Monte-Carlo Tree Search Solver

Mark H.M. Winands[1], Yngvi Björnsson[2], and Jahn-Takeshi Saito[1]

[1] Games and AI Group, MICC, Faculty of Humanities and Sciences, Universiteit Maastricht, Maastricht, The Netherlands
{m.winands,j.saito}@micc.unimaas.nl

[2] School of Computer Science, Reykjavík University, Reykjavík, Iceland
yngvi@ru.is

Abstract. Recently, Monte-Carlo Tree Search (MCTS) has advanced the field of computer Go substantially. In this article we investigate the application of MCTS for the game Lines of Action (LOA). A new MCTS variant, called MCTS-Solver, has been designed to play narrow tactical lines better in sudden-death games such as LOA. The variant differs from the traditional MCTS in respect to backpropagation and selection strategy. It is able to prove the game-theoretical value of a position given sufficient time. Experiments show that a Monte-Carlo LOA program using MCTS-Solver defeats a program using MCTS by a winning score of 65%. Moreover, MCTS-Solver performs much better than a program using MCTS against several different versions of the world-class $\alpha\beta$ program MIA. Thus, MCTS-Solver constitutes genuine progress in using simulation-based search approaches in sudden-death games, significantly improving upon MCTS-based programs.

1 Introduction

For decades $\alpha\beta$ search has been the standard approach used by programs for playing two-person zero-sum games such as chess and checkers (and many others). Over the years many search enhancements have been proposed for this framework. However, in some games where it is difficult to construct an accurate positional evaluation function (e.g., Go) the $\alpha\beta$ approach was hardly successful. In the past, Monte-Carlo (MC) methods have been used as an evaluation function in a search-tree context [1, 6, 7]. A direct descendent of that approach is a new general search method, called Monte-Carlo Tree Search (MCTS) [10, 14]. MCTS is not a classical tree search followed by a MC evaluation, but rather a best-first search guided by the results of Monte-Carlo simulations. In the last two years MCTS has advanced the field of computer Go substantially. Moreover, it is used in other games as well (Phantom Go [8], Clobber [15]), even for games where there exists already a reasonable evaluation function (e.g., Amazons [13]). Although MCTS is able to find the best move, it is not able to prove the game-theoretic value of (even parts of) the search tree. A search method that is not able to prove or estimate (quickly) the game-theoretic value of a node may run into problems. This is especially true for sudden-death games, such as chess, that may abruptly end by the creation of one of a prespecified set of patterns

[2] (e.g., checkmate in chess). In this case $\alpha\beta$ search or a special endgame solver (i.e., Proof-Number Search [3]) is traditionally preferred above MCTS.

In this article we introduce a new MCTS variant, called MCTS-Solver, which has been designed to prove the game-theoretical value of a node in a search tree. This is an important step towards being able to use MCTS-based approaches effectively in sudden-death like games (including chess). We use the game Lines of Action (LOA) as a testbed. It is an ideal candidate because its intricacies are less complicated than those of chess. So, we can focus on the sudden-death property. Furthermore, because LOA was used as a domain for various other AI techniques [5, 12, 20], the level of the state-of-the-art LOA programs is high, allowing us to look at how MCTS approaches perform against increasingly stronger evaluation functions. Moreover, the search engine of a LOA program is quite similar to the one of a chess program.

The article is organized as follows. Section 2 explains briefly the rules of LOA. In Sect. 3 we discuss MCTS and its application to Monte-Carlo LOA. In Sect. 4 we introduce MCTS-Solver. We empirically evaluate the method in Sect. 5. Finally, Sect. 6 gives conclusions and an outlook on future research.

2 Lines of Action

Lines of Action (LOA) [16] is a two-person zero-sum connection game with perfect information. It is played on an 8×8 board by two sides, Black and White. Each side has twelve pieces at its disposal. The black pieces are placed in two rows along the top and bottom of the board, while the white pieces are placed in two files at the left and right edge of the board. The players alternately move a piece, starting with Black. A move takes place in a straight line, exactly as many squares as there are pieces of either color anywhere along the line of movement. A player may jump over its own pieces. A player may not jump over the opponent's pieces, but can capture them by landing on them. The goal of a player is to be the first to create a configuration on the board in which all own pieces are connected in one unit (i.e., the sudden-death property). In the case of simultaneous connection, the game is drawn. The connections within the unit may be either orthogonal or diagonal. If a player cannot move, this player has to pass. If a position with the same player to move occurs for the third time, the game is drawn.

