A Comparison of Monte-Carlo Methods for Phantom Go

Joris Borsboom Jahn-Takeshi Saito Guillaume Chaslot
Jos W.H.M. Uiterwijk

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

Abstract

Throughout recent years, Monte-Carlo methods have considerably improved computer Go programs. In particular, Monte-Carlo Tree Search algorithms such as UCT have enabled significant advances in this domain. Phantom Go is a variant of Go which is complicated by the condition of imperfect information. This article compares four Monte-Carlo methods for Phantom Go in a self-play experiment: (1) Monte-Carlo evaluation with standard sampling, (2) Monte-Carlo evaluation with all-as-first sampling, (3) UCT with late random opponent-move guessing heuristic, and (4) UCT with early probabilistic opponent-move guessing heuristic. Our experimental findings indicate that Monte-Carlo methods can be applied to Phantom Go effectively. Surprisingly, Monte-Carlo Tree Search performs comparable to Monte-Carlo evaluation but not much better.

1 Introduction

In recent years steady progress has been made in the domain of computer Go. This article examines how Monte-Carlo methods, which have proven to be successful in Go, can be applied to the game of Phantom Go. An experiment comparing four Monte-Carlo methods by self-play is presented.

This article is structured as follows. Section 2 provides an overview of AI research related to Phantom Go. Section 3 gives an outline of relevant Monte-Carlo methods and outlines known and novel ways of adopting these methods to Phantom Go. Section 4 describes the experiment. The results of this experiment are discussed in Section 5. Section 6 concludes this article.

The remainder of this section describes the game of Phantom Go.

1.1 Phantom Go

Go is a two-player perfect-information game of no chance. Two players, *Black* and *White*, place stones on the intersections of a square grid (the board) in alternating turns. Starting with an empty board, the board is gradually populated with more stones. The objective of the game is to occupy or encircle more territory, i.e., intersections of the grid, than the opponent. Stones of the same color can be connected along the lines of the grid. Connected stones are called groups. A player can capture the stones of a group belonging to the opponent by placing his own stones on every empty intersection neighbouring the opponent group. Captured stones are taken off the board.

Phantom Go is an imperfect-information variant of the game of Go. While in Go all information is fully available to both players at any time, in Phantom Go this condition of perfect-information is suspended: each player knows the position of the stones he has placed but the location of the opponent stones may be unknown to him. Phantom Go is played with an arbiter to whom all information is available. Three game boards are used: the arbiter's board reflects the actual game state. The players' boards, there is one for each player, represent the game state as known to the respective player. The arbiter's and the opponent board are hidden to the players.

Instead of placing stones directly on the board, a player attempts to make a move as follows. Player P announces his move decision to the arbiter only. The arbiter then checks whether the move is a legal Go move on the arbiter's board. In this situation one of two cases applies: (1) if

