Developments in Monte-Carlo Proof-Number Search

Jahn-Takeshi Saito Guillaume Chaslot Jos W.H.M. Uiterwijk
H. Jaap van den Herik Mark H.M. Winands

MICC-IKAT Universiteit Maastricht
P.O. Box 616, 6200 MD Maastricht, The Netherlands

{*j.saito, g.chaslot, uiterwijk, herik, m.winands*}*@micc.unimaas.nl*

Abstract

Over the years, proof-number search has successfully been applied to many game domains. Only recently, the combination of Monte-Carlo sampling and proof-number search was introduced. This proof-number enhancement applies Monte-Carlo sampling to initialize proof numbers. This article outlines Monte-Carlo Proof-Number Search and presents two refinements to the algorithm: (1) novel tuning of standard parameters, and (2) static pattern evaluation. Experiments yield that (1) tuning of parameters can lead to better results than recorded previously, and (2) static patterns are difficult to apply directly. Future work will focus on examining how Monte-Carlo sampling can be carried over to depth-first proof-number search.

1 Introduction

Proof-number search (*PNS*, [2]) is a tree-search algorithm for finding binary goals, i.e., goals which can either be reached (yielding a proof) or not reached (yielding a disproof).

Monte-Carlo Proof-Number Search (*MCPNS*, [12]) is a variation of *PNS*. It was able to double the speed required for proving goals compared to *PNS*. Yet several questions remained open. This article will investigate two of these previously unanswered questions: (1) Can further tuning of standard parameters speed up *MCPNS*? (2) Do small patterns suffice to represent domain knowledge (for the game of Go) such that their application speeds up the proof procedure?

The structure of this article is as follows. Section 2 provides an outline of *PNS* and *MCPNS*. Section 3 describes two possible improvements to *MCPNS* (1) tuning standard parameters, and (2) implementing pattern knowledge. Two small experiments are documented, one for testing each improvement. The results of the experiments are discussed and evaluated in Section 4. Section 5 concludes by summarizing this work and giving an outlook on future research.

2 PNS and MCPNS

The *PNS* algorithm performs search on AND-OR trees. This characteristic renders *PNS* an attractive choice for the two-player combinatorial games domain.

The improvements suggested to extend standard *PNS* can be divided into two categories. The first category contains procedural and implementational refinements. These yield speed gains by systematically introducing hash-tables and minimizing the cost of tree traversal (e.g., *df-pn* by [10], or *PN** by [13]). The second category contains heuristic extensions of *PNS* (e.g., [14], [15], [7]). Such extensions may help ordering branches to reach the proof or disproof by fewer node expansions.

The *MCPNS* variation is an instance of a heuristic move-ordering extension for *PNS*. It applies Monte-Carlo sampling (*MCS*, [1]) as follows. Random sequences of moves are launched for each heuristic evaluation to estimate the minimal number of moves required to prove or disprove a goal. This number then serves as heuristic evaluation for newly expanded nodes.

2.1 Proof-Number Search

PNS is a best-first search algorithm for proving or disproving binary goals in AND-OR trees. For OR-level nodes, proving one successor node is sufficient to prove the goal. For proving AND-level nodes, all successor nodes must be proved. For disproving OR- or AND-level nodes the corresponding reverse holds. *PNS*'s best-first heuristic is defined as follows: expand first the node that promises to require the least number of further node expansions to prove or disprove the goal.

A *proof number* (*pn*) and a *disproof number* (*dn*) are introduced for each node *M*. They keep record of the most promising node in the whole search tree. Following the heuristic above, *pn* is the number of nodes which are at least required to prove the goal at *M*. Analogously, *dn* is the number of nodes which are at least required to disprove the goal starting at *M*. We denote the corresponding *pn* and *dn* for some node *X* by $pn(X)$ and $dn(X)$ if required by

the context.

The *PNS* algorithm has two stages which are repeated during the search. In the first stage, *PNS* descends from the root node to a leaf node with lowest *pn* or *dn*. The numbers *pn* and *dn* do not increase along the path. In OR nodes, the branch with smallest *pn* is chosen for descending. In AND nodes the branch with smallest *dn* is selected. When a leaf node L with smallest value is found, L is expanded. The values of the newly expanded children are set as follows: $pn = 0$ and $dn = \infty$ in case the goal is proved, $pn = \infty$ and $dn = 0$ in case the goal is disproved, $pn = 1$ and $dn = 1$ otherwise. According to the number of its children $pn(L)$ and $dn(L)$ are set. The second stage tracks back from the leaf node to the root node. Starting at L, *pn* and *dn* are adjusted for all predecessors following the path back up to the root node.

