Evaluation Function Based Monte-Carlo LOA

Mark H.M. Winands[1] and Yngvi Björnsson[2]

[1] Games and AI Group, Department of Knowledge Engineering,
Faculty of Humanities and Sciences,
Maastricht University, Maastricht, The Netherlands
m.winands@maastrichtuniversity.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. Also in the game of Lines of Action (LOA), which has been dominated so far by $\alpha\beta$, MCTS is making an inroad. In this paper we investigate how to use a positional evaluation function in a Monte-Carlo simulation-based LOA program (MC-LOA). Four different simulation strategies are designed, called Evaluation Cut-Off, Corrective, Greedy, and Mixed. They use an evaluation function in several ways. Experimental results reveal that the Mixed strategy is the best among them. This strategy draws the moves randomly based on their transition probabilities in the first part of a simulation, but selects them based on their evaluation score in the second part of a simulation. Using this simulation strategy the MC-LOA program plays at the same level as the $\alpha\beta$ program MIA, the best LOA-playing entity in the world.

1 Introduction

The $\alpha\beta$ search algorithm has for decades been the standard technique used by game 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 to further improve its effectiveness. However, for some games this approach has been less successful, either because of a large branching factor preventing a deep look-a-head, or because of complications in constructing an effective evaluation function.

In recent years a new paradigm for game-tree search has emerged, the so-called Monte-Carlo Tree Search (MCTS) [9,12]. In the context of game playing, Monte-Carlo simulations were first used as a mechanism for dynamically evaluating the merits of leaf nodes of a traditional $\alpha\beta$-based search [1,3,4], but under the new paradigm MCTS has evolved into a full-fledged best-first search procedure that replaces traditional $\alpha\beta$-based search altogether. MCTS has in the past couple of years substantially advanced the state-of-the-art in several game domains where $\alpha\beta$-based search has had difficulties. In particular we mention computer Go, but other domains include General Game Playing [10], Phantom Go [5], and Amazons [13]. These are, however, all examples of game domains where either a

J. van den Herik and P. Spronck (Eds.): ACG 2009, LNCS 6048, pp. 33–44, 2010.

large branching factor or a complex static state evaluation do restrain $\alpha\beta$ search in one way or another.

In this paper we introduce an improved MCTS variant that performs at the same level as a world-class $\alpha\beta$-based player in the game Lines of Action (LOA) [14]. This is an important milestone for MCTS, because up until now the traditional game-tree search approach has generally been considered to be better suited for LOA, which features both a moderate branching factor and good state evaluators (the best LOA programs use highly sophisticated evaluation functions). The previously best game-playing programs for this game, MIA [16], Bing, YL [2], and Mona [2], are all $\alpha\beta$ based. Recent work on using a special MCTS-solver variant in the world-class LOA program MIA did improve the program's tactical ability, although it still lacked the overall robustness to play against its $\alpha\beta$-based counterpart on a close to equal footing [18]. In this work our MCTS-solver variant enriched with a positional evaluation function, is able to hold its own. Moreover, the solver is easily parallelizable and when allowed to use more than one processor it does handily outperform the best $\alpha\beta$-based LOA programs.

The article is organized as follows. Section 2 explains briefly the rules of LOA. In Sect. 3 we discuss the application of MCTS in the game of LOA. In Sect. 4 we present several play-out strategies for MC-LOA. We test them in Sect. 5. Finally, Sect. 6 gives conclusions and an outlook on future research.

2 Lines of Action

Lines of Action (LOA) 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 along the top and bottom rows of the board, while the white pieces are placed in the left- and right-most files of the board (see Fig. 1a). The players alternately move a piece, starting with Black. A piece moves in a straight line, exactly as many squares as there are pieces of either color anywhere along the line of movement (see Fig. 1b). A player may jump over its own pieces, but not over the opponent's. The rule is that opponent's pieces are captured (and removed from the board) by landing on them. The goal of the players is to be the first to create a configuration on the board in which all own pieces are connected in one unit (e.g., see Fig. 1c). The connections within the unit may be either orthogonal or diagonal. In the case of simultaneous connection, the game is drawn.

