Monte-Carlo Proof-Number Search for Computer Go

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

MICC-IKAT,
Maastricht University, Maastricht, The Netherlands
{j.saito,g.chaslot,uiterwijk,herik}@micc.unimaas.nl

Abstract. In the last decade, proof-number search and Monte-Carlo methods have successfully been applied to the combinatorial-games domain. Proof-number search is a reliable algorithm. It requires a well defined goal to prove. This can be seen as a disadvantage. In contrast to proof-number search, Monte-Carlo evaluation is a flexible stochastic evaluation for game-tree search. In order to improve the efficiency of proof-number search, we introduce a new algorithm, Monte-Carlo Proof-Number search. It enhances proof-number search by adding the flexible Monte-Carlo evaluation. We present the new algorithm and evaluate it on a sub-problem of Go, the Life-and-Death problem. The results show a clear improvement in time efficiency and memory usage: the test problems are solved two times faster and four times less nodes are expanded on average. Future work will assess possibilities to extend this method to other enhanced Proof-Number techniques.

1 Introduction

Since 1994, proof-number search (PNS) has enjoyed wide acceptance in the combinatorial-games domain (cf. Sect. 1.1). The advantage of PNS is its effectiveness and speed. Given enough time and space, it finds a proof or disproof of a binary goal in a game tree. Its disadvantage lies in the requirement for a testable binary goal. In contrast, Monte-Carlo evaluations of combinatorial game positions, introduced in 1993 (cf. Sect. 1.2), offer the advantage of flexibility. A Monte-Carlo evaluation may change its goal because it looks for scores.

This article describes a new algorithm, Monte-Carlo Proof-Number search. It successfully exploits the flexibility of Monte-Carlo evaluation and so it improves the efficiency of the reliable proof-number search.

The remainder of this section outlines the two established techniques, presents the motivation for the new algorithm, and situates the work in the context of proof-number search and Monte-Carlo evaluation. Section 2 offers a detailed description of the new algorithm. Section 3 describes an experiment applying the algorithm to the Life-and-Death problem in Go. Section 4 presents experimental results. Section 5 discusses the findings. Section 6 provides a conclusion and an outlook on future research.

H.J. van den Herik et al. (Eds.): CG 2006, LNCS 4630, pp. 50–61, 2007.

1.1 Proof-Number Search

Proof-number search (PNS) mechanisms are nowadays accepted as standard means of game-tree search. They were introduced by [1] and since then developed into a whole family of search algorithms (e.g., [5,10,15]) with applications to many combinatorial games such as Shogi [12], the one-eye problem in Go [9], and checkers [11]. PNS is a heuristic search applied to game trees to prove or disprove a goal. The status of the goal has to be determined as reachable or non-reachable.[1] The core idea of standard PNS is to order branches efficiently. To that end, the expansion mechanism prefers nodes which require the least number of estimated further expansions for proving or disproving the goal. To achieve such an ordering, the algorithm performs a best-first search strategy based on two figures in each node. One of the numbers is the *proof number*. It maintains the minimal number of successor nodes that require expansion to prove the goal. Analogously, the *disproof number* represents the minimal number of successors necessary to disprove the goal. Leaf nodes are evaluated, interior nodes receive their respective proof and disproof numbers by bottom-up back-propagation. The heuristic exploits the AND/OR tree characteristic that proving the goal requires only to prove the single best OR branch.

1.2 Monte-Carlo Evaluation

Extensive investigation has been conducted on the Monte-Carlo method within the field of Go. The first occurrence of Monte-Carlo (MC) evaluation for the game of Go emerged more than a decade ago [6]. Since then, MC evaluation has been attracting attention in the field of computer Go, particularly in recent years [2,3,4,7,13]. The core idea of the MC approach is to evaluate a game position statistically. This is achieved by playing a number of randomly generated games. The results are stored and then mapped by statistical evaluation to a single value, e.g., by calculating the mean of the scores of all games. In this paper, we call the mapping that achieves the statistical evaluation on the games' values the *accumulative function*. Each random game we call a *simulated game*. A *simulated move* is a move played in a simulated game. A defining characteristic of the MC evaluation lies in its scarce requirement for domain-specific knowledge as the amount of knowledge necessary for playing random games and for scoring finished games suffices.

