HAL
archives-ouvertes.fr

Knowledge complement for Monte Carlo Tree Search: an application to combinatorial games

André Fabbri, Frédéric Armetta, Eric Duchêne, Salima Hassas

▶ **To cite this version:**

André Fabbri, Frédéric Armetta, Eric Duchêne, Salima Hassas. Knowledge complement for Monte Carlo Tree Search: an application to combinatorial games. 2014 IEEE 26th International Conference on Tools with Artificial Intelligence, Nov 2014, Limassol, Cyprus. pp.997-1003, 10.1109/ICTAI.2014.151 . hal-01083449

HAL Id: hal-01083449

https://hal.archives-ouvertes.fr/hal-01083449

Submitted on 10 Dec 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

©2014 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1109/ICTAI.2014.151

Knowledge complement for Monte Carlo Tree Search: an application to combinatorial games

André Fabbri, Frédéric Armetta, Éric Duchêne and Salima Hassas
Université de Lyon, CNRS
Université Lyon 1, LIRIS, UMR5205, F-69622
Lyon, France
email: {name}.{surname}@liris.cnrs.fr

***Abstract*—MCTS (Monte Carlo Tree Search) is a well-known and efficient process to cover and evaluate a large range of states for combinatorial problems. We choose to study MCTS for the Computer Go problem, which is one of the most challenging problem in the field in Artificial Intelligence. For this game, a single combinatorial approach does not always lead to a reliable evaluation of the game states. In order to enhance MCTS ability to tackle such problems, one can benefit from game specific knowledge in order to increase the accuracy of the game state evaluation. Such a knowledge is not easy to acquire. It is the result of a constructivist learning mechanism based on the experience of the player. That is why we explore the idea to endow the MCTS with a process inspired by constructivist learning, to self-acquire knowledge from playing experience. In this paper, we propose a complementary process for MCTS called BHRF (Background History Reply Forest), which allows to memorize efficient patterns in order to promote their use through the MCTS process. Our experimental results lead to promising results and underline how self-acquired data can be useful for MCTS based algorithms.**

***Index Terms*—Monte Carlo Tree Search; Computer-Game; Reinforcement Learning; Knowledge Engineering**

I. INTRODUCTION

In this paper, we propose a self-acquiring knowledge process to deal with the resolution of hard combinatorial problems. The generic MCTS enhancement is applied to a difficult combinatorial game: the game of Go. We observe that MCTS is a very efficient process to cover a huge set of states. Nevertheless it does not take full advantage of its experience. This statement motivates us to explore a new approach to increase the ability for such a process to capitalize its experience, which lead us to consider the formed system as a cognitive system.

The game of Go is a good testbed for Artificial Intelligence [1]. The rules are simple but capturing the underlying explanations for an efficient sequence of moves remains an open problem. The human players acquire an advanced representation of the game by an extensive practice. This explains why the best players still defeat computer programs. Indeed a better representation allows to focus on the most relevant parts of the game, unlike to the default tree search which considers all the possible evolutions of the game. In order to cope with the combinatorial hardness of the problem, a recurrent approach has been to endow the programs with a large amount of encoded expert knowledge (rules, patterns, etc.).

MCTS led to a major breakthrough for the game of Go [2] and is now applied to a wide set of problems [3]. Contrary to the former approaches, the evaluations of possible evolutions are learnt on-line, through random simulations. The program acquires hence some knowledge about the current situation by a self-play simulated experience. Nevertheless, MCTS does not suffice to overcome the combinatorial complexity of the game of Go yet. The performance of the programs stagnate for an increasing number of simulations, even combined with expert knowledge [4]. In our understanding, past a certain threshold, the pure computational approach cannot be a substitute for a better cognitive integration of the experience.

