A new self-acquired knowledge process for Monte Carlo Tree Search

André Fabbri, Frédéric Armetta, Éric Duchêne and Salima Hassas

Laboratoire GAMA

Abstract. Computer Go Game is one of the most challenging field in Artificial Intelligence Game. In this area the use of Monte Carlo Tree Search has emerged as a very attractive research direction, where recent advances have been achieved and permitted to significantly increase programs efficiency.
These enhancements result from combining tree search used to identify best next moves, and a Monte Carlo process to estimate and gradually refine a position accuracy (estimation function). The more the estimation process is accurate, the better the Go program performs.
In this paper, we propose a new approach to extract knowledge from the Go tree search which allows to increase the evaluation function accuracy (BHRF: Background History Reply Forest). The experiments results provided by this new approach are very promising.

1 Introduction

The Go game is a deterministic fully observable two player game. Despite its simple rules, it stands for one of the great challenge in the field of AI Game.

Go tactical and strategic decisions are extremely difficult to tackle, since each move could have a high impact a long time after having been played. A single stone move could completely change the board configuration and therefore, there is no static evaluation function available during the game [22]. Professional human players succeed where programs fail, thanks to their expertise and knowledge acquired from their long experience.

Recent developments in Monte Carlo Tree Search allowed to considerably increase program's efficiency. These enhancements result from combining Tree Search used to identify best next moves to a Monte Carlo process based on random playing attempts to estimate and gradually refine a position accuracy (estimation function).

Monte Carlo Tree Search programs are able to reach up a professional level by simulating thousands of pseudo random games [16]. However these program's performances does not scale with computational power increasing [2].

A promising way to increase program's efficiency arises from learning on-line local knowledge to enhance the relevance of random simulations used to evaluate positions [1,17,18].

In this context, one can note that the issue is not only focused on the computing power available, but in our ability to manage additional more sophisticated

self-acquired knowledge. Then, in the rest of the paper, we will focus on this ability. Further work will propose additional enhancements to optimize the process associated to knowledge management in order to decrease the computing-resources consumption.

In this paper we propose a new approach to collect the knowledge acquired through the tree search of estimated positions. Our proposal appears as a complementary data structure. As described in section 3, a tree search is used to define what are the best next moves to play from a completly described current position. The structure we propose to build addresses a more abstract question: if a set of relatives moves are played, what to play next ? We show with our approach presented in section 4 that a natural manner to tackle this question is to constitute a Background History Reply Forest (BHRF). We then describe how to build and maintain this generic forest and how to exploit it. Theses two points raise new questions, but we show that a first straightforward setup produces good results. Some promising experimental results are presented in 5 and show that using BHRF allows to play better.

The next section presents the general structure of knowledge involved in a learning process. The section 3 focus on the kind of knowledge used in the existing Go programs. Section 4 introduces a Background History Reply Forest (BHRF) for the Monte Carlo Tree Search. The last section sums up experiments carried on the model. A conclusion is presented in 6.

2 Knowledge in computer Go

While a game is running, for each step, a Go player aims at selecting the best move according to the game overall state. The overall game state involves any information related to the current game such as the board setup, the history of played moves, local fights and also the opponent's level. During the decision process, players need to select the relevant parts of the game to focus on. Human players accumulate knowledge by studying standard techniques, watching professional games or interacting with other players. Computer programs encode directly Go expert knowledge (apriori knowledge) or create their own knowledge using machine learning technic. This information is evaluated and stored in a data structure (knowledge acquisition or *learning*) for a further exploitation (knowledge exploitation or *planning*). Reinforcement learning methods iteratively apply these two steps to progressively build an effective policy [21,19].

2.1 Go knowledge acquiring

Knowledge is useful to evaluate a position (current state or targeted state resulting from a set of planned moves). Either generic (fitting many cases) or specific knowledge could be used to do this evaluation.

For instance, understanding the shapes formed by the stones on a Goban is a relevant way to solve local fights. In this case, local patterns are a powerful way to encode Go expert knowledge [19,12].

An other way is to consider a knowledge based on high-level characteristics of the game: groups, groups stability, influences between groups, etc [11]. The representation can then focus on the intersection around the last move played to find local replies [22].