The repetition of this cycle terminates when $pn(root) = 0$ or $pn(root) = \infty$. In the first case, the goal is proved as no further node requires to be expanded. In the second case, a disproof is reached because an infinite number of expansions would be required to prove the goal.

PNS produces the correct solution but requires maintenance of a large search tree.

2.2 Monte-Carlo Proof-Number Search

The basis for *MCPNS* is *MCS* which applies a simple principle. Random actions gain an evaluation for a given state S under investigation. In combinatorial games, the board configuration constitutes such a state. A random action here consists of a random sequence of legal moves. Each sequence is evaluated, e.g., by scoring the end position of a sequence. The value of S is a statistical aggregate of all random sequence values. The mean of the scores of all random sequences is an example for such an aggregate.

MCPNS launches N random sequences with a fixed length *la* (i.e., the number of moves) from the leaf node X to be evaluated. The statistical evaluation performed is a ratio. It takes into account the number of sequences yielding a proof (N_+) for setting the value of *dn*. It takes into account the number of sequences not yielding a proof (N_-) for setting the value of *pn*. It may seem paradoxical that counting fewer sequences proving the goal corresponds to the value of a greater proof number. The underlying rationale is that the proof number represents the minimal number of nodes which require expanding. Thus, a smaller proof number represents better chances of reaching the goal soon.

More formally the heuristic is $h_{pn}(X) = \frac{N_-+1}{N}$ and $h_{dn}(X) = \frac{(N_++1)}{N}$.

3 Extending *MCPNS*

As shown in [12], *MCPNS* can double the speed of the search process for Life-and-Death problems in the Go domain by reducing the number of visited nodes to a quarter compared to standard *PNS*. This section proposes two means for extending *MCPNS*. Both means target at increasing the speed of *MCPNS* even further.

Subsection 3.1 describes tuning of parameters beyond the previously documented scope, Subsection 3.2 outlines how to apply domain knowledge by small patterns.

3.1 Tuning Standard Parameters

In previous research [12], the authors tested the impact of several parameters on *MCPNS*. It was found that the speed of *MCPNS* is controlled mainly by two parameters, namely: (1) the number N of random sequences played at each evaluation, and (2) the sequence length *la* (cf. Section 2.2). The ranges of the parameters which were tested are $N \in 3, 5, 10, 20$ and $la \in 3, 5, 10$. Within these boundaries, the solver was found to perform fastest with smallest N and largest *la* yielding an increase of speed by a factor of 2 in comparison with standard *PNS*.

This result proposes that small N and large *la* are a good choice for speeding up the algorithm but it leaves unanswered which parameter setting provides for an optimal speed performance. The number of sequences N cannot grow arbitrarily small without impeding the search speed at some point, because it is bounded by $N = 0$ (representing the stage at which no evaluation takes place). Similarly, *la* must be limited by an upper bound for achieving less run time, because playing longer sequences requires additional evaluation time.

It is not so much of interest to record the strongly domain-dependent parameter setting of the described parameters *per se* but it is of interest to utilize an optimal parameter setting for estimating the algorithm's maximal speed enhancement.

Thus motivated, a parameter tuning was conducted with $N \in [1, 4]$ and $la \in [8, 25]$ measuring the speed and the number of expanded search tree nodes for this work. The algorithm was tested on a set of 30 Tsume Go problems as described in [12].

The optimal setting was found to be $N = 4$ and $la = 25$ yielding a 16% increase of speed (on each test case on average) compared to the best previously found setting. Omitting small statistical noise, a single rule characterizes the outcome in dependence of the two parameters within the tested limits. The rule is: the exploration effort is proportional to the time gain. Parameter settings with $N = 4$ produced the fastest results. The bigger the choice of *la* the faster the algorithm.

3.2 Implementing Pattern Knowledge

A common principle of extending heuristic search is the exploitation of domain-specific knowledge. This section investigates an instance of combining domain-specific knowledge in the form of patterns with *MCPNS*.

Patterns are a standard means for representing knowledge in the test domain of computer Go (e.g., [9], [3], [4]). A pattern in computer Go is a configuration of intersections. High-level representations contain features additional to the colouring of the intersections. Two examples are tactical information on connectivity and Life-and-Death status. The sizes of patterns vary considerably depending on the application's purpose. Large-scaled patterns are employed for openings, while tactical analysis is often guided by smaller patterns. Patterns can be hand-tuned or auto-generated ([3], [6], [16], [5]).

The patterns applied in the small experiment described in this section are auto-generated 3×3 patterns. They offer the advantage of low pattern-matching costs. This characteristic is a precondition to any application of *MCS* as required by *MCPNS* because it must be feasible to match each pattern for each move in each random sequence. The patterns are generated by statistical ranking in self play as described in [5]. Each of the 6,561 patterns is assigned with a numerical value representing the desirability of the move in the pattern's centre. The patterns do not account for information on the edge of the Go board. They were generated for the whole game and are not specifically tailored to Life-and-Death problems.