3 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) [10, 14] is a best-first search method that does not require a positional evaluation function. It is based on a randomized exploration of the search space. Using the results of previous explorations, the algorithm gradually builds up a game tree in memory, and successively becomes better at accurately estimating the values of the most promising moves.

MCTS consists of four strategic steps, repeated as long as there is time left. The steps are as follows. (1) In the *selection step* the tree is traversed from the root node until we reach a node E, where we select a position that is not added to the tree yet. (2) Next, during the *play-out step* moves are played in self-play until the end of the game is reached. The result R of this "simulated" game is $+1$ in case of a win for Black (the first player in LOA), 0 in case of a draw, and -1 in case of a win for White. (3) Subsequently, in the *expansion step* children of E are added to the tree. (4) Finally, R is propagated back along the path from E to the root node in the *backpropagation step*. When time is up, the move played by the program is the child of the root with the highest value.

3.1 The four strategic steps

The four strategic steps of MCTS are discussed in detail below. We will demonstrate how each of these steps is used in our Monte-Carlo LOA program.

Selection. Selection picks a child to be searched based on previous gained information. It controls the balance between exploitation and exploration. On the one hand, the task often consists of selecting the move that leads to the best results so far (exploitation). On the other hand, the less promising moves still must be tried, due to the uncertainty of the evaluation (exploration).

We use the UCT (**U**pper **C**onfidence Bounds applied to **T**rees) strategy [14], enhanced with Progressive Bias (PB [9]). UCT is easy to implement and used in many Monte-Carlo Go programs. PB is a technique to embed domain-knowledge bias into the UCT formula. It is successfully applied in the Go program MANGO. UCT with PB works as follows. Let I be the set of nodes immediately reachable from the current node p. The selection strategy selects the child k of the node p that satisfies Formula 1:

$$k \in argmax_{i \in I} \left(v_i + \sqrt{\frac{C \times \ln n_p}{n_i}} + \frac{W \times P_c}{n_i + 1} \right) \quad , \tag{1}$$

where v_i is the value of the node i, n_i is the visit count of i, and n_p is the visit count of p. C is a coefficient, which has to be tuned experimentally. $\frac{W \times P_c}{n_i+1}$ is the PB part of the formula. W is a constant, which has to be set manually (here $W = 100$). P_c is the *transition probability* of a move category c [17].

For each move category (e.g., capture, blocking) the probability that a move belonging to that category will be played is determined. The probability is called the *transition probability*. This statistic is obtained from game records of matches played by expert players. The transition probability for a move category c is calculated as follows:

$$P_c = \frac{n_{played(c)}}{n_{available(c)}} \quad , \tag{2}$$

where $n_{played(c)}$ is the number of game positions in which a move belonging to category c was played, and $n_{available(c)}$ is the number of positions in which moves belonging to category c were available.

The move categories of our Monte-Carlo LOA program are similar to the ones used in the Realization-Probability Search of the program MIA [21]. They are used in the following way. First, we classify moves as captures or non-captures. Next, moves are further sub-classified based on the origin and destination squares. The board is divided into five different regions: the corners, the 8×8 outer rim (except corners), the 6×6 inner rim, the 4×4 inner rim, and the central 2×2 board. Finally, moves are further classified based on the number of squares traveled away from or towards the center-of-mass. In total 277 move categories can occur according to this classification.

This selection strategy is only applied in nodes with visit count higher than a certain threshold T (here 50) [10]. If the node has been visited fewer times than this threshold, the next move is selected according to the *simulation strategy* discussed in the next strategic step.