A promising way to increase the efficiency of a program would be to enhance its ability to accumulate knowledge about its simulated experience. The general idea consists in a better assimilation of the inherent knowledge associated with the states covered by MCTS. This approach has been partially considered in the literature but we claim and argue that this kind of process can be improved in many ways. With our approach BHRF (Background History Reply Forest), we choose to endow the program with the ability to memorize patterns learnt on-line and adapt their estimated value during the game. These patterns will influence back the simulations in order to enrich the simulated experience. This paper give insights about the potential of such an approach. Note that our results mainly focus on the quality of the learning rather than the effective performance in a competitive setting.

More details about BHRF will be provided in Section III. The MCTS baseline and the main knowledge endowment will be presented in Section II. Experimental results are given and analyzed in Section IV. A conclusion and some perspectives are drawn in Section V.

II. HOW TO COMPLEMENT MCTS ?

MCTS progressively weights by self-play several possible evolutions of the game. However, additional knowledge can substantially enhance the learning process. A brief presentation of the MCTS process along with its dynamic is presented in Section II-A. Section II-B reviewed the main enhancement in the current programs based on MCTS and Section II-C details the underlying data structure.

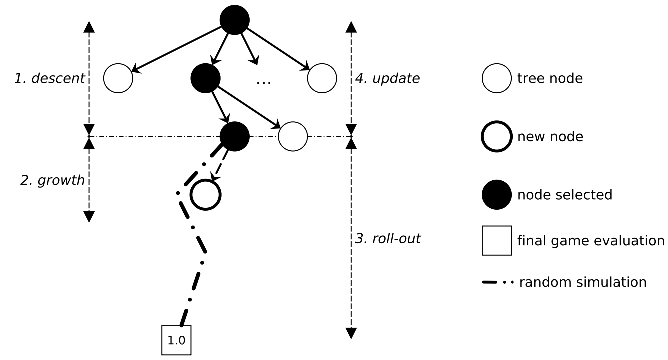


Figure 1: Monte Carlo Tree Search process

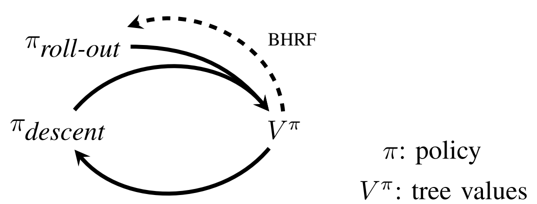


Figure 2: Generalized policy iteration process for MCTS

An extensive presentation of MCTS and its enhancements is beyond the scope of this paper, we invite the reader to refer to [3] for more details.

A. Monte Carlo Tree Search

The standard MCTS algorithm gradually expands a search tree starting from the current state. The four steps *descent*, *growth*, *roll-out* and *update* (see Figure 1) are iteratively applied until we meet some restraining constraints (time, memory or iteration number). The *descent* policy covers the tree and selects a new node to sample. The *growth* phase adds a node to the tree search. From this node, the *roll-out* policy generates the remaining moves until the simulation reaches a final state. The *update* phase finally propagates back the outcome in the tree search.

The value learnt in the search tree are tightened with the underlying policies. On the one hand, the *descent* policy considers the node's values to reach the most promising node to deepen. On the other hand, the outcome of the simulated end game adjusts the values of the node selected during the *descent* and influence back the next *descent* policy. The interaction between the policy and the learnt values refers to the *generalized policy iteration* process [5].

In MCTS, the policy iteration involves also a *roll-out* policy. This policy generates the last moves leading to the final state and therefore contributes to the learning process. However the *roll-out* policy does not benefit from the learnt weights. The purpose of the presented method is to influence back the *roll-out* policy as presented in Figure 2.

B. Enhanced policy iteration process

As pointed out by the generalized policy iteration, the policies play a major role in the learning. The *descent* and *roll-out* policies have been progressively enhanced to cope with the issues addressed by each phase. In this section, the main enhancements for each policy are reviewed from a Go-specific and a more general perspective.