The value of this knowledge generally represents the probability to win if we reach this game state representation. The quality of the estimator determines the accuracy of the considered knowledge. The value given by poor estimators are not reliable and have to be considered with caution. Several estimators for the same knowledge can be combined to produce a more effective estimation. The knowledge values are computed based on professional game records [4,20], professional players comments [13] or self-playing [19]. These knowledge values will be exploited to choose an action according to the current game state.

2.2 Go knowledge exploitation

The policy maps an action to each game state. The action is selected based on the knowledge matching that game state. A high knowledge value means a good move but the knowledge nature and its estimator might be misleading. Therefore uncertainty is generally inserted in the policy to mitigate the influence of wrong estimations. The ϵ-*greedy* policy selects with a probability ϵ the action associated to the highest knowledge value otherwise it generally plays a random move. The *softmax* policy selects each action with a probability depending on the associated knowledge value [21].

A policy would be selected according to its context of application. During a tutorial process, a policy might tolerate exploratory moves but during a challenge the policy has to select the best possible action.

2.3 Reinforcement learning

Reinforcement learning is a way to manage learning in an autocatalytic way. The program is not taught what to do and learns therefore by its own experience [21]. *Temporal difference learning* is one of the most applied method for such problems and has been successfully applied to computer games. This method dynamically bootstraps its knowledge from simulated games.

The policy evolves as the simulations runs and will influence the incoming simulations. Since the knowledge is built on-line, the policy has to deal with exploiting the accumulated knowledge (*exploitation*) or gathering more knowledge (*exploration*). A good policy should produce accurate simulations to enable the learning process. A too strong policy can actually lead to a weaker program [21].

$TD(\lambda)$ methods reinforce their knowledge values according to the estimation difference of the game state between two time-step. A Monte Carlo method is a *temporal difference* method which reinforcement depends on the final game state of the simulation [19].

3 Monte Carlo Go programs

The main idea of Monte Carlo Tree Search is to build a tree of possible sequence of actions. The tree root corresponds to the current board situation and each child node is a possible future game state. The tree will be progressively expanded towards the most promising situation by repeating the 4 phases: *descent*, *roll-out*, *update* and *growth* (Fig.1).

During the *descent* phase, the simulation starts from the root node and progresses trough the tree until it reaches a game state outside of the tree.

For each node the program will iteratively select the best action with respect to the *descent* policy. Once it leaves the Monte Carlo Tree, the *roll-out* phase generates the remaining moves according to the *roll-out* policy until the game reaches a final state. The *update* phase propagates the final game results in the node reached during the *descent* and the *growth* phase appends the first game situtation ouside of the tree to the overall structure [5].

Monte Carlo Go programs involve two kinds of knowledge to guide the simulations from the current game state. The Monte Carlo Tree stores knowledge on-line. This knowledge is exploited during the *descent* phase. The current best programs exploit Go domain specific knowledge in the *roll-out* phase because the tree knowledge is no more available. In the last section we will present methods that attempt to dynamically build up knowledge for the *roll-out* phase.

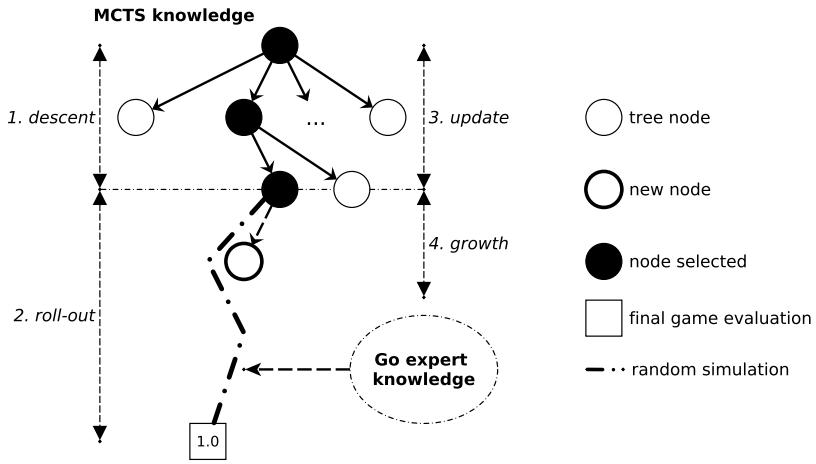

Fig. 1. General MCTS process

3.1 Monte Carlo tree

In the Monte Carlo tree, each node is a knowledge piece that associates a possible future board setups with an accuracy value estimated from previous Monte Carlo simulations. Board setups are very precise game state representations which will be progressively selected as the simulations run. However the built tree depends on the current board situation. Local structures are not shared between branches leading to a new kind of horizon effect [6] and at each new turn only a subtree is kept. All previously acquired knowledge in other branches is forgotten.