Play-out. The play-out step begins when we enter a position that is not a part of the tree yet. Moves are selected in self-play until the end of the game. This task might consist of playing plain random moves or – better – pseudo-random moves chosen according to a *simulation strategy*. It is well-known that the use of an adequate simulation strategy improves the level of play significantly [11]. The main idea is to play interesting moves according to heuristic knowledge. In our Monte-Carlo LOA program, the move categories together with their transition probabilities, as discussed in the selection step, are used to select the moves pseudo-randomly during the play-out.

A simulation requires that the number of moves per game is limited. When considering the game of LOA, the simulated game is stopped after 200 moves and scored as a draw. The game is also stopped when heuristic knowledge indicates that the game is probably over. The reason for doing this is that despite the use of an elaborate simulation strategy it may happen that the game-theoretical value and the average result of the Monte-Carlo simulations differ substantially from each other in some positions. In our Monte-Carlo LOA program this so-called noise is reduced by using the MIA 4.5 evaluation function [23]. When the evaluation function gives a value that exceeds a certain threshold (i.e., 1,000 points), the game is scored as a win. If the evaluation function gives a value that is below a certain threshold (i.e., -1,000 points), the game is scored as a loss. For speed reasons the evaluation function is called only every 3 plies, determined by trial and error.

Expansion. Expansion is the strategic task that decides whether nodes will be added to the tree. Here, we apply a simple rule: one node is added per simulated game [10]. The added leaf node L corresponds to the first position encountered during the traversal that was not already stored.

Backpropagation. Backpropagation is the procedure that propagates the *result* of a simulated game k back from the leaf node L, through the previously tra-

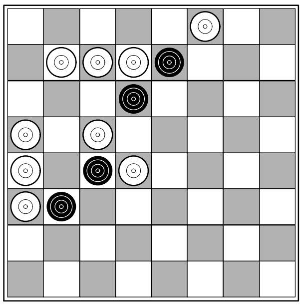

Fig. 1. White to move

versed node, all the way up to the root. The result is scored positively ($R_k = +1$) if the game is won, and negatively ($R_k = -1$) if the game is lost. Draws lead to a result $R_k = 0$. A *backpropagation strategy* is applied to the *value* v_L of a node. Here, it is computed by taking the average of the results of all simulated games made through this node [10], i.e., $v_L = (\sum_k R_k)/n_L$.

4 Monte-Carlo Tree Search Solver

Although MCTS is unable to *prove* the game-theoretic value, in the long run MCTS equipped with the UCT formula is able to *converge* to the game-theoretic value. For a fixed termination game like Go, MCTS is able to find the optimal move relatively fast [25]. But in a sudden-death game like LOA, where the main line towards the winning position is narrow, MCTS may often lead to an erroneous outcome because the nodes' values in the tree do not converge fast enough to their game-theoretical value. For example, if we let MCTS analyze the position in Fig. 1 for 5 seconds, it selects **c7xc4** as the best move, winning 67.2% of the simulations. However, this move is a forced 8-ply loss, while **f8-f7** (scoring 48.2%) is a 7-ply win. Only when we let MCTS search for 60 seconds, it selects the optimal move. For a reference, we remark that it takes $\alpha\beta$ in this position less than a second to select the best move and prove the win.

We designed a new variant called MCTS-Solver, which is able to prove the game-theoretical value of a position. The backpropagation and selection mechanisms have been modified for this variant. The changes are discussed in Subsections 4.1 and 4.2, respectively. Moreover, we discuss the consequences for final move selection in Subsection 4.3. The pseudo-code of MCTS-Solver is given in Subsection 4.4.

4.1 Backpropagation

In addition to backpropagating the values {1,0,−1}, the search also propagates the game-theoretical values ∞ or $-\infty$.[3] The search assigns ∞ or $-\infty$ to a won or lost terminal position for the player to move in the tree, respectively. Propagating the values back in the tree is performed similar to negamax in the context of minimax searching in such a way that we do not need to distinguish between MIN and MAX nodes. If the selected move (child) of a node returns ∞, the node is a win. To prove that a node is a win, it suffices to prove that one child of that node is a win. Because of negamax, the value of the node will be set to $-\infty$. In the minimax framework it would be set to ∞. In the case that the selected child of a node returns $-\infty$, all its siblings have to be checked. If their values are also $-\infty$, the node is a loss. To prove that a node is a loss, we must prove that all its children lead to a loss. Because of negamax, the node's value will be set to ∞. In the minimax framework it would have been set to $-\infty$. In the case that one or more siblings of the node have a different value, we cannot prove the loss. Therefore, we will propagate −1, the result for a lost game, instead of $-\infty$, the game-theoretical value of a position. The value of the node will be updated according to the backpropagation strategy as described in Subsection 3.1.