Over the iterations, the node's weights are progressively refined and the *descent* policy has to focus quickly on the most promising parts of the tree. For the game of Go, expert off-line knowledge may efficiently promote states consequent to interesting moves and avoid silly ones. This knowledge may enhance the search by biasing the values or pruning the tree [6]. From a broader perspective, the Upper Confidence bound applied to Tree [7] considers the number of updates to achieve a good balance between the exploration of current sub-optimal states and the exploitation of the current best states.

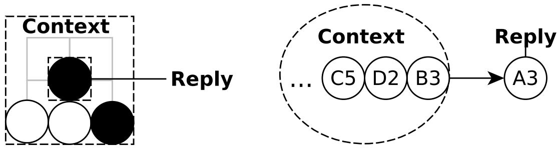
A pure random *roll-out* policy generates many non-representative final states whose outcome slows the learning of the system. Thus, the *roll-out* policies generally involve additional knowledge to enhance the relevance of the final states. For the game of Go, the sequence-like policies successfully consider expert or off-line knowledge to guide the simulations [8]. However such a *roll-out* policy is difficult to improve because it has to balance carefully the distribution of the final states to cover [9]. A promising way consists in designing adaptive (rather than static) *roll-out* policies.

General-game approaches such as N-Grams [10] propose more adaptive kind of knowledge. They enhance the *roll-out* policy with move sequences evaluated on-line. However the move sequences considered come from the *roll-out* itself rather from the search tree. To the best of our knowledge, the Pool-RAVE enhancement [11] is the only attempt to exploit knowledge coming from the tree but considers single moves rather than sequences. A pool of potential best moves are picked up during the *descent* and re-exploited in the *roll-out*. This method achieves good results for the *roll-out* policies without expert knowledge but does not intend to learn explicit knowledge from the tree. Such a learning requires the adequate underlying data-structure as presented in Section II-C.

The search tree actually stores the outcomes of the simulations. Following this perspective, MCTS becomes then a cognitive problem: how to capitalize the simulated experience of the system ? This is a long-term issue and, in our approach, we will focus on the memorization of raw moves sequences coming from the tree.

C. Knowledge data structure

The policies select the action to perform based on the knowledge available for the system. The data-structure supporting this knowledge has an high influence on how this knowledge may be re-exploited. Current programs based on MCTS handle different kinds of knowledge. In the present paper, we differentiate the knowledge learnt in the search tree from the additional knowledge considered in the enhanced policies. The latter shall be applied to different situations



(a) Spatial pattern (b) Temporal pattern

Figure 3: Additional knowledge for MCTS

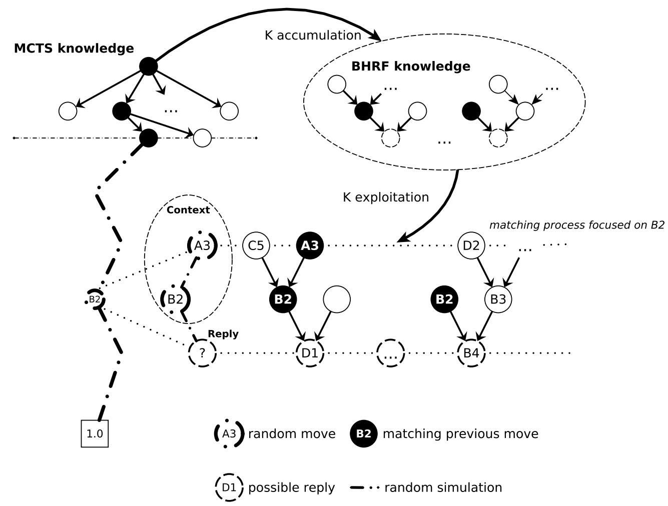


Figure 4: Background History Reply Forest process

contrary to the node's knowledge which is specific to a single game state. In this section, the main knowledge data-structures are reviewed for both kind of knowledge.