At each step of the *descent* phase, the policy selects in a greedy way the best action based on the child's node value or a default value for the resulting states outside of the tree [22]. The node's estimator will have a huge influence on the policy's behaviour. Since the policy is not perfect and the estimation sample is too small, the mean of Monte Carlo rewards (MCTS value) is not a reliable indicator on the long-term run [8]. The Upper Confidence bound applied to Tree estimator (UCT value) reveals to be a good trade-off between exploration and exploitation [15]. Recent results show that, in practice, a combination between the MCTS value and a biased mean estimator called RAVE ensures a good exploratory behaviour [10,8] and also minimizes the knowledge lost at each new turn, but without control.

3.2 Go a priori knowledge

Contrary to the Monte Carlo tree, the Go expert knowledge is independent from the overall stones disposition and therefore can be exploited for a broader set of a board setup. The purpose of this knowledge is to guide the simulation after having left the tree structure. However the *roll-out* policy should not be too strong to do not disturb the learning process [9]. The knowledge encoded corresponds generally to static Go tactical rules based on the current board configuration. Sequence-like policies brought a substantial improvement by searching around the last opponent move [22]. If no rules fit the current situation, the policy plays randomly. Other *roll-out* policies use a linear combination of local pattern. The weights are generally computed with off-line intensive machine learning [9] but recent works obtained promising results by tuning them on-line [14,19].

3.3 Knowledge collected from Game

Several works aim at dynamically learn knowledge relative to the current game in order to exploit it during the *roll-out* phase (see Fig.2). The game chosen state representations are generally small move sequences or patterns. As for the Monte Carlo tree, this knowledge will be built in a autocatalytic manner but this knowledge will persist over the turns.

Pool RAVE [18] exploits the RAVE values in the Monte-Carlo tree to guide the *roll-out*. The best RAVE values stored in a node crossed during the *descent* phase are pooled to biase the next simulation. The *roll-out* policy proposed for the game of Go applies first a "fill-board heuristic" [3] to ensure a good

simulation diversity and then generates the moves associated to one of the pooled RAVE values. Hence the RAVE value reexploited contribute directly to their own reinforcement.

Contextual Monte Carlo [17] stores tiles of two moves played by the same player. Each tiles has Monte Carlo mean value updated when the two moves appears in the same simulation. The main idea is to link the expected success with a couple of moves played by the same player. During the *roll-out* phase a ϵ-*greedy* policy will select the best move according to the last player's move.

Last Good Reply Forgetting heuristic [1] associates a reply to each sequence of one or two consecutive moves. For each encountered sequence the program stores a single reply. In the *update* phase all the reply sequences are updated or forgotten according to the simulation result. Over the simulations the replies are frequently updated but the most persisting moves are spatially local replies [6,1]. In the *roll-out* phase the policy successively tries to apply the reply to the two previous ones, the reply to the previous move or a move generated from a Go expert knowledge policy if no appropriate reply is stored.

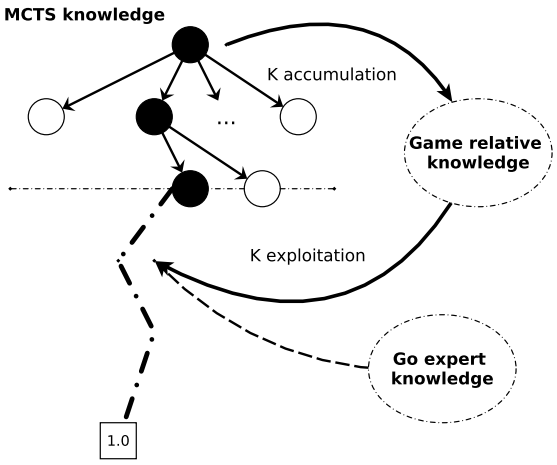

Fig. 2. Game relative knowledge for MCTS process

4 Proposal

In this paper, we propose a new method to learn knowledge collected from game (game-based knowledge) persistent over the turns. We actually assume that we can extract knowledge from the Monte Carlo tree and reuse it during the *roll-out* phase. Therefore we build an independent data structure similar to the Monte Carlo tree to store game based knowledge. The *roll-out* policy will be modified to consider this new form of knowledge.

