterns and strings. Then, the probability of playing on a given intersection is linear in its urgency. From now on, the episodes look like Go games, and they keep their exploratory property. With a probability based on domain-dependent knowledge, the means obtained are more significant than the means using uniform probability. We are now able to provide the features of a Pseudo-Random (PR) player :

- 3x3 pattern urgency table
- 3^8 3x3 pattern (center is empty)
- 25 dispositions to the edge
- #patterns = 250,000
- one-liberty urgency

In the following, we call $Zero$ the PR player that uses a uniform probability. $Zero$ has its urgencies set to zero. It corresponds to the pure MC Go approach. We call $Manual$ the PR program based on domain-dependent concepts that was built in 2003 by a translation of a small 3x3 pattern database manually filled by a Go expert. We call $MC(p)$ the MC program that uses the architecture of Figure 2, and that uses the PR program p in order to carry out its simulations. In 2003, we made the match between $MC(Manual)$ and $MC(Zero)$ on 9x9, 13x13 and 19x19 boards [41]. Table II gives the results.

board size	9x9	13x13	19x19
mean	+8	+40	+100
% wins	68%	93%	97%

TABLE II
RESULTS OF $MC(Manual)$ VS $MC(Zero)$ FOR THE USUAL BOARD SIZES.

The results clearly show that using a domain-dependent probability is superior to using a uniform probability. The larger the board, the clearer the result. On 19x19 boards, the difference equals 100 points on average, which is huge by Go standards. At this stage, it is normal to look for automatic methods and see whether they can do better than $MC(Manual)$. This leads us to the core elements of this paper: how to use RL in an MC Go architecture.

IV. REINFORCEMENT LEARNING AND MONTE-CARLO GO

The general goal is to automatically build a PR player p for $MC(p)$ as strong as possible. In this paper we explore the use of RL deeply influenced by Richard Sutton's work. Sutton is the author of Temporal Difference (TD) method [3], and with Barto co-author of a book describing the state of the art [1] (also described by [2]). RL is also known for the success of Q-learning [42]. RL often uses the Markov Decision Process (MDP) formalism: an agent evolves in a non-deterministic environment. He performs actions according to his own policy. His actions make him change from state to state, and result in returns. The aim of the agent is to maximize his cumulated return in the long term. To this purpose, every state has a value determined by the state value function V, and each action associated to a state has an action value determined by the action value function Q. The learning agent either updates action values and state values according to his policy, or greedily improves his policy depending on action values and/or state values. RL inherits from Dynamic Programming (DP) [43] the updating rule for state values and action values. But RL is different from DP because sweeping of the state space is replaced by the experience of the agent. In our work, if RL did not provide better results than $MC(Manual)$, we would plan to use Evolutionary Computation (EC) principles [44] in a following stage.

Before the RL experiments, the PR player is $Manual$. It uses 3x3 patterns manually built by an expert and by means of an automatic translation from a database to a table. The expert was not be able to build a larger database easily containing larger patterns and adequate urgencies. If we wish to enlarge this knowledge, we must use an automatic method. The playing level of $MC(Manual)$ is quite good, and it is not easy to find p such as $MC(p)$ be better than $MC(Manual)$. But if we succeed with 3x3 patterns, we will be certain that the automatic method produces better results that the manual method on larger patterns, even if the expert manually tunes the large database.

Subsequently, we can say that p_1 is better than p_2 at the low level, or random level, when p_1 beats p_2 by a positive score on average after a sufficient number of games. We can say that p_1 is better than p_2 at the high level, or MC level, when $MC(p_1)$ beat $MC(p_2)$ by a positive score on average after a sufficient number of games. We aim at seeing the PR players improving at the MC level, and not necessarily at the low level. Improving a PR player at the low level can be a red herring. For instance, a PR player p that is quite good (because he beats $Zero$ at the low level by a given score) can be improved at the low level only by making him less exploratory. This determinisation results in a better score for the PR player p against $Zero$ but, his exploratory capacity being low, $MC(p)$ may be weak, and even be beaten by $MC(Zero)$. When considering the balance between exploration and exploitation [1], we may draw Figure 3 showing the programs on a randomness dimension. On the left, there are deterministic and greedy programs, then, on their right, ϵ-greedy programs that play randomly in an ϵ proportion, and that play deterministically in a $1-\epsilon$ proportion. On the right of Figure 3, there is $Zero$, the random program based on the uniform probability, and on its left the PR programs used in our MC architecture. Those programs are constrained to keep their exploratory capacity and to stay on the right of the figure.