In the best Go-program, the expert knowledge involved in *descent* and *roll-out* policies considers immediate reply to small spatial context. For each pattern, the surrounding positions stand for the context and the middle move corresponds to an appropriated reply (Figure 3a). This knowledge successfully simulate local fights in sequence-like policies but is insufficiently expressive to evaluate states with multiple independent sub-problems. Indeed spatial patterns alone are not able to express the diversity of a sequential decision process [12].

The general-game approach considers immediate reply to small temporal context (or pattern). For each pattern, the first moves stand for the context and the last move stands for an appropriated reply (Figure 3b). This context is generally small. One of the most relevant approach is detailed in [13] and considers up to two moves for the context. However this short term perimeter for the contexts may raise a short-sighted phenomenon, i.e., the so formed sequences can be applied to many different states but are not relevant for each of them. Previous attempts such as the move answer tree in [14] or the local tree in [15] propose to specialize the temporal pattern but does not provide effective results yet.

The values learnt in the search tree corresponds to the estimated win probability of the specific game states covered during the *descent* phase. This knowledge is not prone to be re-exploited in similar states (except for the very same game state see [16]). As pointed out in [17], if a branch of the tree learns a sequence of actions that solves a local sub-problem, this sequence has to be rediscovered in the other branches where this sub-problem occurs. Moreover each time the opponent has played his move, one has to prune the tree to keep only the subtree associated with the moves that are effectively played. As a result, the knowledge accumulated in the other branches is also lost.

The purpose of our approach is to design temporal pattern in order to extract the knowledge inside the search tree. We choose to extend the size of the patterns so that they can specialize to more specific contexts, as for the search tree.

III. A TRANSVERSAL DATA STRUCTURE TO COMPLETE MCTS

Our proposal is generic, and can be applied to MCTS whatever the considered problem. The purpose of our approach is to increase the system ability to memorize data, and also its ability to adapt it to close applicative contexts. The overall idea of BHRF is presented in Section III-A and an implementation in detailed in Section III-B.

A. Background History Reply Forest

We propose to build up an independent data-structure to complement the Monte Carlo tree and allow transversal knowledge memorizing. In order to improve the *roll-out* reliability, we choose to exploit the self-acquired and long-term data to influence the *roll-out* policy. Moreover the purpose of this contribution is to produce a policy that improves the learning process using knowledge that is already gathered. Therefore, the sequence estimation process is fed by the current sequence played during the descent phase, according to the simulation outcome (Winning, Undefined, Loosing).

We then extract from the considered contexts small sub-sequences giving rise to many local contexts to learn. Progressively, the contexts are refined, i.e., their sizes increase. As presented in Figure 4, the accumulated knowledge is stored inside a forest. The root of each tree stands for a reply while considering a context formed by a path between a leaf and the root. The set of all reply trees defines the Background History Reply Forest (BHRF).

B. Knowledge accumulation and exploitation

During the *update* phase, MCTS and BHRF are independently updated using the same simulation outcomes. The BHRF update is detailed through two algorithms.

First, considering knowledge acquiring, we cover each move of the descent sequence and launch a reply tree update (Algorithm 1). Indeed, the whole descent sequence contains

Algorithm 1 *update* algorithm - Reply Forest

```
procedure UPDATEREPLYFOREST(
    descentSequence: Array<Move>,
    outcome: Result {Win, Undefined, Lost})
//descentSequence: moves selected in the last tree descent
//outcome: result of the simulation following the descent
  for i ← descentSequence.size −1 to 0 do
    rt: ReplyTree
    rt ← self.getReplyTree(descentSequence[i])
// the reply tree leading to move i is considered
    if opponentMove(i) then
      rt.updateTree(i,descentSequence,inv(outcome))
    else
      rt.updateTree(i,descentSequence,outcome)
    end if
  end for
end procedure
```