4.1 Background HistoryReply Forest

A Monte Carlo node represents a future possible board setup. If we consider the background history of the board rather than the position of the stones, a board setup is also the sequence of moves played since the beginning of the match. Hence each child node of the tree is a possible reply to that sequence. Our idea is to extend the LGRF heuristic to catch the knowledge stored in the tree over the simulations. The size of the previous sequence will be lengthened and the reply estimation will be based on the tree values. This knowledge will be more biased than the one provided by Monte Carlo nodes but it will be exploited in the *roll-out* phase and will persist over the turns.

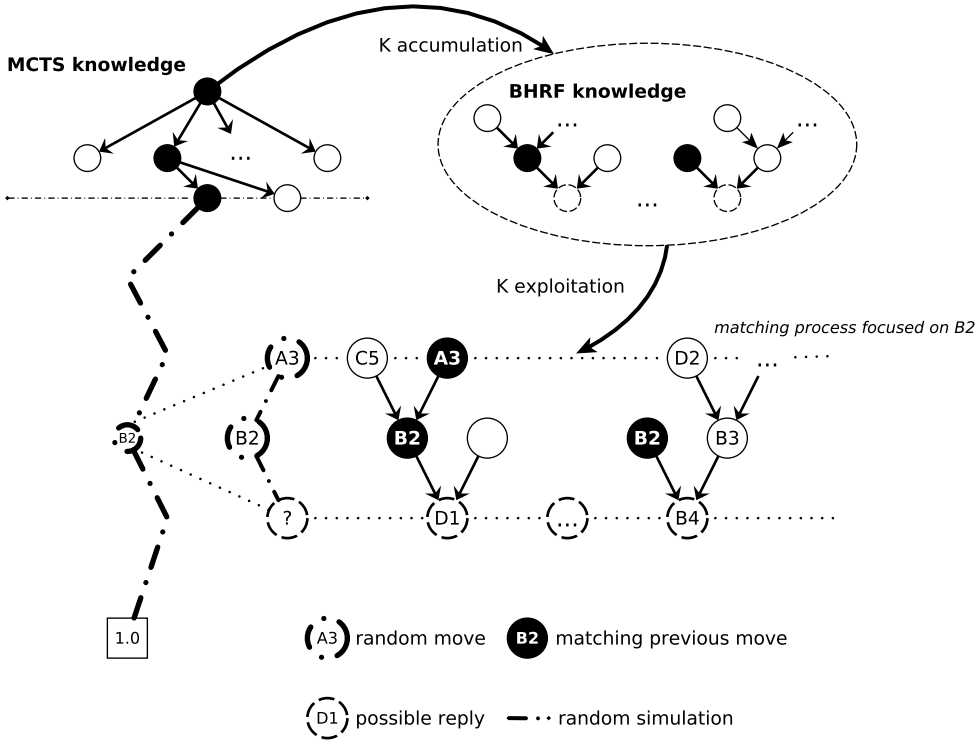

Fig. 3. BHRF knowledge exploitation

Hence we build a forest of search trees such that each tree root is a potential reply and each child node a previous sequence to this reply (Fig.3). The value associated to each node comes from the Monte Carlo tree. In the *update* phase all nodes corresponding to sequences chosen in the *descent* phase are reinforced. The search trees are progressively expanded in the *growth* phase as done in

the Monte Carlo tree [5]. This knowledge is then exploited during the *roll-out* phase, according to the previous moves played. The selected nodes with the longest corresponding sequences will be the most dependent on the game state.

4.2 BHRF exploitaiton

At each *roll-out* game state, we choose a reply among all legal positions on the board. The program progresses through the associated tree search to select the nodes with the longest previous sequences (up to a maximum size parameter *depth*). The policy computes on-line the UCT value for the nodes with the same preceding sequence according to their previous Monte Carlo rewards.

Due to the biased nature of this knowledge and to do not constrain the simulation diversity we implemented a non determinist policy to generate the moves at each *roll-out* game state. The policy plays the root reply associated to a node selected among those with the longest preceding sequence found. If no reply move was played by the policy, a default *roll-out* policy is applied.