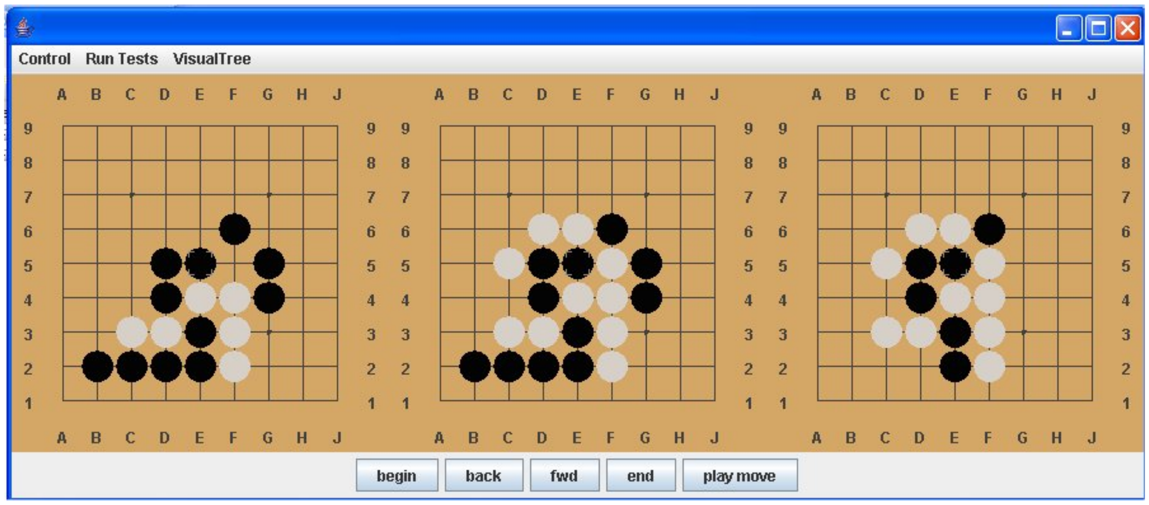

Figure 1: A game of Phantom Go with the three game boards: Black's board (left), the arbiter's board (center), and White's board (right). The screen shot is taken from our program InTheDark.

the move is found to be legal, the arbiter communicates to P that the move is legal. In case stones were captured, the number of stones captured is disclosed to P and his opponent. The arbiter plays the move on his board and P plays the move on his board. (2) If otherwise the move is illegal, the arbiter discloses this information to P. Player P then may again attempt to play a move.

Figure 1.1 depicts a game of Phantom Go with the three game boards. The center shows the arbiter's board with all stones played in the course of the game. The left board is Black's board and the right board is White's board. The latter boards show only those opponent stones whose location has been inferred already.

Discarding the perfect-information condition in Phantom Go results in a complication for game-tree search applicable to the game of Go. The reason for this complication lies in the growth of search space when going from Go to Phantom Go. This in turn is caused by the unknown position of opponent stones. Analogous growths of complexity can be observed when scaling from Chess to Kriegspiel.

2 Related work

This section describes AI approaches to Phantom Go and related work. Subsection 2.1 summarizes achievements made in the related field of Kriegspiel. Subsection 2.2 outlines the progress in Computer Go. Subsection 2.3 presents the findings of [5] which is the only available work on Phantom Go known to the authors.

2.1 Kriegspiel

It may be said by analogy that Phantom Go is to Go what the game of Kriegspiel is to Chess. Kriegspiel is an imperfect-information variant of Chess. Like Phantom Go it is played with an arbiter who oversees the actual game state and two players who can lack information about the respective opponent pieces. While there is only one publication on AI methods for Phantom Go according to the knowledge of the authors, Kriegspiel has so far inspired more game research.

Arguably, the most significant technique designed for Kriegspiel is the application of *Metapositions* introduced by [12]. A Metaposition can be represented by a node in a game tree. Unlike common game-tree nodes, such Metaposition nodes represent not one game state but a set of game states. A Metaposition node represents a set of game states consistent with all observations made by a player of an imperfect-information game up to a given time. Metapositions constitute a means of coping with the combinatorial explosion of Kriegspiel because they group multiple possible game states into single nodes of the game tree. This may result in reducing the branching factor of the game tree. However, an obvious drawback of this technique lies in the difficulty to group the game states. Successfully deciding which game states are grouped into which Metapositions requires domain knowledge and furthermore opponent modeling and a reliable evaluation function [1]. Although progress has been made with this technique, as documented by the work of [1, 12, 7] which focuses on endgames in Kriegspiel, the lack of a good evaluation function in Go has so far effectively discouraged the application of Metapositions to the domain of Phantom Go.

Only lately, [10] documented the possibility of applying Monte-Carlo techniques to Kriegspiel.

2.2 Go

While Phantom Go has escaped wider attention of the mainstream of computer-game research, a large corpus documenting AI research devoted to the game of Go has been compiled steadily over the years.