1.3 Integrating MC and PNS

For integrating MC evaluation and PNS two approaches are possible. First, a global MC move-selection framework might take advantage of local PNS. In this case a top-level co-ordination mechanism needs to determine when to shift from MC evaluation to local PNS. Second, joining MC evaluation with PNS for solving local tactical search may be advantageous. In this article, we address only the latter approach.

[1] One variation of the algorithm also deals with the *unknown* status [8, Section 2.1.8].

2 MC-PNS

Below, we propose a new algorithm, Monte-Carlo Proof-Number Search (MC-PNS). It extends PNS' best-first heuristic by adding an MC evaluation. The aim is to achieve a better ordering of moves and thereby to omit investigating branches which are not promising according to the MC evaluation.

2.1 Algorithm

MC-PNS performs a best-first search in an AND/OR game tree. The search aims at proving or disproving a binary goal, i.e., a goal that can be reached by player MAX or be refuted by player MIN under optimal play by both sides. Each node n in the tree contains two real-valued numbers called the proof number ($pn(n)$) and the disproof number ($dn(n)$), respectively.

MC-PNS performs a two-step cycle fully identical to that of PNS. For reasons of readability we summarize the two-step cycle here. In the first step, the best-first strategy requires the algorithm to traverse down the tree starting at the root guided by the smallest proof or disproof number until leaves are reached and expanded. The second step of the cycle begins after that expansion of the leaf. Now, the newly assigned proof and disproof numbers are propagated back to the root updating the proof and disproof number in each node throughout the path back to the root. The cycle is complete as soon as the root has been reached and its values are updated. The cycle is repeated until the termination criterion is met. The criterion is satisfied exactly if either the root's proof number is 0 and the disproof number is infinity, or vice versa. In the first case, the goal is proven. In the latter instance it is refuted. Still, there is a difference between PNS and MC-PNS. The next paragraph outlines the details of the algorithm more formally and shows the small differences.[2]

Let l be a leaf node. If l is a node proving the goal then $pn(l) = 0$ and $dn(l) = \infty$ holds. If l is a node disproving the goal then $pn(l) = \infty$ and $dn(l) = 0$ holds. If l does not immediately prove or disprove a goal $pn(l) = pmc(l)$ and $dn(l) = dmc(l)$, where pmc and dmc are the Monte-Carlo based evaluation functions mapping a node to a target domain. This domain could be $(0, 1)$, with pmc reflecting a node's estimated probability to reach the goal, and dmc reflecting a node's expected probability not to reach a goal. The value 0 is excluded to avoid incorrectly setting $pn = 0$ or $dn = 0$ which could exclude the node from further exploration falsely.

The pmc and dmc numbers for a position with N simulated games are gained by calculating the evaluation function $eval_N : \{0, ..., N\} \rightarrow (0, 1)$. The function $eval_N$ depends on the number N_+ of simulated games in which the goal was reached and the number N_- of simulated games in which the goal was not reached. Since $N = N_+ + N_-$, $eval_N$ depends on N. We let $eval_N(0) = \epsilon$ for

[2] The two-step cycle can be optimised by not fully back-propagating the values when this is not required. Thereby, the cost of traversal can be reduced. The implementation used in the experiments described here does not include this optimisation.

some small positive real number $\epsilon < 1$ and $eval_N(N_+) = N_+/(N+1)$ for $N_+ \neq 0$. We set $dmc = eval_N(N_+)$ and $pmc = 1 - dmc$.

Starting at the leaf node expanded last, *pn* and *dn* values are back-propagated by PNS' back-propagation rule (Fig. 1). Note that MC-PNS assigns a new ordering to the branches.