3 Monte-Carlo LOA

In this section we discuss how we applied MCTS in LOA. First, we briefly sketch MCTS and its variant MCTS-Solver in Subsect. 3.1. Next, we explain MCTS (-Solver) in detail in Subsect. 3.2. Finally, we explain how we parallelized the search in Subsect. 3.3.

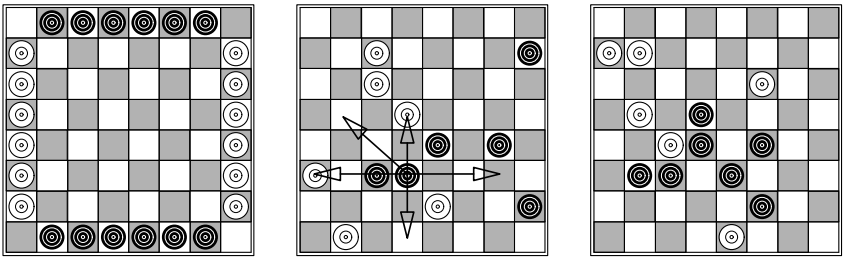

Fig. 1. (a) The initial position. (b) Example of possible moves. (c) A terminal position.

3.1 MCTS and MCTS-Solver

Monte-Carlo Tree Search (MCTS) [9,12] 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 deliberation 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.

MCTS is unable to *prove* the game-theoretic value. However, in the long run MCTS equipped with the UCT formula [12] *converges* to the game-theoretic value. For a fixed termination game such as Go, MCTS is able to find the optimal move relatively fast in endgame positions [20]. But in a sudden-death game such as 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. We use therefore a newly proposed variant called Monte-Carlo Tree Search Solver (MCTS-Solver) [18] in our MC-LOA program. The MCTS-Solver is able to prove the game-theoretical value of a position. The backpropagation and selection mechanisms have been modified for this variant.

3.2 The Four Strategic Steps

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

Selection. Selection picks a child to be searched based on previously 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 [12], enhanced with Progressive Bias (PB [7]). UCT is easy to implement and used in many Monte-Carlo Go programs. PB is a technique to embed the domain-knowledge bias into the UCT formula. 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:

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

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 must be tuned experimentally. $\frac{W \times P_{mc}}{n_i+1}$ is the PB part of the formula. W is a constant, which must be set manually (here $W = 10$). P_{mc} is the *transition probability* of a move category mc [15].

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 off-line from game records of matches played by expert players. The transition probability for a move category c is calculated as follows:

$$P_{mc} = \frac{n_{played(mc)}}{n_{available(mc)}} \ , \tag{2}$$

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

The move categories of our MC-LOA program are similar to the ones used in the Realization-Probability Search of the program MIA [17]. They are applied 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.

This selection strategy is applied only at nodes with a visit count higher than a certain threshold T (here 5) [9]. 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.

For all the children of a current leaf node 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. 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. Experiments conducted in the past revealed that this check improved both the playing and solving strength of the engine.

Play-out. The play-out step begins when we enter a position that is not yet a part of the tree. 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 [10,11]. The main idea is to play interesting moves according to heuristic knowledge. We describe the simulation strategies in detail in the next section.

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 [9]. 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 traversed 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 [9], i.e., $v_L = (\sum_k R_k)/n_L$.

In addition to backpropagating the values $\{1,0,-1\}$, MCTS-Solver also propagates the game-theoretical values ∞ or $-\infty$. 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. 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 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 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 node will be updated according to the backpropagation strategy as described previously.

3.3 Parallelization

The parallel version of our MC-LOA program uses the so-called "single-run" parallelization [6], also called *root parallelization* [8]. It consists of building multiple MCTS trees in parallel, with one thread per tree. These threads do not share information with each other. When the available time is up, all the root children of the separate MCTS trees are merged with their corresponding clones. For each group of clones, the scores of all games played are added. Based on this grand

total, the best move is selected. This parallelization method only requires a minimal amount of communication between threads, so the parallelization is easy, even on a cluster. For a small number of threads, root parallelization performs remarkably well in comparison to other parallelization methods [6,8].