4.2 Selection

As seen in the previous subsection, a node can have the game-theoretical value ∞ or $-\infty$. The question arises how these game-theoretical values affect the selection strategy. Of course, when a child is a proven win, the node itself is a proven win, and no selection has to take place. But when one or more children are proven to be a loss, it is tempting to discard them in the selection phase. However, this can lead to overestimating the value of a node, especially when moves are pseudo-randomly selected by the simulation strategy. For example, in Fig. 2 we have three one-ply subtrees. Leaf nodes B and C are proven to be a loss, indicated by $-\infty$; the numbers below the other leaves are the *expected* pay-off values. Assume that we select the moves with the same likelihood (as could happen when a simulation strategy is applied). If we would prune the loss nodes, we would prefer node A above E. The average of A would be 0.4 and 0.37 for E. It is easy to see that A is overestimated because E has more good moves.

If we do not prune proven loss nodes, we run the risk of underestimation. Especially, when we have a strong preference for certain moves (because of a bias) or we would like to explore our options (because of the UCT formula), we could underestimate positions. Assume that we have a strong preference for the first move in the subtrees of Fig. 2. We would prefer node I above A. It is easy to see that A is underestimated because I has no good moves at all.

Based on preliminary experiments, selection is here performed in the following way. In case Formula (1) is applied, child nodes with the value $-\infty$ will never

[3] Draws are in general more problematic to prove than wins and losses. Because draws only happen in exceptional cases in LOA, we took the decision not to handle proven draws for efficiency reasons.

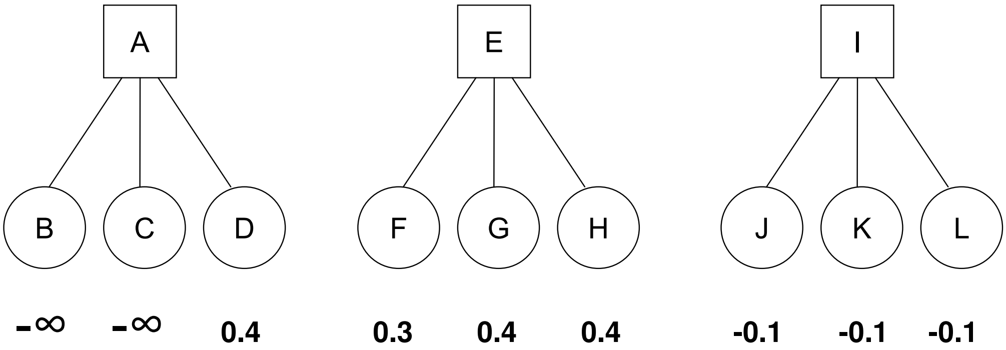

Fig. 2. Monte-Carlo Subtrees

be selected. For nodes of which the visit count is below the threshold, moves are selected according to the simulation strategy instead of using Formula (1). In that case, children with the value $-\infty$ *can* be selected. However, when a child with a value $-\infty$ is selected, the search is not continued at that point. The results are propagated backwards according to the strategy described in the previous subsection.

For all the children of a leaf node (i.e., the visit count equals one) we check whether they lead to a direct win for the player to move. If there is such a move, we stop searching at this node and set the node's value (negamax: $-\infty$; minimax: ∞). This check at the leaf node must be performed because otherwise it could take many simulations before the child leading to a mate-in-one is selected and the node is proven.