The *softmax* policy selects a node according to a *softmax* distribution based on their UCT value and plays the associated move with a probability depending on its UCT value multiplied by a parameter α. This parameter aims to generate random moves rather than replies poorly estimated by the forest.

Algorithm 1 *softmax* policy for BHRF knowledge

```
curLength = 0
for m in legalMoves do
  i = replyTree[m].selectNode(depth,lastmoves)
  nodeSelection.add(i)
  if n.length > curLength then
    curLength = n.length
  end if
end for
n = nodeSelection.softMaxUct(curLength)
if randomValue() < α × n.uctValue then
  return n.rootReply
else
  return defaultRandomMove()
end if
```

5 Experimental results

We programmed this proposed knowledge-based model on top of the state of art program Fuego [7]. To prove the effectiveness of our approach we tested the BHRF player against a Monte-Carlo player using a random *roll-out* policy. We disabled the expert heuristics involved in the Fuego *roll-out* policy except

for the random move generation[1]. The experimental results were carried out on a 9x9 goban against the same program without the BHRF heuristic. The implementation proposed is not thread-safe and time-consuming. Therefore to provide a fair comparison we desactivated the clock and both programs run, on a single thread, 10000 Monte-Carlo simulations at each turn.

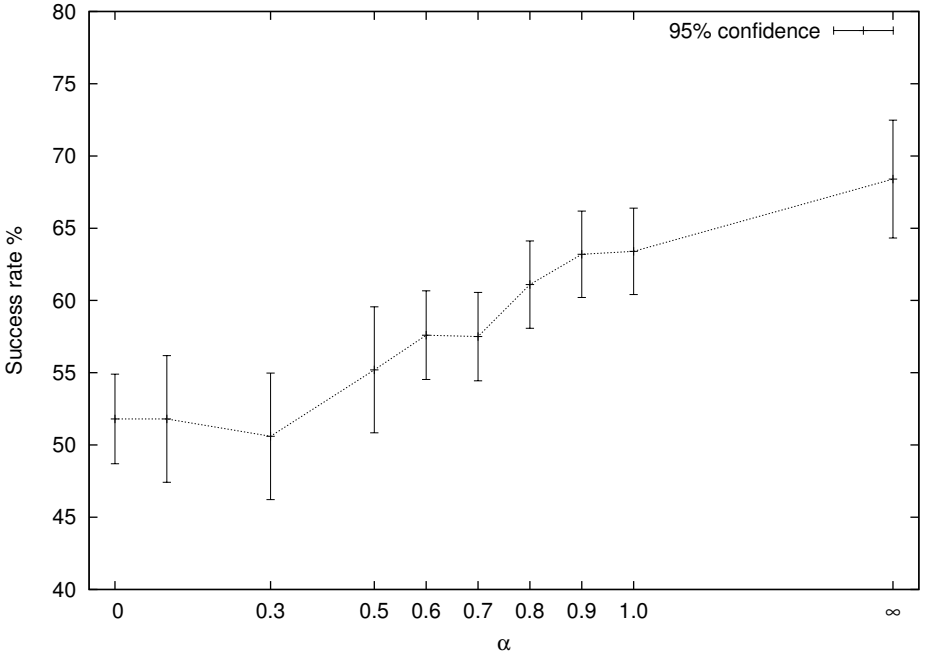

Fig. 4. BHRF success rate against Fuego over α

We first study the influence of the parameter α for the *softmax* policy. This parameter tunes the involvement of the BHRF knowledge during the *roll-out* phase (see Algo. 1). As we see in the figure (Fig. 4), the performance increases with respect to α. The programs actually performs best for $\alpha = 100$ where the BHRF replies are applied systematically during the *roll-out* phase if a BHRF node matches (only 2% are rejected).[2]

Hence we focus on the *depth* parameter for $\alpha = 100$. The *depth* parameter determines the "history relevance" of the selected knowledge. As the maximum *depth* increases the reply estimates should be more accurate. The experiments were carried out up to a *depth* of 10 because of execution constraints. Further experiments required an optimised BHRF implementation. As we see in the

[1] Fuego default random move generation use heuristics to avoid "suicidal" moves

[2] tirer une conclusion comme quoi BHRF possède un bon comportement exploratoire par elle-même ?