Rules for *AND* nodes:	Rules for *OR* nodes:
$pn(n) = \sum_{s \in successor(n)} pn(s)$	$pn(n) = \min_{s \in successor(N)} (pn(s))$
$dn(n) = \min_{s \in successor(n)} (pn(s))$	$dn(n) = \sum_{s \in successor(N)} dn(s)$

Fig. 1. Rules for updating proof and disproof numbers, respectively

2.2 Controlling Parameters

MC-PNS imposes an evaluation cost for each expansion of a node. The algorithm aims at achieving a better move ordering as a trade-off. Two extreme approaches can be distinguished: (1) MC-PNS spends hardly any time on evaluation, and (2) MC-PNS takes plenty time on evaluation. Below, three control parameters are introduced which will enable a trade-off between these extremes.

Number of MC evaluations per node. The precision of the MC evaluation can be determined by the number of simulated games at each evaluated node. Henceforth, this number will be denoted by N.

Look-ahead for MC evaluation. The look-ahead (la) is the maximal length of a simulated game.

Launching level of the MC evaluation. Given a limited look-ahead, it is not useful to waste time on evaluating close to the root of the search-tree. The nodes close to the root must be expected to be expanded anyway. The launching level (*depth*) is the minimal required level of the tree at which nodes are evaluated.

3 Experimental Application of MC-PNS to Go

This section describes an experimental comparison of PNS and the MC-PNS variations introduced above (Subsection 2.2). The experiment is conducted by applying the algorithms to a test set of Life-and-Death problems. Subsection 3.1 outlines the experiment's application domain, Go. Subsection 3.2 poses research questions to be tested. Subsection 3.3 provides an account of the experimental setup.

3.1 Application to the Go Domain

We applied MC-PNS for solving instances of the Life-and-Death problem which is a frequently occurring sub-problem in Go games. Search for solving the Life-and-Death problem has been addressed before (e.g., [16]). The general Life-and-Death problem consists of a locally bounded game position with a target group of stones. Black moves first and has to determine the group's status as either alive, dead, ko, or seki. For the current investigation, however, the problem is reduced to a binary classification of the target group to either alive or dead.

In order to fit the algorithm to the Go domain, the goal to prove, i.e., the status of a group of stones as either alive or dead, is checked by a simple status-detection function. This function is called by each simulated move to determine whether to play further moves or stop expanding. Each simulated game halts after either the goal has been met or a certain previously determined number of moves has been played.

3.2 Research Questions

The experiment is conducted in order to test the viability of the MC-PNS algorithm and its variations created by the parameter settings. The experiment should answer which, if any, of the MC-PNS settings can be of practical use or may lead to enhanced algorithms. To account for the tested expectations concisely, we formulate a list of five operational research questions which should be answered by the experiment. The questions are ordered descendingly by the expected likelihood of being answered positively. The first question is therefore the weakest and the last one the strongest.

1. Can assigning a heuristic weighting in the expansion of PNS achieve a more efficient move ordering?
2. Can a MC-PNS variation achieve such an ordering?
3. Do the number of evaluations per node, the look-ahead, and the launching level trade off between the run-time characteristics of PNS and MC-PNS?
4. Do they thereby contribute to establishing a trade-off for practical work?
5. Does MC-PNS work more efficiently with respect to time and space complexity than PNS?

3.3 Setup

In order to describe the experimental setup, this subsection specifies the Life-and-Death test set, algorithmic details, and the test procedure.

Life-and-Death test set. The test set consists of 30 Life-and-Death problems. The game problems have a beginner and intermediate level (10 Kyu to 1 Dan) and are taken from a set publicly available at GoBase [14]. All test problems can be proven to be advantageous for player Black. For each case there exists a best first move. The test cases were annotated with marks for playable intersections and marks for the groups subject to the alive-or-dead classification. The number

of intersections I becoming playable during search is determined by the initial empty intersections and by intersections which are initially occupied but become playable because the occupying stones are captured. For the test cases this indicator I of the search space varied from 8 to 20. The factorial of I provides a rough lower bound for the number of nodes in the fully expanded search tree. Thereby, it provides an estimate for the size of the search space.