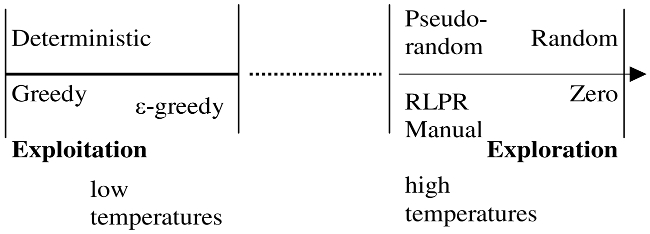

Fig. 3. The randomness dimension: the completely deterministic programs are situated on the left. $Zero$ is situated on the right. On the left of $Zero$, there are the PR programs and $Manual$. On the right of deterministic programs, there are ϵ-greedy programs [1]. The temperature indicates a randomisation degree: 0 for deterministic programs, and infinite for $Zero$, the uniform probability player.

In the following, we call $RLPR$, a PR player whose table is built with RL techniques. We may perform experiments at the low level, or at the MC level. The upside of the low level is the high speed of games. Its downside is to favour exploitation against exploration. Despite of its slowness, MC level remains in keeping with our aim..

V. Experiments

This section describes two experiments: one experiment (experiment 1a) performed at the low level, with one program. This experiment uncovers the obstacle of determinisation during learning. Experiment 1b attempts to solve this obstacle by replacing the sole program by a population of programs. Experiment 2 is performed at the MC level with one single player, and explicitly manages the obstacle of determinisation.

A. Experiment 1: low level, one program or a population of programs

This subsection describes an experiment made at the low level, with one program in self-play or with a population of programs. The result of a game is either its outcome ($+1$ for a win and -1 for loss) or a score. This subsection assumes that the result used is the outcome. A pattern has an associated action: playing the advised move when the pattern is matching. A pattern has an action value Q that is the mean of the games' results when the pattern has been matched and played. Q belongs to $]-1,+1[$. In our experiment, a pattern has an urgency U linked to Q by:

$U = \left(\frac{1+Q}{1-Q}\right)^k$

During a pseudo-random game, the probability of playing a move advised by a pattern is linear in U. k is a parameter corresponding to the determinisation degree of the program. When $k = 0$, then $U = 1$ for every patterns, and the probability of playing a move is uniform. When k is high, the highest urgency dominates all the other urgencies and the program is almost deterministic. The n^{th} update of Q for a pattern is given by:

$Q_n = Q_{n-1} + \alpha(R - Q_{n-1})$

R is the result of the random game, and $\alpha = 1/(1+n)$. Thus, Q_n converges to the mean value of the results of the random games.

More precisely, two tables are used: one for playing, one for learning. This is an off-line learning. After a block of games, the values of the learnt table Q_{learn} update the values of the table used for playing Q_{play} by:

$Q_{play} = Q_{play} + \lambda^b Q_{learn}$

λ is a parameter set in $]0,1[$. Its actual value is set by the experiments. b is the number of the block. In the updating formula, the addition is used to keep track of good patterns. During the first block of games, all Q_{play} values equal zero, and the games follow a uniform probability. At the end of the first block of games, a good pattern has a high Q_{learn} value because it generates good moves among a set of games played badly. This value corresponds to the mean value of results of games given that the policy follows the uniform probability. Q_{learn} is copied into Q_{play} to be used for playing in the next block of games. A good pattern quickly increases its Q_{play} value. At the end of a block of games, Q_{learn} corresponds to the mean value of results of games given that the policy uses the Q_{play} table. Because λ is strictly inferior to 1, Q_{play} converges to a limit when b increases.

1) Experiment 1a: one unique learning program: This first experiment contains results on 9x9 boards only :

- $RLPR >> Zero$
- $RLPR < Manual$
- $MC(RLPR) << MC(Manual)$

$RLPR >> Zero$ shows a learning at the low level. This is the minimal result expected. However, $RLPR < Manual$ shows that learning is not completely satisfactory. $MC(RLPR) << MC(Manual)$ lets us think that $RLPR$ is too deterministic. As soon as the learner has learnt Q values for relevant patterns, instead of learning new Q values for new patterns, the learner prefers to increase the existing Q values. This results in a player becoming too deterministic to be used as a basis of the MC player. We call this phenomenon determinisation. Experiment 2 will show a different update rule that avoids determinisation in self-play. However, experiment 1b will use the same update rule as experiment 1a but it will prevent determinisation by using a population of learners.