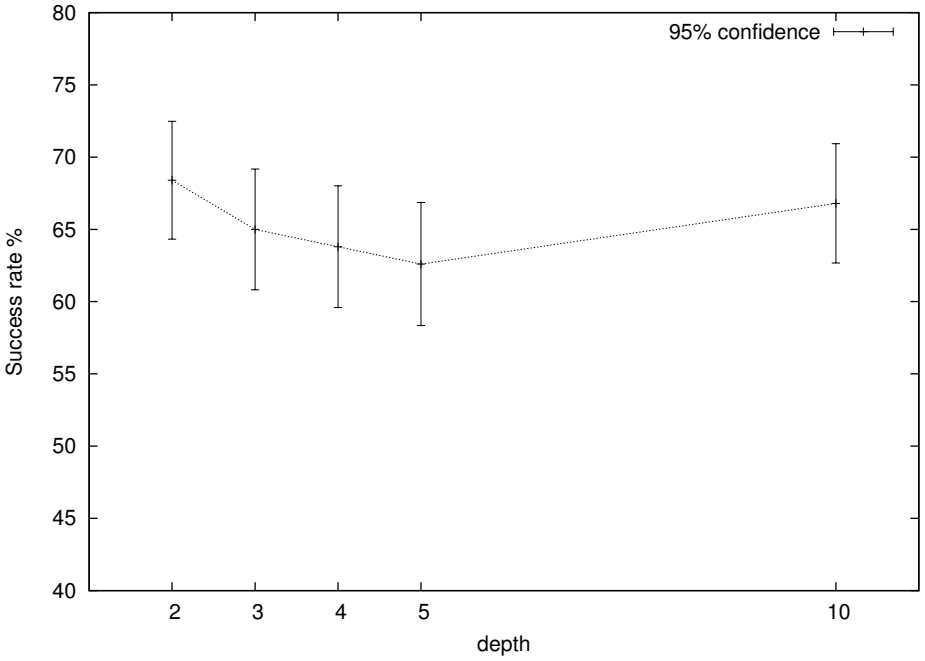

Fig. 5. BHRF success rate against Fuego over *depth*

figure (Fig. 5), the overall performance of the program slightly decreases as the depth grows.[3]

Finally we compare the performance of the BHRF heuristic according to the number of Monte-Carlo games simulated at each turn. Table 1 shows that BHRF performs best for 10000 simulations search. The resulting Background HistoryReply Forestfor deep Monte Carlo search produces better estimators and actually matches to a wider set of game situation. About 90% of *roll-out* game state have a legal BHRF reply when MCTS runs 10000 games against 80% for 1000 games simulated.

Table 1. Comparative between 1000 and 10000 games simulated for the *softmax* policy

α	1		100	
games simulated	1000	10000	1000	10000
success rate %	58.7 ± 2.49	63.4 ± 2.99	58.5 ± 2.49	68.4 ± 4.08

These results show an overall improvement when reusing game-based knowledge, that is extracted from the Monte Carlo tree. As we mentionned before,

[3] tirer une conclusion comme quoi il ne faut peut-être pas sélectionné en priorité les noeuds les plus profond ?

our main concern was to build self-acquired knowledge to improve Monte Carlo simulations. The computing time of this algorithm is still higher than classical MCTS algorithm (up to 10 times) but improvements could be made. Indeed the heuristic searches at each *roll-out* state among all the potential moves on the board contrary to the Go expert policy which generates actions that are around the last move played [22,7]. Furthermore we would like to point out that our main concern was to show the good impact of Game-based knowledge model. An optimised version will requires a deep modification of the Fuego libraries.

6 Conclusion

In this paper we present a new approach to complement the Monte Carlo Tree Search programs. The proposed framework is inspired from the reinforcement learning paradigm, and aims at distinguishing incorporated knowledge from its exploitation.

The proposed model extracts game-based knowledge from the Go Tree Search to introduce more accurate moves through simulations. The incorporated knowledge is based on history data and tends to find a trade-off between the specific knowledge stored in the tree and more biased game-relative heuristics.

We show with our approach that a natural manner to complement Go tree search knowledge is to constitute a Background History Reply Forest (BHRF). We then describe how to build and maintain this generic forest and how to exploit it. These two points raise new questions, but we show that a first straightforward setup provides good results.