The complexity of the game of Go [13] and the lack of good evaluation functions has long delayed big-leap progress in computer Go. However, in recent years, Monte-Carlo methods have enabled substantial advances in the field. This progress unfolded in two steps. First, Monte-Carlo evaluation (cf. Subsection 3.1) was introduced and helped to remedy the weaknesses of handcrafted evaluation functions to some degree. Second, Monte-Carlo Game-Tree Search (cf. Subsection 3.2) was developed as a search framework for Monte-Carlo evaluation. The currently strongest Go programs are based on Monte-Carlo Tree Search.[1]

2.3 Phantom Go

While Monte-Carlo evaluation has been applied to Phantom Go [5], Monte-Carlo Tree Search has so far not been applied to this domain according to the knowledge of the authors. Cazenave [5] proposed applying Monte-Carlo evaluation to Phantom Go using all-as-first sampling (cf. Subsection 3.1). The program based on all-as-first sampling was able to beat experienced human Go players in human-vs-machine games of Phantom Go [5].

3 Monte-Carlo methods for Go and Phantom Go

This section gives a detailed description of Monte-Carlo methods relevant to the experiment described in Section 4. Subsection 3.1 outlines Monte-Carlo evaluation and two sampling techniques: (1) standard sampling, and (2) all-as-first sampling. Subsection 3.2 describes Monte-Carlo Tree Search. Subsection 3.3.1 presents how these methods can be applied to Phantom Go. In particular, it presents how UCT (a special move-selection function for Monte-Carlo Tree Search) can be applied to Phantom Go. Two new heuristics for estimating the positions of opponent stones are introduced: (1) late random opponent-move guessing, and (2) early probabilistic opponent-move guessing.

3.1 Monte-Carlo evaluation

Monte-Carlo evaluation is an evaluation function for game-tree search based on randomly sampled games. It was first proposed by [4], but the first competitive algorithmic adaptation of this evaluation method for computer Go was brought forward by [3]. Since then, numerous extensions to this evaluation have been brought forward (e.g., [2]).

Given a game state S, its *Monte-Carlo evaluation* is computed in three steps. (1) A number of random games is played from S to a terminal game state. (2) Each random game (*random sample*) is evaluated according to the rules of the game. (3) A statistical aggregate is computed for all games based on their respective results (e.g., the mean score, or the mean ratio of observed wins for the player to move in game state S).

In order to decide which game state S_i of a set of game states has the best Monte-Carlo evaluation, a number of Monte-Carlo evaluations are played out. A crucial distinction for computing the best evaluation refers to two sampling techniques: *standard sampling* and *all-as-first sampling* can be distinguished.

[1]Three Monte-Carlo Tree Search based programs, MOGO, CRAZY STONE, and STEENVRETER, finished as best 9×9 Go programs, MOGO and CRAZY STONE as best 19×19 programs in the 2007 Computer Olympiad in Amsterdam.

Standard sampling A number of Monte-Carlo evaluations are made for each S_i. The best S_i is the game state with the best evaluation. In order to decide which game state is the best, multiple samples are made succinctly.

All-as-first sampling In all-as-first sampling each sample game contributes to the evaluation of multiple game states S_i simultaneously. Each move random sample which results in a victory for the player to move first contributes to a statistic maintained for each of the S_i occurring in the random sample.

Standard sampling was found to produce better evaluations for Go than all-as-first sampling by [3]. [5] documented that all-as-first sampling produced acceptable evaluations in Phantom Go. The reason for this seemingly surprising difference is that all-as-first sampling exploits each sample more thoroughly because every sample contributes to multiple evaluations. At the same time the order of the moves is disregarded. Therefore, the resulting evaluation is less precise. Thus all-as-first sampling produces a more robust but less precise evaluation compared to standard sampling if only few samples are available. In Phantom Go, the state space is larger than in Go because the positions of the opponent stones are guessed. Therefore, only comparably few samples can be made. Thus all-as-first sampling produces better evaluations for Phantom Go.

3.2 Monte-Carlo Tree Search

Monte-Carlo Tree Search is a best-first search which gradually builds up a search tree based on Monte-Carlo evaluations. This section first describes Monte-Carlo Tree Search in general (Subsection 3.2.1). In Subsection 3.2.2 a specific move-selection function called UCT is explained.

3.2.1 The Monte-Carlo Tree Search Framework

Monte-Carlo Tree Search is a further development of the Monte-Carlo evaluation. It provides a tree-search framework for employing Monte-Carlo evaluations at the leaf nodes of a particular search tree.