4 Simulation Strategies

In both the selection and the play-out steps move categories together with their associated transition probabilities are used to bias the move selection. In this section we introduce four simulation strategies for further biasing and enhancing the simulation roll-outs. They are *Evaluation Cut-Off*, *Corrective*, *Greedy*, and *Mixed*, and are discussed in detail in Subsect. 4.1 to 4.4, respectively.

4.1 Evaluation Cut-Off

The *Evaluation Cut-Off* strategy stops a simulated game before a terminal state is reached if, according to a heuristic knowledge, the game is judged to be effectively over. In general, once a LOA position becomes quite lopsided, an evaluation function can return a quite trustworthy score, more so than even elaborate simulation strategies. The game can thus be safely terminated both earlier and with a more accurate score than if continuing the simulation (which might, e.g., fail to deliver the win).

We use the MIA 4.5 evaluation function [19] for this purpose. When the evaluation function gives a value that exceeds a certain threshold (e.g., 700 points), the game is scored as a win. Conversely, if the evaluation function gives a value that is below a certain threshold (e.g., -700 points), the game is scored as a loss. For efficiency reasons the evaluation function is called only every 3 plies, starting at the second ply (thus at 2, 5, 8, 11 etc.). This strategy is applied solely in the play-out phase. We remark that a similar strategy was already described by Winands *et al.* in [18]. The Amazons program InvaderMC [13] also terminates simulations early based on an evaluation score. The difference is that in InvaderMC the simulation stops after a fixed length (and subsequently is scored based on the value of the evaluation function), whereas in our approach the simulation may terminate at any time.

4.2 Corrective

One known disadvantage of simulation strategies is that they may draw and play a move which immediately ruins a perfectly healthy position. Embedding domain knowledge, e.g., by the use of Progressive Bias, somewhat alleviates the disadvantage.

In the *Corrective* strategy we use the evaluation function to bias the move selection towards minimizing the risk of choosing an obviously bad move. This is done in the following way. First, we evaluate the position for which we are choosing a move. Next, we generate the moves and scan them to obtain their

```
correctiveStrategy(board){

    defaultValue = evaluate(board);
    moveList = generateMoves();
    scoreSum = 0;

    foreach(Move m in moveList){
        value = evaluate(board, m);
        if (value > bound)
            return m;
        else if (value <= defaultValue)
            m.score = Epsilon;
        else
            m.score = m.getMoveCategoryWeight(board);
        scoreSum += m.score;
    }

    scoreSum = scoreSum*random();
    foreach(Move m in moveList){
        scoreSum -= m.score;
        if(scoreSum <= 0)
            return m;
    }
}
```

Fig. 2. Pseudo code for the Corrective strategy

weights. If the move leads to a successor which has a lower evaluation score than its parent, we set the weight of a move to a preset minimum value (close to zero). If a move leads to a win, it will be immediately played. The pseudo code for this strategy is given in Fig. 2.

4.3 Greedy

In the *Greedy* strategy the evaluation function is more directly applied for selecting moves: the move leading to the position with the highest evaluation score is selected. However, because evaluating every move is time consuming, we evaluate only moves that have a good potential for being the best. For this strategy it means that only the k-best moves according to their transition probabilities are fully evaluated. As in the Evaluation Cut-Off strategy, when a move leads to a position with an evaluation over a preset threshold, the play-out is stopped and scored as a win. Finally, the remaining moves — which are not heuristically evaluated — are checked for a mate. The pseudo code for the Greedy strategy is given in Fig. 3.