Algorithmic and implementational details. PNS and various parameter settings for MC-PNS were implemented in a C++ framework. All experiments were conducted on a Linux workstation with AMD Opteron architecture and 2.8 GHz clock rate. 32 GB of working memory were available. No special pattern matcher or other feature detector was implemented to enhance the MC evaluation or detect the Life-and-Death status of a position. Instead, tree search and MC evaluation are carried out in a brute-force manner in the implementation. In the experiment, Zobrist hashing [17] was implemented to store proof and disproof numbers estimated by MC evaluation. In order to save memory the game boards are not stored in memory for any leaf. A complete game is played for each cycle. The proof tree is stored completely in memory.

The MC evaluation used was based on our Go program MANGO. The program's MC engine provides a speed of up to 5,000 games of 19×19 Go of an average length of 120 moves per second on the hardware specified above. The MC engine was not specially designed for the local Life-and-Death task. Its speed and memory consumption must therefore be expected to perform sub-optimally. The MC evaluation was introduced in Subsection 2.1 as $eval_N$.

Test procedure. Three independent parameters and three dependent variables describe the experiment. The independent parameters control the amount and manner of MC evaluation during search. The dependent variables measure the time and memory resources required to solve a problem by a specified configuration of the algorithm. The configuration is synonymously called parameter setting.

The independent parameters available for testing are: (1) the number of simulated games per evaluated node ($N \in \mathbb{N}$), (2) the look ahead ($la \in \mathbb{N}$), and (3) the launching depth level for MC evaluations ($depth \in \mathbb{N}$).

There are three dependent test variables measured: (1) the time spent for solving a problem measured in milliseconds ($time \in Time = \mathbb{R}$), (2) the number of nodes expanded during search ($nodes \in Nodes = \mathbb{N}$), and (3) a move ($move \in Moves = \{0, 1, ..., 361\}$) proven to reach the goal.[3] The set $Moves$ represents the range of possible intersections of the 19×19 Go board together with an additional null move (0). The null move is returned if no answer can be found in the time provided (see below). Because no other dynamic-memory cost is imposed, *nodes* suffices for calculating the amount of memory occupied. The memory consumption of a single node in the search tree is 288 bytes.

For the experiment, each configuration consisted of a triple of preset parameters $\in (N, la, depth)$. The following parameter ranges were applied: $N \in$

[3] For three test problems, some proof systems found a forcing move first, i.e., there were two best first moves.

$\{3, 5, 10, 20\}$, $la \in \{3, 5, 10\}$, $depth \in \{I, \frac{1}{2}I, \frac{3}{4}I\}$. The *depth* parameter requires additional explanation. In the experiment's implementation, the number of initially empty intersections I is employed to calculate a heuristical *depth* value. Thus $I, \frac{1}{2}I$, and $\frac{3}{4}I$ represent functions dependent on the specific instance of I given by each test case. We will call these choices of I the starting strategies and refer to them as 1, 2, and 3 for I, $\frac{1}{2}I$, and $\frac{3}{4}I$, respectively.

The outcome of an experiment is a triple of measured variables $\in Time \times Nodes \times Moves$. Each experimental record consists of a configuration, a problem it is applied to, and an outcome. An experiment delivers a set of such records. In order to account for the randomness of the MC evaluations and potential inaccuracy for measuring the small time spans well below a 10th of a second, PNS and each configuration of MC-PNS was applied to each test case 20 times resulting in 20 records.

4 Results

This section outlines the results of the experiment. First, a statistical measure for comparing the algorithms is introduced, then the experimental results for different configurations are presented and described in detail. The section concludes by summarizing the results in six propositions.

The experiment consumed about 21 hours and produced 22,200 records. All solutions by PNS as well as by MC-PNS configurations were correct.