We may comment upon the off-line learning used in this experiment. λ is strictly inferior to 1 to garantee convergence of Q_{play}. However, in practice, we set $\lambda = 1$ because we observed that, for good patterns, Q_{learn} converges to 0. Furthermore, we observed that, even though $\lambda = 1$, Q_{play} practically stays in $]-1, +1[$. We do not have theoretical proof of this phenomenon, but we may provide an intuitive explanation: when b is sufficiently high, at the end of a block of games, Q_{learn} corresponds to the mean value of results of games given that the policy is good as well. Thus, when a good pattern is chosen during a game using a good policy, this is not a surprise, and the mean value of results of games given that this good pattern is chosen, roughly equals zero. Finally, with this comment, we observe that what happens in the first block of random games is crucial to the actual final value of Q_{play}. Launching several executions of the process leads to players that roughly share the same playing level but may have quite different tables. Using a population of learners intends to lower the importance of the first block of games.

2) Experiment 1b: a population of learning programs: To avoid determinisation of a program, and inspired by the rule: “when RL does not work, try EC principles”, we performed an experiment similar to experiment 1a by replacing one $RLPR$ program by a population of $RLPR$ programs. The size of the population is $N = 64$. The underlying idea is that each individual program learns in its own manner (increases Q values of specific patterns only). If a program learns by determinisation, he cannot survive the next generation against other programs having learnt differently. A generation includes three phases: reinforcement learning, test and selection. During the reinforcement learning phase, the $RLPR$ programs play against each other while learning with the update rule of experiment 1a. Then, for each learner, the learnt table is added into the playing table. During the test phase, the $RLPR$ programs play against fixed opponents ($Zero$ and $Manual$) without learning. This phase yields a ranking. The selection phase follows the code below:

```
Delete the N/2 worst RLPR players
For (D=N/4; D>0; D=D/2)
  copy the best D RLPR players
Add Zero player
```

(The best $RLPR$ program of the generation is copied five times). This experiment does not use other classical EC concepts: mutation or cross-over. We obtained results on 19x19:

- Starting population = $Zero$
  - $RLPR = Zero + 180$
  - $RLPR = Manual - 50$
  - $MC(RLPR) << MC(Manual)$
- Starting population = $Manual$
  - $RLPR = Zero + 180$
  - $RLPR = Manual + 50$
  - $MC(RLPR) = MC(Manual) - 20$

With a population of programs, learning is possible on 19x19, which was not possible with one unique program. In the whole set of programs, some of them learn without determinisation, which is right. The convergence depends on the starting program. When starting with $Zero$, the population goes toward a local maximum ($RLPR = Manual - 50$) inferior to the maximum reached when starting from $Manual$ ($RLPR = Manual + 50$). Besides, $MC(RLPR) = MC(Manual) - 20$ is a better result than $MC(RLPR) << MC(Manual)$ obtained in the previous experiment. In this perspective, this result is good (20 points can be considered as a reasonable difference on 19x19). However, the two results $MC(RLPR) = MC(Manual) - 20$ and $RLPR = Manual + 50$ underlines that learning still corresponds to determinisation.

Our conclusion on experiment 1 is that, at the low level, the $RLPR$ programs have a tendency to determinisation that hides true learning. Replacing one program by a population lowers the determinisation problem without removing it completely. Therefore, in the following experiment, we leave the low level to perform games at the MC level, even if this costs computing time.

B. Experiment 2: relative difference at MC level

In this experiment, a $MC(RLPR)$ player plays against itself again. There is no population. We need a mechanism that prevents determinisation of experiment 1a. Therefore, the update rule of experiment 2 is different from the update rule of experiment 1a. Instead of updating pattern urgencies one by one, our idea is to consider pairs of patterns, and a relative differences between variables associated to the pairs of patterns. Thus, the player uses a relative difference formula to learn. a and b being two patterns with two MC evaluations, V_a and V_b, and two urgencies, u_a and u_b, on average we aim at:

$$\exp(C(V_a - V_b)) = u_a/u_b$$

This is the basic formula underlying this experiment. It establishes a correlation between a difference of evaluations on average, and a ratio between the two urgencies that we seek. This way, the over-determinisation of pattern urgencies should not occur. For pattern i, we define Q_i :

$Q_i = \log(u_i)$

Thus, for two patterns a and b, we look for:

$Q_a - Q_b = C(V_a - V_b)$

C is assumed to be constant. On a given position with a and b matching, the observed relative difference is actually:

$delta = Q_a - Q_b - C(V_a - V_b)$

The updating rules are:

$Q_a = Q_a - \alpha delta$

$Q_b = Q_b + \alpha delta$

When comparing two patterns, a et b, these rules update the ratio u_a/u_b according to $delta$ avoiding exagerated determinisation. We performed learning on 9x9. We have used a small number of random games to compute V_a and V_b: 20 random games only. $C = 0.7, 0.8, 0.9, 1.0$ were rather good values. If N_i is the number of times that pattern i matches, we set α proportional to the inverse of $\sqrt{N_i}$. We have tested our 9x9 learner on 9x9 and 19x19 boards.

- on 9x9:
  - $MC(RLPR) = MC(Manual) + 3$
- on 19x19:
  - $MC(RLPR) = MC(Manual) - 30$

An investigation on aspects in which $MC(RLPR)$ plays different, better or worse than $MC(Manual)$ player can be performed along the way of playing or along the achieved result. Concerning the achieved result, and assessing on 19x19 boards a $MC(RLPR)$ player that learnt on 9x9 boards, the achieved result (-30) is similar to the result of experiment 1b (-20). In other terms, the results obtained on 19x19 are promising. On 19x19, the results could have been better if we performed learning on 19x19 as well, but we did not have enough time to do it. Additionally, $MC(RLPR) = MC(Manual) + 3$ shows that the method works better on 9x9 at the MC level than the manual method (this result was what we aimed at). The determinisation problem seems to be solved partially. The way we used relative difference looks like advantage updating [45]. We may hardly investigate on the way of playing, and on the style of $MC(RLPR)$ against $MC(Manual)$, because both programs share the same design, and their playing style is almost identical. However, we may give some remarks concerning the inside of the urgency tables. Because the patterns used by $Manual$ were created by a human expert, the patterns always correspond to go concepts such as cut and connect. Thus, the urgency table of $Manual$ contains non-zero-and-very-high values very sparsely, and the intersection urgency computing process is optimized to this respect. A drawback of $RLPR$ players, is that the urgency table is almost completely filled with non-zero values with a smooth continuum of values. The intersection urgency computing process during random games cannot be optimized in this respect, which slows down $RLPR$ players. Thus, to be efficiently used, the tables of $RLPR$ players should be adequately post-processed after learning.

VI. Conclusion

This paper has presented the Monte-Carlo Go architecture using domain-dependent knowledge, and has described RL experiments to enhance 3x3 pattern urgencies used during simulations. In experiment 1a, we identified the determinisation obstacle that negated a good learning. Experiment 1b, a copy of experiment 1a at the low level and replacing one RL learner by a population of RL learners, avoided determinisation. Experiment 2 using relative difference and using Q values instead of raw urgencies, explicitly managed the determinisation. Consequently, experiment 2 worked well at the MC level with one learner only, instead of a population of learners. Quantitatively, the results obtained by experiment 1b and 2 are very promising: after learning on 9x9, the automatic method is 3 points better than the manual method. On 19x19, the automatic method is (only) 20 points below the manual method. But in experiment 2, learning was performed on 9x9 and tested on 19x19. Thus, the perspective is to perform learning of experiment 2 on 19x19 and test on 19x19. Nevertheless, the results of the automatic method must be reinforced to be certain that the automatic method is really better than the manual one for 3x3 patterns. With such certainty, we may replace 3x3 patterns by larger patterns that a Go expert would have too difficulties to qualify with adequate urgencies, whereas the automatic method would easily tackle them.

Discussing ideas linked to EC might be enlightening. Experiments has been carried out on Go with EC [46]. The size of the board, although small in these experiments played a key role: a preliminary learning on a small board speeds up the following learning performed on a larger board. In our work, learning urgencies of 3x3 patterns on 9x9 boards yields a playing level well-tuned for 9x9 boards, but less adapted to 19x19 boards. To play well on 19x19 boards, learning on 19x19 boards is advisable. However, it is possible to play or learn on 19x19 boards with a player that learnt on 9x9 boards.

Besides, in experiment 1b, we observed that the result depended on the initial conditions, and the optimum reached was only local. This experimental result confirmed the theoretical result known on partially observable MDP [47].