4.4 Mixed

A potential weakness of the Greedy strategy is that despite a small random factor in the evaluation function, it is too deterministic. The *Mixed* strategy combines

```
Greedy(Board b){

    moveList = generateMoves();
    assignAndSort(moveList);
    counter = 0;

    foreach(Move m in moveList){
        if(counter < k){
            value = evaluate(board, m);
            if(value > bound){
                return m;
            }
            if(value > max){
                best = m;
                max = value;
            }
        }
        else {
            if(evaluateWin(board, m)) {
                return m;
            }
        }
        counter++;
   }
   return best;
}
```

Fig. 3. Pseudo code for the Greedy strategy

the Corrective strategy and the Greedy strategy. The Corrective strategy is used in the selection step, i.e., at tree nodes where a simulation strategy is needed (i.e., $n < T$), as well as in the first position entered in the play-out step. For the remainder of the play-out the Greedy strategy is applied.

5 Experiments

In this section we evaluate the performance of the four aforementioned simulation strategies by letting them play against themselves and the LOA program MIA. First, we briefly explain MIA in Subsect. 5.1. Next, several settings of the Evaluation Cut-Off strategy are tested in Subsect. 5.2. Subsequently, in order to test the quality of the four simulation strategies, we match them in a round-robin tournament in Subsect. 5.3. Finally, in Subsect. 5.4 we evaluate the playing strength of a MC-based LOA program, using the best settings found in previous subsections, against the $\alpha\beta$-based program MIA.

In the following experiments each match data point represents the result of 1,000 games, with both colors played equally. A standardized set of 100 three-ply starting positions [2] was used, with a small random factor in the evaluation

function preventing games from being repeated. The thinking time was set to 1 second per move. All experiments were performed on an AMD Opteron 2.2 GHz computer.

5.1 MIA

MIA is a world-class LOA program, which won the LOA tournament at the eighth (2003), ninth (2004), and eleventh (2006) Computer Olympiad. It is considered the best LOA-playing entity in the world. All our experiments were performed using the latest and strongest version of the program, MIA 4.5.[1] MIA performs an $\alpha\beta$ depth-first iterative-deepening search in the Enhanced-Realization-Probability-Search framework [17]. The program uses state-of-the-art $\alpha\beta$ enhancements [16].

5.2 Parameter Tuning

In the first series of experiments we tested different cut-off bounds for the Evaluation Cut-Off strategy. For each setting, a program using the Cut-Off strategy played a match against three other programs. The results are given in Table 1. The No-Bound strategy produces moves in the same way as the Evaluation Cut-Off strategy, but always plays simulations out to the end. As we see, such a play-out strategy loses the majority of its games against every setting of the Evaluation Cut-Off strategy: the higher the bound the more the No-Bound strategy loses. Of interest is the relatively good performance of even a bound value of 0, meaning practically that every simulation is cut-off and scored after 2-ply. Nonetheless, under this setting Evaluation Cut-Off still wins most of its games against the No-Bound strategy. The early termination allows more simulations to be performed in the same amount of time, e.g., using a cut-off value of 700 results in over twice as many simulations.

Tuning the bound parameter against a similar (weaker) MC-LOA program may lead to a suboptimal value. Therefore we also played the program against two MIA versions. The first used the MIA III evaluator and the second the MIA 4.5 evaluator. Both versions used the latest search engine. We see in Table 1 that the best bound parameter is somewhere around 600–700. The value of 700 for the bound parameter will be used for the remaining experiments. Moreover, the relative good performance of the 0 setting is again remarkable. What is different from the runs against the No-Bound strategy is that the performance starts to deteriorate when the bound exceeds 1000. Next, the Evaluation Cut-Off strategy significantly improves the way how well simulations do against an $\alpha\beta$ program. For example, whereas the two versions of MIA win 99% of their games against the No-Bound strategy, MIA III and MIA 4.5 only win 22% and 72% of the games against the best settings of the Evaluation Cut-Off strategy. For a proper comparison, the average length and the speed of the play-outs are given in the last two rows of Table 1, respectively.