Aggregates for each combination of test cases and configurations are considered in order to enable a comparison of different parameter settings. For a parameter setting $p = (N, la, depth)$ and a test case ϑ the average result is the triple $(t, s, m) \in Time \times Nodes \times Moves$. For this triple t, s, and m are averaged over all 20 records with the parameter p applied to the test case ϑ. $t_{(p,\vartheta)}$ is the average time and $s_{(p,\vartheta)}$ the average number of nodes expanded for solving ϑ. In order to make a parameter setting p comparable, its average result is compared to average result of PNS. We define the gain of time as $gain_{time}(p) = \frac{1}{30} \sum_{\vartheta=1}^{30} t_{(pns,\vartheta)}/t_{(p,\vartheta)}$. (Here, $t_{(pns,\vartheta)}$ is the average time consumed by PNS to solve test case ϑ.) The gain of space is defined analogously. The positive real numbers $gain_{time}$ and $gain_{space}$ express the average gain of a parameter setting for all thirty test cases. Each gain value is a factor relative to the performance of the PNS benchmark.

In the experiment the configuration using 3 MC evaluations per node, a look-ahead of 10 moves, and starting strategy 3, is found to be the fastest configuration ($p_{fast} = (3, 10, 3)$). The $gain_{time}(p_{fast}) = 2.05$ indicating that it is about twice as fast as the PNS benchmark (see Table 1, left and the definition of the gain of speed and gain of space). The $gain_{space}(p_{fast}) = 4.26$. Thus p_{fast} expands less than a quarter of nodes which PNS expands. The parameter setting $p_{narrow} = (20, 10, 3)$ is expanding the smallest number of nodes. It requires less than a fifth of the expansions compared to the benchmark on average on the test set. It is slightly slower than PNS in spite of that (see Table 1, right).

Table 1. Time and node consumption of PNS and various configurations of MC-PNS relative to PNS. Each table presents the ten best ranked settings of the 36 parameter settings. Left: ordered by a factor representing the gain of speed. Right: ordered by a factor representing the decrease of nodes.

Rank	N	*la*	*depth*	$gain_{time}$	$gain_{space}$
1	3	10	3	2.05	4.26
2	5	10	3	1.93	4.54
3	3	10	2	1.87	3.90
4	5	10	1	1.80	4.59
5	3	10	1	1.75	4.30
6	5	10	2	1.74	4.11
7	3	5	3	1.56	2.70
8	3	5	1	1.51	2.70
9	10	10	3	1.42	4.99
10	10	10	1	1.36	4.95

Rank	N	*la*	*depth*	$gain_{space}$	$gain_{time}$
1	20	10	3	5.31	0.95
2	20	10	1	5.23	0.96
3	10	10	3	4.99	1.42
4	10	10	1	4.95	1.36
5	5	10	1	4.59	1.80
6	20	10	2	4.56	0.87
7	5	10	3	4.54	1.93
8	3	10	1	4.30	1.75
9	10	10	2	4.28	1.26
10	3	10	3	4.26	2.05

Parameter settings with large N and large look-ahead require the least number of nodes to prove or disprove the goal. Parameter settings with small N but large look-ahead perform fastest.

So far, this subsection focused on outlining the results relevant for characterizing the set consisting of PNS and MC-PNS variations. The remainder of this subsection describes the results required for comparing p_{fast}, p_{narrow}, and PNS in greater detail. For this purpose, the data hidden in the aggregates of test cases are unfolded. This is achieved by comparing the average time and space performance for each test case. Figures 2 and 3 illustrate this comparison.