4.3 Final move selection

For standard MCTS several ways exist to select the move finally played by the program in the actual game. Often, it is the child with the highest visit count, or with the highest value, or a combination of the two. In practice, there is no significant difference when a sufficient amount of simulations for each root move has been played. However, for MCTS-Solver it *does* matter. Because of the backpropagation of game-theoretical values, the score of a move can suddenly drop or rise. Therefore, we have chosen a method called *Secure child* [9]. It is the child that maximizes the quantity $v + \frac{A}{\sqrt{n}}$, where A is a parameter (here, set to 1), v is the node's value, and n is the node's visit count.

Finally, when a win can be proven for the root node, the search is stopped and the winning move is played. For the position in Fig. 1, MCTS-Solver is able to select the best move and prove the win for the position depicted in less than a second.

4.4 Pseudo Code for MCTS-Solver

A C-like pseudo code of MCTS-Solver is provided in Fig. 3. The algorithm is constructed similar to negamax in the context of minimax search. `select(Node N)` is the selection function as discussed in Subsection 4.2, which returns the best child of the node N. The procedure `addToTree(Node node)` adds one more node to the tree; `playOut(Node N)` is the function which plays a simulated game from the node N, and returns the result $R \in \{1, 0, -1\}$ of this game; `computeAverage(Integer R)` is the procedure that updates the value of the node depending on the result R of the last simulated game; `getChildren(Node N)` generates the children of node N.

5 Experiments

In this section we evaluate the performance of MCTS-Solver. First, we matched MCTS-Solver against MCTS, and provide results in Subsection 5.1. Next, we evaluated the playing-strength of MCTS and MCTS-Solver against different versions of the tournament LOA program MIA, as shown in Subsection 5.2. All experiments were performed on a Pentium IV 3.2 GHz computer.

5.1 MCTS vs. MCTS-Solver

In the first series of experiments MCTS and MCTS-Solver played 1,000 games against each other, playing both colors equally. They always started from the same standardized set of 100 three-ply positions [5]. The thinking time was limited to 5 seconds per move.

Table 1. 1,000-game match results

	Score	Win %	Winning ratio
MCTS-Solver vs. MCTS	646.5 - 353.5	65%	1.83

The results are given in Table 1. MCTS-Solver outplayed MCTS with a winning score of 65% of the available points. The winning ratio is 1.83, meaning that it scored 83% more points than the opponent. This result shows that the MCTS-Solver mechanism improves the playing strength of the Monte-Carlo LOA program.

5.2 Monte-Carlo LOA vs. MIA

In the previous subsection, we saw that MCTS-Solver outperformed MCTS. In the next series of experiments, we further examine whether MCTS-Solver is superior to MCTS by comparing the playing strength of both algorithms against

```
Integer MCTSSolver(Node N){

 if(playerToMoveWins(N))
    return INFINITY
 else (playerToMoveLoses(N))
    return -INFINITY

 bestChild = select(N)
 N.visitCount++

 if(bestChild.value != -INFINITY AND bestChild.value != INFINITY)
    if(bestChild.visitCount == 0){
        R = -playOut(bestChild)
        addToTree(bestChild)
        goto DONE
    }
    else
     R = -MCTSSolver(bestChild)
 else
  R = bestChild.value

 if(R == INFINITY){
        N.value = -INFINITY
        return R
    }
 else
    if(R == -INFINITY){

        foreach(child in getChildren(N))
          if(child.value != R){
             R = -1
             goto DONE
          }

          N.value = INFINITY
          return R
     }

  DONE:
  N.computeAverage(R)
  return R
 }
```

Fig. 3. Pseudo code for MCTS-Solver

a non-MC program. We used three different versions of MIA, considered being the best LOA playing entity in the world.[4] The three different versions were all equipped with the same latest search engine but used three different evaluation functions (called MIA 2000 [19], MIA 2002 [22], and MIA 2006 [23]). The search engine is an $\alpha\beta$ depth-first iterative-deepening search in the Enhanced Realization-Probability Search (ERPS) framework [21] using several forward pruning mechanisms [24]. To prevent the programs from repeating games, a small random factor was included in the evaluation functions. All programs played under the same tournament conditions as used in Subsection 5.1. The results are given in Table 2. Each match consisted of 1,000 games.