Monte-Carlo Tree Search constitutes a family of tree-search algorithms applicable to the domain of board games [6, 8, 9]. In general, a Monte-Carlo Tree Search algorithm repeatedly applies a best-first-search iteration at the top level. Monte-Carlo sampling is used as an evaluation function at leaf nodes. The results of previous Monte-Carlo evaluations are used for developing the search tree. During each iteration, four stages are consecutively applied: (1) move selection; (2) expansion; (3) leaf-node evaluation; (4) back-propagation.

Each node N in the tree contains at least three different tokens of information: (i) a move representing the game-state transition associated with this node, (ii) the number $t(N)$ of times the node has been played during all previous iterations, and (iii) a value $v(N)$ representing an estimate of the nodes' game value. The search tree is held in memory. Before the first iteration, the tree consists only of the root node. While applying the four stages successively in each iteration, the tree grows gradually. The four stages of the iteration work as follows.

(1) The move selection determines a path from the root to a leaf node. This path is gradually developed. At each node, starting with the root node, the best successor node is selected by applying a move-selection function to all child nodes. Then, the same procedure is applied to the selected child node. This procedure is repeated until a leaf node is reached.

(2) After the leaf node is reached, it is decided whether this node will be expanded by storing some of its children in memory. The simplest rule, proposed by Coulom [8], is to expand one node per evaluation. The node expanded corresponds to the first position encountered that was not stored yet.

(3) A Monte-Carlo evaluation (also called *playout* or *simulation*) is applied to the leaf node. Monte-Carlo evaluation is the strategic task that selects moves in self-play until the end of the game. This task might consist of playing plain random moves or – better – pseudo-random moves. The main idea is to play more reasonable moves by using patterns, capture considerations, and proximity to the last move.

(4) During the back-propagation stage, the result of the leaf node is back-propagated through the path created in the move-selection stage. For each node in the path back to the root, the

node's game values are updated according to the updating function.[2] After the root node has been updated, this stage and the iteration are completed.

As a consequence of altering the values of the nodes on the path, the move selection of the next iteration is influenced. The various Monte-Carlo Tree Search algorithms proposed in the literature differ in their move-selection functions and updating functions. The following subsection briefly describes a specific move-selection function called UCT.

3.2.2 UCT

Kocsis and Szepesvári [9] introduced the move-selection function UCT. It was successfully applied in top-level Go programs such as Crazy Stone, Mogo, and Mango. These programs entered in KGS tournaments successfully.

Given a node N with children N_i, the move-selection function of UCT chooses the child node N_i which maximizes[3] the following criterion:

$$\overline{X}_i \; + \; C \sqrt{\frac{ln \;\, t(N)}{t(N_i)}} \;\; . \tag{1}$$

This criterion takes into account the number of times $t(N)$ that node N was played in previous iterations, the number of times $t(N_i)$ the child N_i was selected in previous iterations, and the average evaluation $\overline{X}_i$ of the child node N_i.

The weighted square-root term in Equation 1 describes an upper confidence bound for the average game value. This value is assumed to be normally distributed among the evaluations selected during all iterations. Each iteration passing through N represents a random experiment influencing the estimate of the mean parameter $\overline{X}$ of this distribution.

The constant C controls the balance between exploration and exploitation [6]. It prescribes how often a move represented by child node N_i with high confidence (large $t(N_i)$) and good average game value (large $\overline{X}_i$) is preferred to a move with lower confidence or lower average game value.

The updating function used together with UCT sets the value $\overline{X}$ of a node N to the average of all the values of the previously selected children including the latest selected child's value $\overline{X}_i$.

3.3 Monte-Carlo methods for Phantom Go

In this subsection we outline how the Monte-Carlo methods described so far in this section can be applied to the game of Phantom Go. Subsection 3.3.1 describes how Monte-Carlo evaluations have been applied to Phantom Go, and Subsection 3.3.2 proposes a way for applying UCT to Phantom Go and introduces two heuristics for guessing the locations of opponent stones in UCT.