The patterns are employed to alter the probability distribution of the moves selected for the random sequence. Pure *MCPNS* plays randomly distributed legal moves in each random sequence implying a uniform distribution of the probability for selecting a move. This uniform distribution is altered by matching the patterns. Whenever a move is played in a random sequence on an intersection I, the pattern values of the effected intersections in the vicinity of I are updated. The neighbours' new pattern values are set according to the matching patterns. The probability to play at a neighbour is proportional to its pattern value. Thus, moves evaluated well by the patterns are more likely to be played than moves less promisingly assessed by the patterns.

In a small experiment for this work, the suggested patterns are found to slow down the proof procedure overall slightly by about 3%. In more detail, a third of the test problems are solved slightly faster with patterns than without while the majority is solved slightly slower. A positive effect of the procedure can be seen in a decrease in node expansions of about 6% on average per test case. This effect is small compared to the reduction of node expansions by 75% which *MCPNS* achieves on *PNS*.

4 Discussion

The results described in the previous section indicate that parameter tuning can improve *MCPNS* beyond previously documented limits while the straight forward pattern application does not yield an increase in efficiency.

The outcome of the tuning within the space of parameters that was open for investigation suggests two conclusions. Firstly, *MCPNS* yields better results with more frequent and deeper sampling given the constraints of the domain found in previous experiments. Secondly, the speed can still be improved by another 16% (on average per test case) in addition to the highest previously found speed.

The principal reason for the pattern matcher's disappointing performance is the negative trade-off between the time invested in pattern matching and the time gained by cutting off nodes. This negative trade-off in turn could be produced by different causes. Two candidates seem particularly likely: (1) too expansive pattern matching, and (2) too static evaluation. Too expansive pattern matching costs could be countered by training patterns specifically for Tsume-Go problems and by only matching the most promising patterns. A too static evaluation constitues a principle problem. Patterns alter the probability distribution of random moves statically, i.e., the evaluation is subject to an inherent static bias which is not altered in the cause of developing the game tree. In situations requiring untypical moves, the pattern matcher will suggest to play wrong moves (namely those which are *typically* good), because it does not take into account the search history or any other information derived from the search process. This effect must be expected to impact our choice of patterns particularly strongly because the patterns are very small and thus untypical moves will occur frequently. A mechanism for detecting untypical sequences might change this unwanted effect and render matching more dynamical.

5 Conclusions and Future Research

This contribution has provided an overview of the working of *MCPNS*. It has proposed and examined two means of extending the algorithm: (1) tuning of standard parameters, and (2) inclusion of pattern knowledge. We found experimentally that tuning could further increase *MCPNS's* speed slightly beyond what had previously been documented resulting in an overall reduction of the time cost to 42% of the time cost of *PNS* on the test set. The pattern approach chosen for the current work so far did not lead to an increase in the algorithm's efficiency.

We therefore may tentatively conclude that simple pattern matching does not seem to be sufficient to increase the speed of *MCPNS*. It may be characterized as possible reason that that pattern matching is too slow and too static.

The next research steps will elaborate on three points: (1) comparing the results with existing work on initialization in *PNS*, (2) assessing the possibility of improving the use of patterns, and (3) applying *MCS* to *df-pn*.

Ad (1). Nagai and Imai [11] propose an enhancement of df-pn (df-pn+) including heuristic initialization of proof and disproof numbers and reports that the number of nodes visited is thereby reduced by a factor of 1/6 in Othello endgames. Kishimoto [8] showed that the total time and the number of nodes expanded are reduced by a factor of 1/2 by using heuristic initialization in Tsume-Go problems. While the first of these two contributions is difficult to compare with this work because it is applied in a different domain, the second work described by Kishimoto [8] seems somewhat comparable. However, the test sets are different. Future work needs to account for a direct comparison of the different methods and describe if and how the two approaches mentioned can be combined with the current framework.

Ad (2). To improve the pattern approach, we suggest to reduce the number of patterns by a selection mechanism and to elaborate on a less static evaluation.

Ad (3). In practical applications, *PNS* is usually subject to time constraints and thus replaced by faster variations. The arguably most prominent representative of practical *PNS* variations is *df-pn* proposed by [10]. This variation offers two advantages. Firstly, it reduces the memory cost by applying depth-first search (but it requires a large hash table). Secondly, it reduces the time cost by avoiding unnecessary traversals. As described above, *MCPNS* uses heuristic knowledge to initialize the proof numbers and achieves a reduction of node expansions. Because both *MCPNS* and *df-pn* differ without interfering, they can be combined easily. Future work will therefore investigate the integration of *MCS* and *df-pn* and address the research question: What is the speed gain of *MCS* in *df-pn* compared to the speed gain of *MCS* in *MCPNS*?