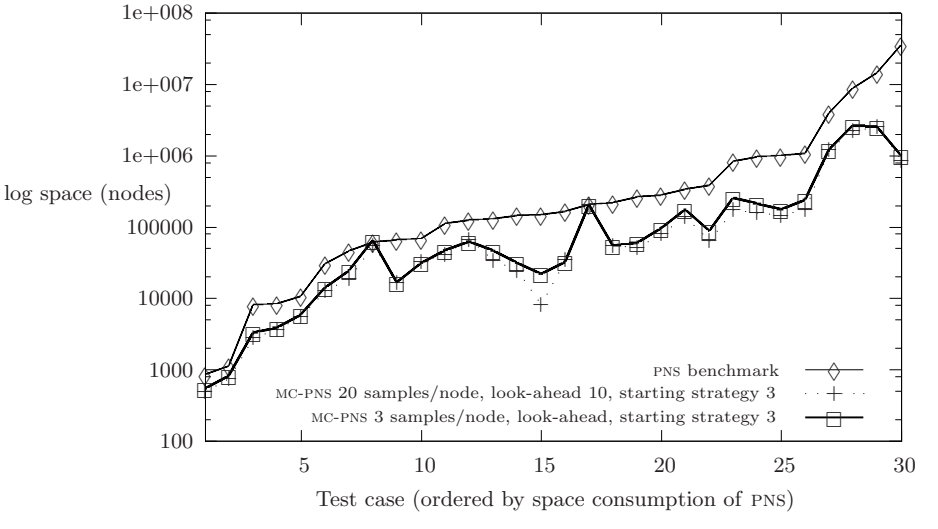

Fig. 2. Space complexity of various configurations

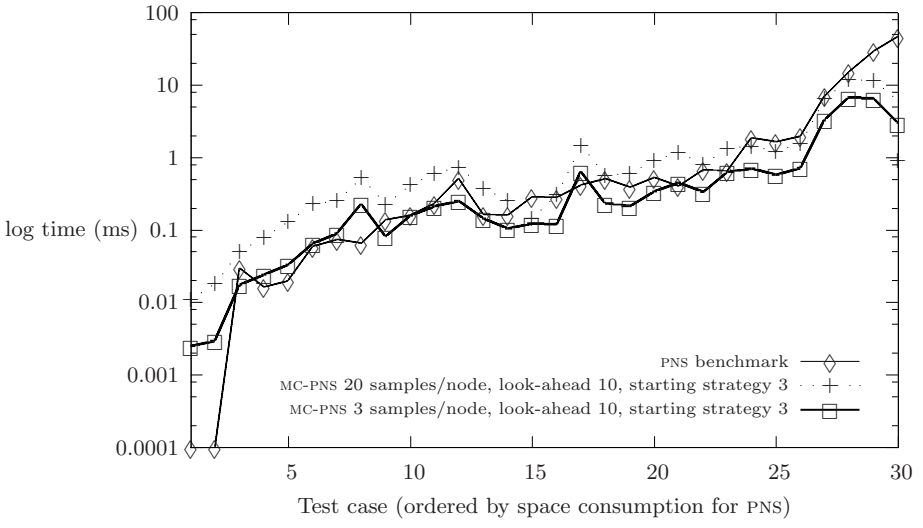

Fig. 3. Time complexity of various configurations

A comparison of the space behaviour (Figure 2) shows that PNS requires the highest number of node expansions to prove or disprove a goal irrespectively of the test case. Its memory requirements are roughly the same as those of p_{fast} and p_{narrow} except for two test cases. PNS requires much more space than the two MC-PNS variations on virtually all other test cases. p_{fast} and p_{narrow} show a similar behaviour in memory consumption with p_{narrow} performing slightly more efficient.

A comparison of the time behaviour (Figure 3) shows a pattern containing more variety. Overall, p_{narrow} is characterized by the least time efficient results: in 22 of the 30 test cases it is the slowest of the three compared proof systems. PNS performs slowest on the remaining 8 test problems. But PNS also performs as single fastest solver in 6 cases and as fast as p_{fast} in 6 cases. p_{fast} shows the most efficient time consumption in 25 cases including the 6 cases in which it is as fast as PNS.

The results show that the performance depends on the complexity inherent to the tested problem. PNS finds its proofs faster than the other problem solvers on simple problems, i.e., problems requiring fewer than 5,000 nodes to be solved. It outperforms its competitors only once in the 20 more complex tasks while achieving this five times in the 10 least complex problems. The two MC-PNS variations perform comparably faster on the 20 most complex tasks. The experimental outcome shows that the speed advantage of p_{fast} relative to PNS grows with the complexity of the tested problem.