3.3.1 Monte-Carlo evaluation for Phantom Go

Cazenave [5] suggested applying Monte-Carlo evaluation to Phantom Go by a simple two-step approach. First, the unknown opponent stones are randomly placed on the board. Second, a Monte-Carlo evaluation with *all-as-first* sampling is conducted to find the best move. While this approach is simple it produced a Phantom Go program which could beat experienced human Go players [5].

3.3.2 Monte-Carlo Tree Search for Phantom Go

When applying UCT to the move selection task in Phantom Go, a heuristic for guessing the unknown opponent stones is required. In Monte-Carlo evaluation, the number of unknown opponent stones was simply randomly added at the move node. We suggest two of such move guessing heuristics: (1) *late random opponent-move guessing*, and (2) *early probabilistic opponent-move guessing*. The two heuristics work as follows.

Late random guessing The depth-1 nodes represent all legal moves of the player P to move. Each branch in the tree is a sequence of moves. The Monte-Carlo sample plays each move in a

[2]Coulom [8] refers to the updating function as the back-propagation operator.

[3]In Min nodes the roles are reversed and the criterion is minimized.

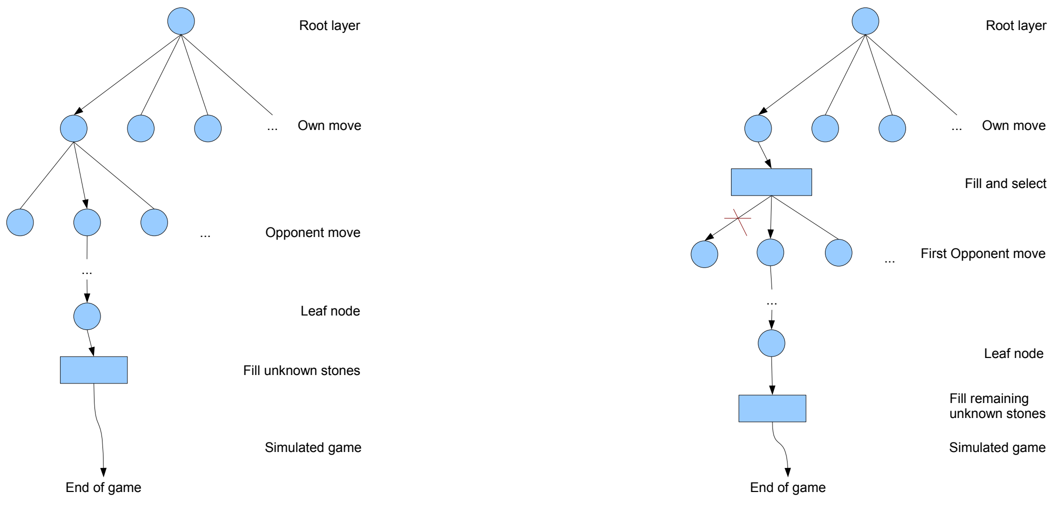

Figure 2: The late random opponent-move guessing heuristic (left) and the early probabilistic opponent-move guessing heuristic (right).

branch until the leaf-node is reached. The missing stones of the opponent are placed at random as soon as the leaf node's move is played out. Then, the move sequence is continued by Monte-Carlo evaluation.

Early probabilistic guessing Up to depth 1, the UCT tree is identical to the search tree maintained by late random opponent-move guessing. At depth 2, the opponent moves are added according to the frequency they have been played out with in previous iterations. More precisely, the probability for opponent stones to be played out is dependent on the number of times the node representing this move was visited (*visited*), the number of times its parent node was visited (*parent visited*) and the number of stones whose position is unknown (*unknown*) as: $visited/parent\ visited \times unknown$.

Early probabilistic opponent-move guessing aims at playing promising opponent moves early. This should lead to more realistic move sequences in the branch below depth 2. Both strategies for opponent-move guessing are depicted in Figure 2.