The first proposed results underline the efficiency of our model, and allows to beat the classical MCTS algorithm with similar setting for up to 2/3 of the considered games. This encourages us to carry on the experiments on a larger board or with other settings (longer sequences, use of BHRF to compute initial values of leaf for the Go tree search, etc.).

References

1. Baier, H., Drake, P.: The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte-Carlo go. IEEE Transactions on Computational Intelligence and AI in Games 2(4), 303–309 (2010)
2. Bourki, A., Chaslot, G., Coulm, M., Danjean, V., Doghmen, H., Hoock, J., Hérault, T., Rimmel, A., Teytaud, F., Teytaud, O., et al.: Scalability and Parallelization of Monte-Carlo Tree Search. In: Proceedings of Advance in Computer Games 13 (2010)
3. Chaslot, G., Fiter, C., Hoock, J., Rimmel, A., Teytaud, O.: Adding expert knowledge and exploration in Monte-Carlo Tree Search. Advances in Computer Games pp. 1–13 (2010)
4. Coulom, R.: Computing Elo Ratings of Move Patterns in the Game of Go. In: Computer Games Workshop, Amsterdam, The Netherlands (2007)
5. Coulom, R.: Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. Computers and Games pp. 72–83 (2007)

6. Drake, P.: The Last-Good-Reply Policy for Monte-Carlo Go. International Computer Games Association Journal 32(4), 221–227 (2009)
7. Enzenberger, M., Muller, M., Arneson, B., Segal, R.: FUEGO - an Open-Source Framework for Board Games and Go Engine Based on Monte-Carlo Tree Search. IEEE Transactions on Computational Intelligence and AI in Games 2(4), 259–270 (2009)
8. Gelly, S., Kocsis, L., Schoenauer, M., Sebag, M., Silver, D., Szepesvári, C., Teytaud, O.: The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions. Communications of the ACM 55(3), 106–113 (2012)
9. Gelly, S., Silver, D.: Combining Online and Offline Knowledge in UCT. In: Proceedings of the 24th international conference on Machine learning. pp. 273–280. ACM (2007)
10. Gelly, S., Silver, D.: Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go. Artificial Intelligence (2011)
11. Graepel, T., Goutrie, M., Krüger, M., Herbrich, R.: Learning on Graphs in the Game of Go. Artificial Neural Networks—ICANN 2001 pp. 347–352 (2001)
12. Helmut, A.M.: Board Representations for Neural Go Players Learning by Temporal Difference. In: Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on. pp. 183–188. IEEE (2007)
13. Hoock, J., Lee, C., Rimmel, A., Teytaud, F., Wang, M., Teytaud, O.: Intelligent Agents for the Game of Go. Computational Intelligence Magazine, IEEE 5(4), 28–42 (2010)
14. Huang, S., Coulom, R., Lin, S.: Monte-Carlo Simulation Balancing in Practice. Computers and Games pp. 81–92 (2011)
15. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo Planning. Machine Learning: ECML 2006 pp. 282–293 (2006)
16. Lee, C., Wang, M., Chaslot, G., Hoock, J., Rimmel, A., Teytaud, O., Tsai, S., Hsu, S., Hong, T.: The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. Computational Intelligence and AI in Games, IEEE Transactions on pp. 73–89 (2009)
17. Rimmel, A., Teytaud, F.: Multiple Overlapping Tiles for Contextual Monte Carlo Tree Search. Applications of Evolutionary Computation pp. 201–210 (2010)
18. Rimmel, A., Teytaud, F., Teytaud, O.: Biasing Monte-Carlo Simulations through RAVE Values. Computers and Games pp. 59–68 (2011)
19. Silver, D., Sutton, R., Müller, M.: Temporal-difference search in computer Go. Machine Learning pp. 1–37 (2012)
20. Stern, D., Herbrich, R., Graepel, T.: Bayesian Pattern Ranking for Move Prediction in the Game of Go. In: Proceedings of the 23rd international conference on Machine learning. pp. 873–880. ACM (2006)
21. Sutton, R., Barto, A.: Reinforcement Learning: An Introduction. Cambridge Univ Press (1998), [online] available at http://webdocs.cs.ualberta.ca/ sutton/book/ebook/the-book.html
22. Wang, Y., Gelly, S.: Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In: IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii. pp. 175–182 (2007)