Table 2. 1,000-game match results

Evaluator	MIA 2000	MIA 2002	MIA 2006
MCTS	585.5	394.0	69.5
MCTS-Solver	692.0	543.5	115.5

In Table 2 we notice that MCTS and MCTS-Solver score more than 50% against MIA 2000. When competing with MIA 2002, only MCTS-Solver is able to outperform the $\alpha\beta$ program. Both MC programs are beaten by MIA 2006, although MCTS-Solver scores a more respectable number of points. Table 2 indicates that MCTS-Solver when playing against each MIA version significantly performs better than MCTS does. These results show that MCTS-Solver is a genuine improvement, significantly enhancing MCTS. The performance of the Monte-Carlo LOA programs in general against MIA — a well-established $\alpha\beta$ program — is quite impressive. One must keep in mind the many man-months of work that are behind the increasingly sophisticated evaluation functions of MIA. The improvement introduced here already makes a big leap in the playing strength of the simulation-based approach, resulting in MCTS-Solver even winning the already quite advanced MIA 2002 version. Admittedly, there is still a considerable gap to be closed for MTCS-Solver before it will be a match for the MIA 2006 version. Nonetheless, with continuing improvements it is not unlikely that in the near future simulation-based approaches may become an interesting alternative in games that the classic $\alpha\beta$ approach has dominated. This work is one step towards that goal being realized.

6 Conclusion and Future Research

In this article we introduced a new MCTS variant, called MCTS-Solver. This variant differs from the traditional MC approaches in that it can prove game-

[4] The program won the LOA tournament at the eighth (2003), ninth (2004), and eleventh (2006) Computer Olympiad.

theoretical outcomes, and thus converges much faster to the best move in narrow tactical lines. This is especially important in tactical sudden-death-like games such as LOA. Our experiments show that a MC-LOA program using MCTS-Solver defeats the original MCTS program by an impressive winning score of 65%. Moreover, when playing against a state-of-the-art $\alpha\beta$-based program, MCTS-Solver performs much better than a regular MCTS program. Thus, we may conclude that MCTS-Solver is a genuine improvement, significantly enhancing MCTS. Although MCTS-Solver is still lacking behind the best $\alpha\beta$-based program, we view this work as one step towards that goal of making simulation-based approaches work in a wider variety of games. For these methods, to be able to handle proven outcomes is one essential step to make. With continuing improvements it is not unlikely that in the not so distant future enhanced simulation-based approaches may become a competitive alternative to $\alpha\beta$ search in games dominated by the latter so far.

As future research, experiments are envisaged in other games to test the performance of MCTS-Solver. One possible next step would be to test the method in Go, a domain in which MCTS is already widely used. What makes this a somewhat more difficult task is that additional work is required in enabling perfect endgame knowledge - such as Benson's Algorithm [4, 18] - in MCTS. We have seen that the performance of the Monte-Carlo LOA programs against MIA in general indicates that they could even be an interesting alternative to the classic $\alpha\beta$ approach. Parallelization of the program using an endgame specific evaluation function instead of a general one such as MIA 4.5 could give a performance boost.

Acknowledgments. The authors thank Guillaume Chaslot for giving valuable advice on MCTS. Part of this work is done in the framework of the NWO Go for Go project, grant number 612.066.409.

References

1. B. Abramson. Expected-outcome: A general model of static evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(2):182–193, 1990.
2. L.V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Rijksuniversiteit Limburg, Maastricht, The Netherlands, 1994.
3. L.V. Allis, M. van der Meulen, and H.J. van den Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–123, 1994.
4. D.B. Benson. Life in the Game of Go. In D.N.L. Levy, editor, *Computer Games*, volume 2, pages 203–213. Springer-Verlag, New York, NY, 1988.
5. D. Billings and Y. Björnsson. Search and knowledge in Lines of Action. In H.J. van den Herik, H. Iida, and E.A. Heinz, editors, *Advances in Computer Games 10: Many Games, Many Challenges*, pages 231–248. Kluwer Academic Publishers, Boston, MA, USA, 2003.
6. B. Bouzy and B. Helmstetter. Monte-Carlo Go Developments. In H.J. van den Herik, H. Iida, and E.A. Heinz, editors, *Advances in Computer Games 10: Many Games, Many Challenges*, pages 159–174. Kluwer Academic Publishers, Boston, MA, USA, 2003.