Algorithm 2 *update* algorithm - Reply Tree

```
procedure UPDATETREE(i: int,
    descentSequence: Array<Move>,
    outcome: Result {Win, Undefined, Lost})
//i: position of the reply move in the descentSequence
  nodeCreated: Boolean
  mv: Move
  nodeCreated ← false
  childNode, lastNode: Node
  lastNode ← self.getRoot()
// the root node of the tree corresponds to the reply
  while i > 0 && ¬ nodeCreated do
    i ← i −1
    mv ← descentSequence[i]
    childNode ← lastNode.getDirectChild(mv)
    if childMove == null then
      childMove ← lastNode.createChild(mv)
      childMove.updateMean(outcome)
      nodeCreated ← true
    else
      childMove.updateMean(outcome)
      lastNode ← childNode
    end if
  end while
end procedure
```

many different context-move associations to memorize. The details of the tree update process is presented in Algorithm 2. As for MCTS, the node estimates are updated each time a stored sequence matches with the current one. For each reply tree, new nodes are regularly added and the whole structure is kept over the turns. We do not restrain accumulation and gather as much data as possible to refine contexts.

Second, considering knowledge exploitation presented in Algorithm 3, the *roll-out* policy controls the involvement of BHRF during the simulation process. Within the set of legal replies, the process can select a move among the ones matching a memorized context.

Algorithm 3 *roll-out* policy

```
procedure ROLLOUTPOLICY(lastMoves: Array<Move>) :
Move
//lastMoves: moves previously selected (temporal context)
  candidate: Move
  lstCandidate: List<Move>

//Probability ε to use BHRF
  if randomValue() < ε then
    for m: Move ∈ legalMoves() do
// Gets the best estimated candidate matching the context
      candidate =
        getReplyTree(m).BestCandidate(lastMoves)
      if candidate ≠ null then
        lstCandidate.add(candidate)
      end if
    end for
// Selects the reply according to Equation 1
    candidate =
      BestCandidateSoftMaxUct(lstCandidate)

    if randomValue() < candidate.uctEstimate() then
      return candidate
    else
      return defaultRandomMove()
    end if
  end if
end procedure
```

Since a richer context defines a more accurate knowledge, we promote the use of the longest context sequences. A *softMax* policy picks up one sequence among the selected ones based on their UCT estimates as follows[1]:

$$P(r|c) = \frac{\bar{x}_{r|c} + b \times \sqrt{\frac{\ln \sum_{i \in \mathscr{C}} n_{i|c}}{n_{r|c}}}}{\sum_{i \in \mathscr{C}} P(i|c)}, \tag{1}$$

where

r : legal reply
c : context
$\mathscr{C}$: set of legal replies for context c
$\bar{x}_{r|c}$: average result of r in c
b : UCT bias term
$n_{r|c}$: selection number of r in c

The associated reply is finally selected according to its estimate value. If no sequence matches, the default policy is applied. The ϵ parameter sets the using rate for BHRF in the simulation process.

[1]the bias term b has been set to 0.7 empyrically

IV. Experimental results

In this section, we study the influence of this self-acquired knowledge over the learning process. The experiments protocol is exposed in Section IV-A. The results presented in Section IV-B show that BHRF successfully catches the knowledge of the tree search and Section IV-C highlights that this tree knowledge may successfully complement an expert *roll-out* policy.

A. Experimental setup

The BHRF heuristic has been implemented using the open source framework *Fuego* (version 1.1) [18]. This framework offers the main enhancements for MCTS computer-go programs such as UCT and expert knowledge. In this program, the expert knowledge is used to initialize the new node of the tree search and also for the *roll-out* policy.