Within the current debate between RL and EC, RL alone seems to be able to tackle our problem almost entirely (experiment 2). But, instead of using one unique RL learner, using a population of learners and a selection mechanism without mutation or cross-over (experiment 1b) unwound the situation (experiment 1a). In this view, experiment 1 demonstrates the success of the cooperation of principles borrowed from both sides, RL and EC. The training method can be viewed as a memetic algorithm in which randomness replaces the role of genetic variation. Furthermore, this conclusion enriches previous results concerning the RL-vs-EC debate using Go as a testbed [48].

Lastly, if we have a closer look at the results on 19x19 boards, how to account for the slightly worse results obtained by the automatic method compared to the manual method ?

The MC environment may be too exploratory, and the determinisation is actually too tempting and easy a solution for RL learners whose goal is to learn by winning. Giving up the MC environment for a while, performing classical Q-learning experiments [42], [49] on ϵ-greedy programs might constitute the first steps to the solution: the ϵ-greedy programs being almost deterministic (see Figure 3), determinisation might be minimized. Then, randomizing such programs, and testing them within the MC environment would be the final steps.

VII. Aknowledgements

This work was started in 2004 by Bruno Bouzy, and was continued during spring 2005 by Guillaume Chaslot for his last year placement at Ecole Centrale de Lille, in cooperation with Rémi Coulom we warmly thank for the discussions and the clever ideas he suggested.

References

[1] R. Sutton and A. Barto, *Reinforcement Learning: an introduction*, T. Dietterich, Ed. MIT Press, 1998.

[2] L. P. Kaelbling, M. Littman, and A. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996. [Online]. Available: citeseer.ist.psu.edu/kaelbling96reinforcement.html

[3] R. Sutton, "Learning to predict by the method of temporal differences," *Machine Learning*, vol. 3, pp. 9–44, 1988.

[4] C. Watkins, "Learning from delayed rewards," Ph.D. dissertation, Cambridge University, 1989.

[5] B. Bouzy, "Indigo home page," www.math-info.univ-paris5.fr/∼bouzy/INDIGO.html, 2005.

[6] J. Schaeffer and J. van den Herik, "Games, Computers, and Artificial Intelligence," *Artificial Intelligence*, vol. 134, pp. 1–7, 2002.

[7] H. van den Herik, J. Uiterwijk, and J. van Rijswijck, "Games solved: Now and in the future," *Artificial Intelligence*, vol. 134, pp. 277–311, 2002.

[8] J. Schaeffer, *One Jump Ahead: Clallenging Human Supremacy in Checkers*. Springer-Verlag, 1997.

[9] M. Campbell, A. Hoane, and F.-H. Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, pp. 57–83, 2002.

[10] J. Schaeffer, Y. Björnsson, N. Burch, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen, "Solving checkers," in *IJCAI*, 2005, pp. 292–297.

[11] M. Buro, "Improving heuristic mini-max search by supervised learning," *Artificial Intelligence Journal*, vol. 134, pp. 85–99, 2002.

[12] M. Müller, "Position evaluation in computer go," *ICGA Journal*, vol. 25, no. 4, pp. 219–228, December 2002.

[13] ——, "Computer go," *Artificial Intelligence*, vol. 134, pp. 145–179, 2002.

[14] B. Bouzy and T. Cazenave, "Computer go: an AI oriented survey," *Artificial Intelligence*, vol. 132, pp. 39–103, 2001.

[15] D. Fotland, "The many faces of go," www.smart-games.com/manyfaces.html.

[16] M. Reiss, "Go++," www.goplusplus.com/.

[17] D. Bump, "Gnugo home page," www.gnu.org/software/gnugo/devel.html, 2006.

[18] M. Enzenberger, "Neurogo," www.markus-enzenberger.de/neurogo.html.

[19] M. Müller, "Explorer," web.cs.ualberta.ca/∼mmueller/cgo/explorer.html, 2005.

[20] T. Cazenave, "Golois," www.ai.univ-paris8.fr/∼cazenave/Golois.html.

[21] K. Chen, "Some practical techniques for global search in go," *ICGA Journal*, vol. 23, no. 2, pp. 67–74, 2000.

[22] M. Enzenberger, "Evaluation in go by a neural network using soft segmentation," in *10th Advances in Computer Games*, E. A. H. H. Jaap van den Herik, Hiroyuki Iida, Ed. Graz: Kluwer Academic Publishers, 2003, pp. 97–108.

[23] M. Müller, "Decomposition search: A combinatorial games approach to game tree search, with applications to solving go endgame," in *IJCAI*, 1999, pp. 578–583.