[1] The program can be found at: http://www.personeel.unimaas.nl/m-winands/loa/

Table 1. 1000-game match results

Bound	0	100	200	400	600	700	800	1000	1200	1400	No-Bound
No-Bound	150.5	149.0	155.5	118.0	112.0	99.0	86.5	57.0	47.0	22.0	X
MIA III	246.0	246.0	227.5	231.5	218.0	253.5	247.5	335.0	398.0	510.5	998
MIA 4.5	794.0	795.0	776.0	768.5	720.0	717.0	754.0	781.0	834.5	868.0	999
Avg. Game Len.	2	2.67	3.56	5.85	8.45	9.83	11.17	13.94	16.65	19.18	53.7
Games per Sec.	10074	9242	8422	6618	5260	4659	4211	3507	2995	2611	2060

5.3 Round-Robin Experiments

In the second series of experiments we quantify the performance of the four simulation strategies in a round-robin tournament. The results are given in Table 2. Surprisingly, the heavily evaluation-function based Greedy strategy was the weakest of the four. The Corrective strategy was better than the Evaluation Cut-Off and Greedy strategy. But, the Mixed strategy, a combination of Corrective and Greedy, outperformed the other ones. The latter result shows that the evaluation function can be directly used for selecting moves as done by Greedy, but not at the start of a simulation. The first moves should be highly randomized.

Table 2. Tournament results

Strategy	Evaluation Cut-Off	Corrective	Greedy	Mixed
Evaluation Cut-Off	-	442.5	598.0	325.5
Corrective	557.5	-	674.0	362.0
Greedy	402.9	326.0	-	167.0
Mixed	674.5	638.0	833.0	-

5.4 MC-LOA vs. MIA

Finally, in the third series of experiments we matched the MC-LOA program using the Mixed strategy against the $\alpha\beta$-based program MIA. The thinking time was set to 5 seconds per move.

The results are given in Table 3. We see from row two that the regular MC-LOA program played almost as well as MIA, receiving a 46% winning score. One nice benefit of MCTS is that it can be parallelized quite easily compared to $\alpha\beta$ search. We tested a two- and four-threaded MC-LOA program against (a single-threaded) MIA and they won 56% and 60%, respectively. We do not have a parallel version of MIA, however, we ran an experiment where the two-threaded MC-LOA program competed against MIA. Here MIA was given 50% more time (simulating a search efficiency increase of 50% if MIA were to be given two processors). A 1,000 game match resulted in a 52% winning percentage for MC-LOA.

It is beyond the scope of paper to investigate to what extent MCTS scales better than $\alpha\beta$. To give an indication, experiments revealed that for 1 second per move MC-LOA won 42% of the games, whereas for 5 seconds per move MC-LOA already won 46% of the games.

Table 3. 1,000-game match results

	Score	Win %	Winning ratio
1 × MC-LOA vs. MIA 4.5	458.0 - 542.0	46%	0.85
2 × MC-LOA vs. MIA 4.5	563.5 - 436.5	56%	1.29
4 × MC-LOA vs. MIA 4.5	602.5 - 397.5	60%	1.52

6 Conclusion and Future Research

In this paper we investigated how to use a positional evaluation function to enhance a simulation-based LOA program. Four different simulation strategies were designed, called Evaluation-Cut Off, Corrective, Greedy, and Mixed.

Our experimental results showed that the Mixed strategy of playing greedily in the play-out phase, but exploring more in the earlier selection phase, although in a way such that it avoids moves that immediately deteriorate the position, works the best. Experiments also showed that applying an evaluation function to stop simulations when a game is judged to be effectively over, resulted in a significant increase in both the number of simulations and playing strength.

Collectively, these enhancements resulted in our simulation-based MC-LOA program to play at a comparable level as the world-class $\alpha\beta$-based program MIA. Moreover, equipped with a simple root parallelization the MC-LOA program outperformed MIA both when using two and four threads. Based on the greatly improved playing strength that we witnessed in the MC-LOA program when adding the enhancements proposed above, we believe that it is only a matter of time until simulation-based programs will significantly outperform $\alpha\beta$-based programs in the game LOA. This is an important milestone for MCTS, because up until now the traditional game-tree search approach has generally been considered to be better suited for the game LOA.