In the following experiments, the program with the BHRF heuristic competes against the same baseline program without BHRF[2]. The common settings of both programs are the same (if not mentioned). The settings we will further consider in the experiments are the following:

- Board size (△): `9x9`,`19x19`: determines the difficulty of the game played. The search space is huge on 19x19 and the program has to focus even more on game state of interest. Moreover games on wider boards produce more complex situations which may not be covered by expert knowledge.
- *roll-out* simulations (▼): `1k, ..., 10k, 30k`: corresponds to the maximum number of simulations granted. A larger value generates a more accurate tree knowledge and therefore a better *descent* policy.
- *roll-out* expert knowledge (■): `True, False`: defines whether the *roll-out* policy involves expert knowledge or not.
- BHRF: ϵ `= 0 .. 100%` (□): tunes the rate of exploitation of the self-acquired knowledge in the *roll-out* phase.

In this article we mainly focus on the potential for using such a knowledge, rather than on the next-step optimisation. That is why we considers both programs with equal number of simulations rather than equal time. Considering our lightly optimized BHRF algorithms and a middle range hardware configuration (Intel(R) Core(TM) i7-2600 CPU 3.40 GHz with 8GB memory), the BHRF module tends to slow down the computing time from 4 to 12 times, according to the game size and the *roll-out* simulation number. The present implementation is not competitive on equal time settings but provides a substantial improvement in the learning quality.

B. Increasing efficiency due to Self-acquired knowledge

For these experiments, competing programs are both set without expert *roll-out* knowledge but both programs initialize the node value using prior expert knowledge. In Table I, we report the values of BHRF knowledge for 9x9 and 19x19 game sizes with the maximum number of *roll-out* simulations allowed.

[2]All game results are provided with 95% confidence interval

△ Board size = 9x9/19x19 ▼ Simulations = 10k/30k/100k
■ Expert roll-out = FALSE □ ε = 100

Simulations	**10000**	**30000**	**100000**
goban 9x9	$+17.3\% \pm 1.6$	$+16.9\% \pm 2$	$+18\% \pm 2.6$
goban 19x19	$+24.3\% \pm 3.5$	$+26.6\% \pm 2.5$	$+27.7\% \pm 3.4$

Table I: Success rate for the BHRF approach (opposed to the same configuration without BHRF)

The program that considers self-acquired knowledge, significantly outperforms the baseline program in all the configurations and whatever is the number of allowed simulations. Pool-RAVE [11] provides similar results for the game of Go without expert *roll-out* knowledge[3]. These results confirm further the interest of using knowledge from the search tree in the *roll-out*.

One can note that for 19x19 game size, BHRF slightly stresses its advance while the *roll-out* simulation increases. As the 19x19 game size involves a huge combinatorial space, it is suitable to enhance the difference between different programs' efficiency. Whatever the considered settings, BHRF highly outperform the baseline program.

In order to appreciate the BHRF ability to manage and benefit from complementary knowledge, we choose to vary the available number of *roll-out* simulations available for BHRF, while keeping it constant (fixed to *10k*) for the baseline program. As shown in Figure 5, BHRF outperforms the baseline program as soon as it reaches the half of the available number of *roll-out* simulations of the baseline program. However, as mentioned previously, the process of real-time self-acquiring knowledge is time-consuming and a further optimization is required before being time-competitive with the current programs.

These results are very encouraging, and show that BHRF successfully embeds the knowledge of the tree. The knowledge used to initialize the tree node is the same that the one normally used in expert *roll-out* policies. Therefore BHRF integrates these knowledge along with the knowledge acquired by the simulation outcomes.

C. Competition with full expert knowledge programs

The knowledge accumulated by BHRF is exploited during *roll-out*, but how does this knowledge face expert *roll-out* rules used by professional-level computer-go programs ?

In this section, we choose to involve expert knowledge heuristics for BHRF *roll-out* as a second choice. When no move is selected by the BHRF *roll-out* policy, the standard rules originating from Fuego *roll-out* policy are applied. BHRF competes then with Fuego set to the best of its ability.

In Figure 6, we show that BHRF outperforms Fuego when we use BHRF data moderately (ϵ around 15). The number of simulation was set to 30k in order to accumulate substantial data about the game. A low ϵ value involves more exploration through the general MCTS process. BHRF nevertheless allows

[3]Their results are provided only for the 9x9 game size.