Acknowledgements

We would like to thank Bruno Bouzy for his willingness to share the patterns developed in co-operation with him. This work is financed by the Dutch Organization for Scientific Research NWO as part of the project Go for Go, grant number 612.066.409

References

[1] Bruce Abramson. Expected-Outcome: A General Model of Static Evaluation. In *IEEE Transaction on PAMI*, volume 12, pages 182–193, 1990.

[2] L. Victor Allis, Maarten van der Meulen, and H. Jaap van den Herik. Proof-Number Search. *Artificial Intelligence*, 66(1):91–124, 1994.

[3] Bruno Bouzy. Go Patterns Generated by Retrograde Analysis. In Jos W.H.M. Uiterwijk, editor, *The CMG Sixth Computer Olympiad. Computer-Games Workshop. Technical Reports in Computer Science, CS 01-04. IKAT, Department of Computer Science, Universiteit Maastricht, Maastricht*, 2001.

[4] Bruno Bouzy and Tristan Cazenave. Computer Go: An AI Oriented Survey. *Artificial Intelligence*, 132(1):39–103, 2001.

[5] Bruno Bouzy and Guillaume Chaslot. Monte-Carlo Go Reinforcement Learning Experiments. In G. Kendall and S. Louis, editors, *IEEE 2006 Symposium on Computational Intelligence in Games, Reno, USA*, 2006. 8 pages (in print).

[6] Thore Greapel, Mike Goutrie, Marco Krüger, and Ralf Herbich. Learning on Graphs in the Game of Go. In Georg Dorffner, Horst Bischof, and Kurt Hornik, editors, *ICANN. Vienna, Austria.*, volume 2130 of *Lecture Notes in Computer Science*, pages 21–25. Springer, 2001.

[7] Tomoyuki Kaneko, Tetsuro Tanaka, Kazunori Yamaguchi, and Satoru Kawai. Df-pn with Fixed-depth Search at Frontier Nodes. In Hitoshi Matsubara, editor, *10th Game Programming Workshop in Japan*, pages 1–8, 2005. Hakone, Japan. In Japanese, abstract in English.

[8] Akihiro Kishimoto. *Correct and Efficient Search Algorithms in the Presence of Repetitions*. PhD thesis, University of Alberta. Edmonton, Canada, 2005.

[9] Takuya Kojima and Atsushi Yoshikawa. Knowledge acquisition from game records. In Johannes Fürnkranz and Miroslav Kubat, editors, *Workshop Notes: Machine Learning in Game Playing. 16th International Conference on Machine Learning. Bled, Slovania.* 1999.

[10] Ayumu Nagai. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. PhD thesis, University of Tokyo. Tokyo, Japan, 2002.

[11] Ayumu Nagai and Hiroshi Imai. Application of df-pn+ to Othello Endgame. In Hitoshi Matsubara, editor, *Game Programming Workshop in Japan '96*, pages 16–23, 1999.

[12] Jahn-Takeshi Saito, Guillaume Chaslot, Jos W.M.H. Uiterwijk, and H. Jaap van den Herik. Monte-Carlo Proof-Number Search for Computer Go. In H. Jaap van den Herik, Paolo Ciancarini, and Jeroen Donkers, editors, *Computers and Games 2006*. International Computer Games Association, 2006. 12 pages (in print).

[13] Masahiro Seo, Hiroyuki Iida, and Jos W.H.M. Uiterwijk. The PN*-Search Algorithm: Application to Tsume-Shogi. *Artificial Intelligence*, 129(1-2):253–277, 2001.

[14] Seiichi Tanaka, Iida Hiroyuki, and Yoshiyuki Kotani. An Approach to Tsume-Shogi: Applying Proof-Number Search with Estimation Function of Mating. In Hitoshi Matsubara, editor, *Game Programming Workshop in Japan '95*, pages 138–147, Kanagawa, Japan, 1995. In Japanese, abstract in English.

[15] Seiichi Tanaka and Yoshiyuki Kotani. Checkmate Search with Checkmate Estimation Function. In Hitoshi Matsubara, editor, *Game Programming Workshop in Japan '96*, pages 141–149, Kanagawa, Japan, 1996. In Japanese, abstract in English.

[16] Erik C. D. van der Werf. *AI Techniques for the Game of Go*. PhD thesis, Universiteit Maastricht. Maastricht, The Netherlands, 2004.