4 Experiment

This section describes a self-play experiment comparing four Monte-Carlo methods for playing Phantom Go. Subsection 4.1 describes the experimental set-up. Subsection 4.2 gives the results.

4.1 Set-up

A self-play experiment was conducted evaluating four Monte-Carlo methods described in Section 3 for playing Phantom Go: (1) Monte-Carlo evaluation with standard sampling (MCE_{std}), (2) Monte-Carlo evaluation with all-as-first sampling (MCE_{all}), (3) UCT with random late opponent-move guessing (UCT_{rand}), and (4) UCT with early probabilistic opponent-move guessing (UCT_{prob}).

All four methods were implemented in a Java framework for playing Phantom Go which we call InTheDark. The move generator for Monte-Carlo sampling is based on Mark Boon's open source library GoTools.[4] Various utilities are included from the open source software GoGui by Markus Enzenberger.[5] Each implementation played 50 games of Phantom Go against each other implementation, playing half of the games as Black and the other half as White. The time setting

[4]GoTools is documented at http://www.sente.ch/pub/software/tesuji/.
[5]GoGui is documented at http://gogui.sourceforge.net/.

	MCE_{std}	MCE_{all}	UCT_{rand}	UCT_{prob}	total wins
MCE_{std}	×	14	4	14	32
MCE_{all}	36	×	28	44	108
UCT_{rand}	46	22	×	42	110
UCT_{prob}	36	6	8	×	50

Table 1: Results of the pairwise comparison between the four Monte-Carlo methods. The rows show the number of victories the player in the leftmost column scored against the players in the other columns. The rightmost column entry of each row is the total number of victories achieved by the program in the first comlumn entry of the row.

for each match was set to 10 minutes per game per player. The experiments were carried out on a computing node with four AMD Opteron 3.4 processors and 32 GB RAM.

4.2 Results

The experiments consumed a total of ca. 42 hours on the above described hardware. The results of the matches between the implementations are given in Table 1.

A comparison of the two players based on Monte-Carlo evaluation shows that MCE_{all} (36 victories) outperforms MCE_{std} (14 victories). Of the two UCT-based players, random UCT_{rand} (42 victories) achieves better results than UCT_{prob} (8 victories). Although UCT_{rand} scores the highest number of total victories (110), MCE_{all} slightly outperforms it in a direct comparison by wining six games more. Players MCE_{all} and UCT_{rand} show a comparable number of total victories (108 and 110, respectively). Clearly, MCE_{std} performs worst with only 32 victories.

A qualitative analysis of the play by UCT_{prob} shows that this player often ranks good moves high but some bad moves even higher.

The results of the experiment allow the four following conclusions: (1) All-as-first sampling is a better choice for Monte-Carlo evaluation than standard sampling. (2) Late random opponent-move guessing is a better choice for UCT than early probabilistic opponent-move guessing. (3) The best player based on UCT does not outperform the best player based on Monte-Carlo evaluation but plays similarly strong. (4) Monte-Carlo evaluation with standard sampling yielded the weakest player.

5 Discussion

The results presented in the previous section indicate that choosing all-as-first sampling leads to better results than choosing standard sampling for a Phantom-Go player based on Monte-Carlo evaluation. In this respect, our findings support the findings of [5]. Surprisingly, the best UCT-based player does not outperform the best player based on Monte-Carlo evaluation although both players roughly play on the same level. The main reason for this seems to be that UCT players overestimate bad move positions. This problem seems to impact the UCT implementation with early probabilistic opponent-move ordering particularly gravely.

6 Conclusion and Outlook

This article compared four Monte-Carlo methods for Phantom Go: (1) Monte-Carlo evaluation with standard sampling, (2) Monte-Carlo evaluation with all-as-first sampling, (3) UCT with late random opponent-move guessing, and (4) UCT with early probabilistic opponent-move guessing. The empirical finding by [5] that Monte-Carlo evaluation with all-as-first sampling yields a reasonable Phantom-Go player was reproduced. The original contribution presented in this article is the application of a Monte-Carlo Tree Search algorithm (UCT) to Phantom Go and the introduction of two move-guessing heuristics. Surprisingly, the experiment showed that players based on UCT could not outperform the best player based on Monte-Carlo evaluation although a UCT-based player can reach a comparable level of play.