The main results of this section can be summarized in six propositions.

(1) (3,10,3) is the fastest parameter configuration. On average it performs two times faster than PNS on the test set and expands less than a quarter of the nodes.

(2) The configuration (20,10,3) is the MC-PNS configuration with the least node expansions on average. It expands only a fifth of the number of nodes expanded by PNS.
(3) p_{fast} and p_{narrow} reliably expand considerably fewer nodes than MC-PNS.
(4) p_{fast} performs reliably faster than MC-PNS.
(5) The advantage of time performance of p_{fast} relative to PNS even grows with the complexity of the problem.
(6) PNS performs better than p_{fast} on problems with small complexity.

5 Discussion

This section discusses the results outlined in the previous section. First, explanations for the results experimentally gained are offered. A critical remark on the suitability of the setup concludes this section.

On average p_{fast} solves problems twice as fast as PNS. This is coherent with a general tendency observed. As outlined above, settings with small N (i.e., few simulated games) and large la (i.e., far look-ahead) are generally the fastest. The speed of a MC-PNS depends mainly on two items: (1) the amount of nodes it expands, and (2) the speed needed to evaluate a node. These two items are mutually dependent. The number of expansions decreases with the intensity of evaluation because the heuristic is more reliable and prunes more nodes. This is reflected by the experimental finding that nodes with thorough evaluation (large N and large la) reach their goals with few expansions (see Table 1). But more intensive evaluations require more time. Therefore, an optimization problem has to be solved to compensate for the intensity of the evaluation. The optimum trades off between the number of nodes visited and the evaluation time for each single node. This optimum is found to be $p_{fast} = (3, 10, 3)$. Extensively evaluating each node, as pursued by p_{narrow}, devotes too much time on each single evaluation. The strategy of omitting a heuristic evaluation entirely, as embodied by PNS, is cheap on each node but fails to prune the tree sufficiently on a global scale.

Thus the N and la parameters can be said to control the intensity of each evaluation successfully. The *depth* parameter has a minor impact on controlling this intensity. More importantly, only few MC evaluations per node can yield a reasonable heuristic for the MC-PNS framework.

The design of the PNS framework requires running up and down the branch leading to the currently expanded node (see Sect. 2.1). The deeper the tree grows the higher are the costs for this traversal. The cost grows linearly with the tree's depth as more nodes are expanded. Additionally, the traversal takes longer. Therefore, pruning the tree will have a much stronger impact on larger trees. The point at which pruning pays off is reached at a search tree complexity equivalent to a size which requires about 5,000 nodes to be solved by PNS. Any problem beyond this threshold of complexity must be expected to be solved faster by p_{fast}. Two thirds of the test cases employed lie beyond that threshold. One might argue that this composition of the set is arbitrary. But there are two strong reasons for assuming that the inferences made about the quality of p_{fast} are valid in general.

First, the test cases chosen are rather easy. One may therefore expect that real-life problems are harder than the cases presented. Thus in practice p_{fast} should be more relevant. Second, the absolute time saved by p_{fast} is much larger for complex problems. For instance, PNS requires 47.7 sec to solve the most complex problem whereas p_{fast} solves the problem in less than 6 sec (eight times faster). The absolute time saved is crucial for real-life applications, e.g., in a Go program. Thus we may conclude that it is valid to generalize our finding that p_{fast} is performing faster than PNS beyond our test set.

6 Conclusion and Outlook

We introduced a new algorithm, MC-PNS, based on MC evaluation within a PNS framework. An experimental application of the new algorithm and several variations to the Life-and-Death sub-problem of Go were described; moreover, its interpretation was presented. It was demonstrated experimentally that given the right setting of parameters MC-PNS will outperform PNS. For such a configuration MC-PNS will be two times faster than PNS on average and use less than a quarter of the node expansions. Section 5 presented strong evidence for assuming that this result will be generalized beyond the test cases observed.

Thus, we may conclude that all research questions posed in Subsection 3.2 can be answered positively on the basis of the experimental findings.