7. B. Brügmann. Monte Carlo Go. Technical report, Physics Department, Syracuse University, 1993.
8. T. Cazenave and J. Borsboom. Golois Wins Phantom Go Tournament. *ICGA Journal*, 30(3):165–166, 2007.
9. G.M.J-B. Chaslot, M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. van den Herik, and B. Bouzy. Progressive strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 4(3):343–357, 2008.
10. R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers, editors, *Proceedings of the 5th International Conference on Computer and Games*, volume 4630 of *Lecture Notes in Computer Science (LNCS)*, pages 72–83. Springer-Verlag, Heidelberg, Germany, 2007.
11. S. Gelly and D. Silver. Combining online and offline knowledge in UCT. In Z. Ghahramani, editor, *Proceedings of the International Conference on Machine Learning (ICML)*, number 227 in ACM International Conference Proceeding Series, pages 273–280. ACM, 2007.
12. B. Helmstetter and T. Cazenave. Architecture d'un programme de Lines of Action. In T. Cazenave, editor, *Intelligence artificielle et jeux*, pages 117–126. Hermes Science, 2006. In French.
13. J. Kloetzer, H. Iida, and B. Bouzy. The Monte-Carlo Approach in Amazons. In H.J. van den Herik, J.W.H.M. Uiterwijk, M.H.M. Winands, and M.P.D. Schadd, editors, *Proceedings of the Computer Games Workshop 2007 (CGW 2007)*, pages 185–192, Universiteit Maastricht, Maastricht, The Netherlands, 2007.
14. L. Kocsis and C. Szepesvári. Bandit Based Monte-Carlo Planning. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Artificial Intelligence*, pages 282–293, 2006.
15. L. Kocsis, C. Szepesvári, and J. Willemson. Improved Monte-Carlo Search, 2006. http://zaphod.aml.sztaki.hu/papers/cg06-ext.pdf.
16. S. Sackson. *A Gamut of Games*. Random House, New York, NY, USA, 1969.
17. Y. Tsuruoka, D. Yokoyama, and T. Chikayama. Game-tree search algorithm based on realization probability. *ICGA Journal*, 25(3):132–144, 2002.
18. E.C.D. van der Werf, H.J. van den Herik, and J.W.H.M. Uiterwijk. Solving Go on small boards. *ICGA Journal*, 26(2):92–107, 2003.
19. M.H.M. Winands. Analysis and implementation of Lines of Action. Master's thesis, Universiteit Maastricht, Maastricht, The Netherlands, 2000.
20. M.H.M. Winands. *Informed Search in Complex Games*. PhD thesis, Universiteit Maastricht, Maastricht, The Netherlands, 2004.
21. M.H.M. Winands and Y. Björnsson. Enhanced realization probability search. *New Mathematics and Natural Computation*, 4(3):329–342, 2008.
22. M.H.M. Winands, L. Kocsis, J.W.H.M. Uiterwijk, and H.J. van den Herik. Temporal difference learning and the Neural MoveMap heuristic in the game of Lines of Action. In Q. Mehdi, N. Gough, and M. Cavazza, editors, *GAME-ON 2002*, pages 99–103, Ghent, Belgium, 2002. SCS Europe Bvba.
23. M.H.M. Winands and H.J. van den Herik. MIA: a world champion LOA program. In *The 11th Game Programming Workshop in Japan (GPW 2006)*, pages 84–91, 2006.
24. M.H.M. Winands, H.J. van den Herik, J.W.H.M. Uiterwijk, and E.C.D. van der Werf. Enhanced forward pruning. *Information Sciences*, 175(4):315–329, 2005.
25. P. Zhang and K. Chen. Monte-Carlo Go tactic search. In P. Wang et al., editors, *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, pages 662–670. World Scientific Publishing Co. Pte. Ltd., 2007.