[24] T. Cazenave, "Abstract proof search," in *Computers and Games*, ser. Lecture Notes in Computer Science, I. F. T. Marsland, Ed., no. 2063. Springer, 2000, pp. 39–54.

[25] E. van der Werf, J. Uiterwijk, and J. van den Herik, "Learning to score final positions in the game of go," in *Advances in Computer Games, Many Games, Many Challenges*, H. J. van den Herik, H. Iida, and E. A. Heinz, Eds., vol. 10. Kluwer Academic Publishers, 2003, pp. 143–158.

[26] Fishman, *Monte-Carlo : Concepts, Algorithms, Applications*. Springer, 1996.

[27] G. Tesauro and G. Galperin, "On-line policy improvement using Monte Carlo search," in *Advances in Neural Information Processing Systems*. Cambridge MA: MIT Press, 1996, pp. 1068–1074.

[28] D. Billings, A. Davidson, J. Schaeffer, and D. Szafron, "The challenge of poker," *Artificial Intelligence*, vol. 134, pp. 201–240, 2002.

[29] B. Sheppard, "World-championship-caliber scrabble," *Artificial Intelligence*, vol. 134, pp. 241–275, 2002.

[30] B. Abramson, "Expected-outcome : a general model of static evaluation," *IEEE Transactions on PAMI*, vol. 12, pp. 182–193, 1990.

[31] B. Brügmann, "Monte Carlo go," 1993, www.joy.ne.jp/welcome/igs/Go/computer/mcgo.tex.Z.

[32] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by simulated annealing," *Science*, May 1983.

[33] B. Bouzy and B. Helmstetter, "Monte Carlo go developments," in *10th Advances in Computer Games*, E. A. H. H. Jaap van den Herik, Hiroyuki Iida, Ed. Graz: Kluwer Academic Publishers, 2003, pp. 159–174.

[34] L. P. Kaelbling, "Learning in embedded systems," Ph.D. dissertation, MIT, 1993.

[35] B. Bouzy, "The move decision process of Indigo," *International Computer Game Association Journal*, vol. 26, no. 1, pp. 14–27, March 2003.

[36] ——, "Associating shallow and selective global tree search with Monte Carlo for 9x9 go," in *Computers and Games: 4th International Conference, CG 2004*, ser. Lecture Notes in Computer Science, N. N. J. van den Herik, Y. Björnsson, Ed., vol. 3846 / 2006. Ramat-Gan, Israel: Springer Verlag, July 2004, pp. 67–80.

[37] P. Kaminski, "Vegos home page," www.ideanest.com/vegos/, 2003.

[38] J. Hamlen, "Seven year itch," *ICGA Journal*, vol. 27, no. 4, pp. 255–258, 2004.

[39] R. Coulom, "Efficient selectivity and back-up operators in monte-carlo tree search," in *Computers and Games*, Torino, Italy, 2006, paper currently submitted.

[40] T. Raiko, "The go-playing program called go81," in *Finnish Artificial Intelligence Conference*, Helsinki, Finland, September 2004, pp. 197–206.

[41] B. Bouzy, "Associating knowledge and Monte Carlo approaches within a go program," *Information Sciences*, vol. 175, no. 4, pp. 247–257, November 2005.

[42] C. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, 1992.

[43] D. P. Bertsekas, *Dynamic Programming and Optimal Control*. Athena Scientific, 1995.

[44] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Co, 1989.

[45] L. Baird, "Advantage updating," 1993. [Online]. Available: citeseer.ist.psu.edu/baird93advantage.html

[46] K. Stanley and R. Miikkulainen, "Evolving a roving eye for go," in *Genetic and Evolutionary Computation Conference*, New-York, 2004.

[47] T. Jaakkola, S. P. Singh, and M. I. Jordan, "Reinforcement learning algorithm for partially observable Markov decision problems," in *Advances in Neural Information Processing Systems*, G. Tesauro, D. Touretzky, and T. Leen, Eds., vol. 7. The MIT Press, 1995, pp. 345–352. [Online]. Available: citeseer.ist.psu.edu/jaakkola95reinforcement.html

[48] T. P. Runarsson and S. Lucas, "Co-evolution versus self-play temporal difference learning for acquiring position evaluation in small-board go," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 6, pp. 628–640, December 2005.

[49] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *Proceedings of the 11th International Conference on Machine Learning (ML-94)*. New Brunswick, NJ: Morgan Kaufmann, 1994, pp. 157–163. [Online]. Available: citeseer.ist.psu.edu/littman94markov.html