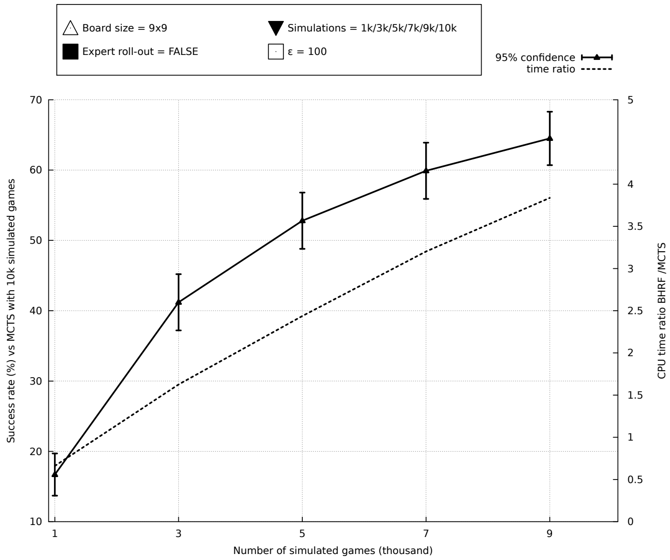


Figure 5: Success rate and CPU time ratio for BHRF approach considering *roll-out* number variations (opposed to the same configuration without BHRF, *roll-out* number fixed to *10k*)

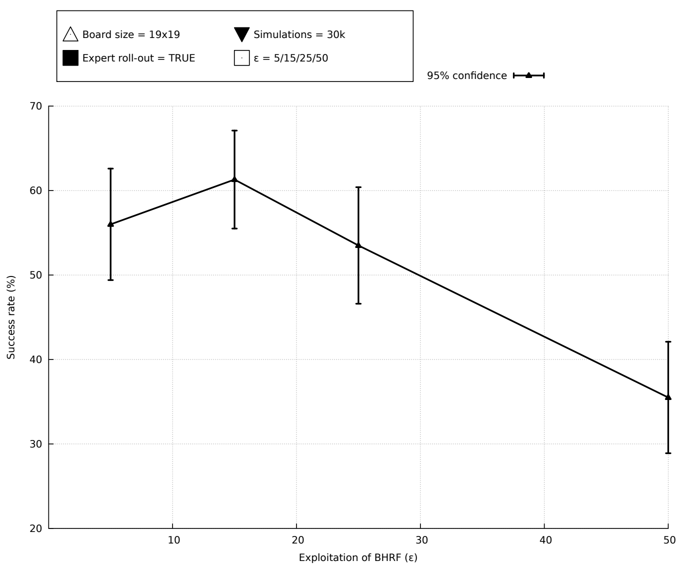


Figure 6: Success rate on 19x19 with standard state of art heuristics

to significantly increase the global performance while memorizing efficient situated patterns.

The expert knowledge involved in the *roll-out* policy plays locally, around the last move generated. On 9x9 board local fights cover quickly the whole board but on 19x19 board, local fights have to consider also the situation in other area of the board. As mentioned before, our data-structure embeds the expert and the self-acquired knowledge of the tree. BHRF knowledge may adjust this locality according to the state covered in the tree. Therefore the enhanced *roll-out* policy benefits from knowledge adapted to the real situation.

V. Conclusion

This paper proposes new enhancements to complete the well-known MCTS search process in the context of combinatorial games. We show that a better assimilation of the knowledge learnt by MCTS may enhance the performance of the system. As presented in this paper, the knowledge stored in the search tree is not prone to be re-exploited. A promising way is to consider MCTS as a cognitive system. Indeed, a better assimilation of this knowledge allows to adapt it to different situations and may avoid to learn redundant patterns among the branches [17] for instance. Moreover, such an approach may provide better insights on how the system considers its simulated experience and therefore the underlying mechanisms of MCTS.