Future research will investigate whether applying grouping nodes for UCT in Phantom Go as was tested for regular Go by [11] is feasible. This approach could counterbalance the problems of applying Monte-Carlo Tree Search to Phantom Go so far. Also, testing the win ratio instead of the territory score as aggregate for the Monte-Carlo evaluation will be examined. Furthermore, the authors intend to produce a strong Phantom-Go player for the 2008 Computer Olympiad.

Acknowledgements

This work is partially funded by the Dutch Organization for Scientific Research NWO as part of the project Go for Go, grant number 612.066.409.

References

[1] Andrea Bolognesi and Paolo Ciancarini. Searching over Metapositions in Kriegspiel. In H. Jaap van den Herik, Yngvi Björnsson, and Nathan S. Netanyahu, editors, *Computers and Games*, volume 3846 of *Lecture Notes in Computer Science*, pages 246–261. Springer-Verlag, 2006.

[2] Bruno Bouzy and Guillaume Chaslot. Monte-Carlo Go Reinforcement Learning Experiments. In Sushil J. Louis and Graham Kendall, editors, *Computational Intelligence in Games*, pages 187–194. IEEE, 2006.

[3] Bruno Bouzy and Bernd Helmstetter. Monte Carlo Developments. In H. Jaap van den Herik, Hiroyuki Iida, and Ernst A. Heinz, editors, *Proceedings of the Advances in Computer Games Conference (ACG-10)*, pages 159–174, Boston, USA, 2003. Kluwer Academic.

[4] Bernd Brügmann. Monte Carlo Go. *White paper*, 1993.

[5] Tristan Cazenave. A Phantom-Go Program. In H. Jaap van den Herik, Shun chin Hsu, Tsan sheng Hsu, and H. H. L. M. Donkers, editors, *Advances in Computer Games, (ACG 11)*, LNCS, pages 120–126, Springer-Verlag, Berlin, 2005.

[6] Guillaume Chaslot, Jahn-Takeshi Saito, Bruno Bouzy, Jos W. H. M. Uiterwijk, and H. Jaap van den Herik. Monte-Carlo Strategies for Computer Go. In Pierre-Yves Schobbens, Wim Vanhoof, and Gabriel Schwanen, editors, *Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91, 2006.

[7] Paolo Ciancarini and Gian Piero Favini. A Program to Play Kriegspiel. *ICGA Journal*, 30(1):3–24, 2007.

[8] Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In Paolo Ciancarini, H. Jaap van den Herik, and Jeroen H.L.M. Donkers, editors, *Proceedings of the Fifth Computers and Games Conference*, LNCS, Springer-Verlag, Berlin, 2007. 12 pages, in print.

[9] Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Proceedings of the EMCL 2006*, volume 4212 of *LNCS*, pages 282–293, Springer-Verlag, Berlin, 2006.

[10] Austin Parker, Dana Nau, and V.S. Subrahmanian. Game-Tree Search with Combinatorially Large Belief States. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 254–259, Edinburgh, Scotland, UK, 2005. Professional Book Center. ISBN 0938075934.

[11] Jahn-Takeshi Saito, Mark H. M. Winands, Jos W.H.M. Uiterwijk, and H. Jaap van den Herik. Grouping Nodes for Monte-Carlo Tree Search. In Jos W.H.M. Uiterwij, Maarten Schadd, Mark H.M. Winands, and H. Jaap van den Herik, editors, *Proceedings of the Computer Games Workshop 2007 (CGW 2007)*, volume 07-06 of *MICC Technical Report Series*, pages 125–132, Maastricht, The Netherlands, 2007.

[12] Makoto Sakuta and Hiroyuki Iida. Solving Kriegspiel-like Problems: Exploiting a Transposition Table. *ICGA Journal*, 23(4):218–229, 2000.

[13] Eric C. D. van der Werf. *AI techniques for the game of Go*. PhD thesis, Universiteit Maastricht, 2004.