As a future research we plan to continue work on enhancing the simulation strategies both by tuning the various parameters involved and by combining the strategies in more elaborate ways.

References

1. Abramson, B.: Expected-outcome: A general model of static evaluation. IEEE Transactions on Pattern Analysis and Machine Intelligence 12(2), 182–193 (1990)
2. Billings, D., Björnsson, Y.: Search and knowledge in Lines of Action. In: van den Herik, H.J., Iida, H., Heinz, E.A. (eds.) Advances in Computer Games 10: Many Games, Many Challenges, pp. 231–248. Kluwer Academic Publishers, Boston (2003)
3. Bouzy, B., Helmstetter, B.: Monte-Carlo Go Developments. In: van den Herik, H.J., Iida, H., Heinz, E.A. (eds.) Advances in Computer Games 10: Many Games, Many Challenges, pp. 159–174. Kluwer Academic Publishers, Boston (2003)
4. Brügmann, B.: Monte Carlo Go. Technical report, Physics Department, Syracuse University (1993)

5. Cazenave, T., Borsboom, J.: Golois Wins Phantom Go Tournament. ICGA Journal 30(3), 165–166 (2007)
6. Cazenave, T., Jouandeau, N.: On the parallelization of UCT. In: van den Herik, H.J., Uiterwijk, J.W.H.M., Winands, M.H.M., Schadd, M.P.D. (eds.) Proceedings of the Computer Games Workshop 2007 (CGW 2007), Universiteit Maastricht, Maastricht, The Netherlands, pp. 93–101 (2007)
7. Chaslot, G.M.J.-B., Winands, M.H.M., Uiterwijk, J.W.H.M., van den Herik, H.J., Bouzy, B.: Progressive strategies for Monte-Carlo Tree Search. New Mathematics and Natural Computation 4(3), 343–357 (2008)
8. Chaslot, G.M.J.-B., Winands, M.H.M., van den Herik, H.J.: Parallel monte-carlo tree search. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 60–71. Springer, Heidelberg (2008)
9. Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo tree search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M. (eds.) CG 2006. LNCS, vol. 4630, pp. 72–83. Springer, Heidelberg (2007)
10. Finnsson, H., Björnsson, Y.: Simulation-based approach to general game playing. In: Fox, D., Gomes, C.P. (eds.) Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, pp. 259–264 (2008)
11. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: Ghahramani, Z. (ed.) Proceedings of the International Conference on Machine Learning (ICML), pp. 273–280. ACM, New York (2007)
12. Kocsis, L., Szepesvári, C.: Bandit Based Monte-Carlo Planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
13. Lorentz, R.J.: Amazons discover monte-carlo. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 13–24. Springer, Heidelberg (2008)
14. Sackson, S.: A Gamut of Games. Random House, New York (1969)
15. Tsuruoka, Y., Yokoyama, D., Chikayama, T.: Game-tree search algorithm based on realization probability. ICGA Journal 25(3), 132–144 (2002)
16. Winands, M.H.M.: Informed Search in Complex Games. PhD thesis, Universiteit Maastricht, Maastricht, The Netherlands (2004)
17. Winands, M.H.M., Björnsson, Y.: Enhanced realization probability search. New Mathematics and Natural Computation 4(3), 329–342 (2008)
18. Winands, M.H.M., Björnsson, Y., Saito, J.-T.: Monte-carlo tree search solver. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 25–36. Springer, Heidelberg (2008)
19. Winands, M.H.M., van den Herik, H.J.: MIA: a world champion LOA program. In: The 11th Game Programming Workshop in Japan (GPW 2006), pp. 84–91 (2006)
20. Zhang, P., Chen, K.-H.: Monte Carlo Go capturing tactic search. New Mathematics and Natural Computation 4(3), 359–367 (2008)