Future work will be required to assess the possibility of successfully extending the MC-PNS approach to the practically more relevant Depth-first Proof-Number Search (DF-PN, [10]). Its characteristics are slightly different but we believe that efficiency of DF-PN can be improved with our approach, too. Furthermore, detailed tuning of the algorithm and including more domain knowledge by applying patterns remain to be tested.

Acknowledgments

We would like to thank Mark Winands, Ulaş Türkmen, Benjamin Torben-Nielsen, Steven de Jong, and Jeroen Donkers for their support. The comments by the anonymous referees are gratefully acknowledged. In particular, we would like to thank one referee for his or her constructive criticism which led to crucial improvements of the implementational work. Without this feedback the current results would not have been possible. The work is financed by the Dutch Organization for Scientific Research in the framework of the project GO FOR GO, grant number 612.066.409.

References

1. Allis, L.V., van der Meulen, M., van den Herik, H.J.: Proof-Number Search. Artificial Intelligence 66, 91–124 (1994)
2. Bouzy, B., Helmstetter, B.: Monte Carlo Developments. In: van den Herik, H.J., Iida, H., Heinz, E.A. (eds.) 10th Advances in Computer Games (ACG10), Many Games, Many Challenges, pp. 159–174. Kluwer Academic Publishers, Dordrecht (2004)

3. Bouzy, B.: Associating Shallow and Selective Global Tree Search with Monte Carlo for 9×9 Go. In: van den Herik, H.J., Björnsson, Y., Netanyahu, N.S. (eds.) CG 2004. LNCS, vol. 3846, pp. 76–80. Springer, Heidelberg (2006)
4. Bouzy, B.: History and Territory Heuristics for Monte-Carlo Go. In: Chen, K., et al. (eds.) Joint Conference on Information Sciences JCIS 2005, p. 4 (2005)
5. Breuker, D.M.: Memory versus Search. PhD thesis, Maastricht University (1998)
6. Brügmann, B.: Monte Carlo Go. White paper (1993)
7. Cazenave, T., Helmstetter, B.: Search for Transitive Connection. Information Sciences 132(1), 93–103 (2004)
8. Kishimoto, A.: Correct and Efficient Search Algorithms in the Presence of Repetitions. PhD thesis, University of Alberta (2005)
9. Kishimoto, A., Müller, M.: DF-PN in Go: Application to the One-Eye Problem. In: van den Herik, H.J., Iida, H., Heinz, E.A. (eds.) 10th Advances in Computer Games (ACG10), Many Games, Many Challenges, pp. 125–141. Kluwer Academic Publishers, Dordrecht (2003)
10. Nagai, A.: Df-pn Algorithm for Searching AND/OR Trees and Its Applications. PhD thesis, University of Tokio (2002)
11. Schaeffer, J., Björnsson, Y., Burch, N., Kishimoto, A., Muller, M., Lake, R., Lu, P., Sutphen, S.: Solving Checkers. In: International Joint Conference on Artificial Intelligence (IJCAI), pp. 292–297 (2005)
12. Seo, M., Iida, H., Uiterwijk, J.W.H.M.: The PN*-Search Algorithm: Application to Tsume-Shogi. Artificial Intelligence 129(1-2), 253–277 (2001)
13. Sheppard, B.: Efficient Control of Selective Simulations. ICGA Journal 27(3), 67–80 (2005)
14. van der Steen, J.: GoBase.org website (2006), `http://www.gobase.org`
15. Winands, M.H.M., Uiterwijk, J.W.H.M., van den Herik, H.J.: An Effective Two-Level Proof-Number Search Algorithm. Theoretical Computer Science 313(3), 511–525 (2004)
16. Wolf, Th.: Forward Pruning and Other Heuristic Search Techniques in Tsume Go. Information Sciences 122(1), 59–76 (2000)
17. Zobrist, A.L.: A New Hashing Method with Application for Game Playing. ICCA Journal 13(2), 69–73 (1990)