The data-structure detailed in this paper, is a raw manner to memorize adaptive knowledge coming form the search tree. The presented results show that this data-structure successfully catches such a knowledge (Section IV-B) and this knowledge may actually complement expert knowledge (Section IV-C). In particular, a professional program combined with BHRF achieves up to a 11% increase in performance. These results points out the potential of such an approach though the slow down of the learning process prevents from experiments with constant time yet. We decided to apply our algorithm to the game of Go because this problem is demanding in terms of knowledge, nevertheless the current implementation is designed for a general-game perspective. A more time-efficient implementation may consider characteristics of the game such as the locality of the reply but this was beyond the scope of the present paper.

References

[1] B. Bouzy and T. Cazenave, "Computer Go: An AI oriented survey," *Artificial Intelligence*, vol. 132, no. 1, pp. 39–103, 2001.

[2] S. Gelly and D. Silver, "Achieving Master Level Play in 9×9 Computer Go," in *Proceedings of AAAI*, 2008, pp. 1537–1540.

[3] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 1, pp. 1–43, 2012.

[4] A. Bourki, G. Chaslot, M. Coulm, V. Danjean, H. Doghmen, J.-B. Hoock, T. Hrault, A. Rimmel, F. Teytaud, O. Teytaud, P. Vayssire, and Z. Yu, "Scalability and Parallelization of Monte-Carlo Tree Search," in *Computers and Games*. Springer Berlin Heidelberg, 2011, vol. 6515, pp. 48–58.

[5] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. Cambridge Univ Press, 1998.

[6] 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*. Springer Berlin Heidelberg, 2010, vol. 6048, pp. 1–13.

[7] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo Planning," in *Machine Learning: ECML 2006*. Springer Berlin Heidelberg, 2006, vol. 4212, pp. 282–293.

[8] Y. Wang and S. Gelly, "Modifications of UCT and sequence-like simulations for Monte-Carlo Go," in *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, 2007, pp. 175–182.

[9] D. Silver and G. Tesauro, "Monte-Carlo Simulation Balancing," in *Proceedings of the 26th International Conference on Machine Learning*. Omnipress, 2009, pp. 945–952.

[10] M. Tak, M. Winands, and Y. Bjornsson, "N-Grams and the Last-Good-Reply Policy Applied in General Game Playing," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 2, pp. 73–83, 2012.

[11] A. Rimmel, F. Teytaud, and O. Teytaud, "Biasing Monte-Carlo Simulations through RAVE Values," in *Computers and Games*. Springer Berlin Heidelberg, 2011, vol. 6515, pp. 59–68.

[12] L. Lew, "Modeling Go game as a Large Decomposable Decision Process," Ph.D. dissertation, Warsaw University - Faculty of Mathematics, Informatics and Mechanics, 2011.

[13] H. Baier and P. Drake, "The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 2, no. 4, pp. 303–309, 2010.

[14] H. Baier, "Adaptive Playout Policies for Monte-Carlo Go," Master's thesis, Institut für Kognitionswissenschaft, Universität Osnabrück, 2010.

[15] P. Baudi, "MCTS with Information Sharing," Ph.D. dissertation, Charles University in Prague - Faculty of Mathematics and Physics, 2011.

[16] B. Childs, J. Brodeur, and L. Kocsis, "Transpositions and Move Groups in Monte Carlo Tree Search," in *Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On*, 2008, pp. 389–395.

[17] P. Drake, "The Last-Good-Reply Policy for Monte-Carlo Go," *International Computer Games Association Journal*, vol. 32, no. 4, pp. 221–227, 2009.

[18] M. Enzenberger, M. Muller, B. Arneson, and R. Segal, "FUEGO - an Open-Source Framework for Board Games and Go Engine Based on Monte-Carlo Tree Search," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 2, no. 4, pp. 259–270, 2010.