Monte-Carlo Tree Search in Lines of Action

Mark H.M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito

***Abstract*—The success of Monte-Carlo Tree Search (MCTS) in many games where $\alpha\beta$-search has failed, naturally raises the question whether Monte-Carlo simulations will eventually also outperform traditional game-tree search in game domains where $\alpha\beta$-based search is now successful. The forte of $\alpha\beta$-search are highly-tactical deterministic game domains with a small to moderate branching factor, where efficient yet knowledge-rich evaluation functions can be applied effectively.**

In here we describe a MCTS-based program for playing the game Lines of Action (LOA), which is a highly-tactical slow-progression game exhibiting many of the properties difficult for MCTS. The program uses an improved MCTS variant that allows it to both prove the game-theoretical value of nodes in a search tree and to focus its simulations better using domain knowledge. This results in simulations superior in both handling tactics and ensuring game progression. Using the improved MCTS variant, our program is able to outperform even the world's strongest $\alpha\beta$-based LOA program. This is an important milestone for MCTS because the traditional game-tree search approach has been considered to be the better suited for playing LOA.

***Index Terms*—Monte-Carlo Tree Search, Game-Tree Solver, Lines of Action.**

I. 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 that further enhance its effectiveness. This traditional game-tree-search approach has, however, been less successful for other types of games, in particular where a large branching factor prevents a deep lookahead or the complexity of game state evaluations hinders the construction of an effective evaluation function. Go is an example of a game that has so far eluded this approach.

In recent years a new paradigm for game-tree search has emerged, so-called Monte-Carlo Tree Search (MCTS) [1], [2]. 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 [3], [4], [5], but under the new paradigm MCTS has evolved into a full-fledged best-first search procedure that replaces traditional $\alpha\beta$-based search altogether. Many non-deterministic games lend themselves well to a simulation-based approach (e.g. Scrabble [6] and Skat [7]), in part because of their chance element. In the past few years MCTS has also substantially advanced the state-of-the-art in several deterministic game domains where $\alpha\beta$-based search has had difficulties, in particular computer Go, but other domains include General Game Playing [8], Phantom Go [9], Hex [10], and Amazons [11]. These are, however, all examples of game domains where either a large branching factor or a complex static state evaluation do restrain $\alpha\beta$ search in one way or another.

This remarkable success of MCTS naturally raises the question as to whether simulation-based programs can also compete successfully against traditional game-tree search programs in domains where the latter have been successfully employed and achieved master-level status, that is, deterministic games with a moderate branching factor and knowledge-rich evaluation functions. Clearly some games are more challenging for simulation-based approaches than others. For example, the progression property has been identified as an important success factor for MCTS [12], that is, ideally each move should bring the game closer towards its natural conclusion (e.g. by gradually filling up the board by adding pieces or blocking squares). Without this property there is a risk of the simulations leading mostly to futile results. Also, games with many tactical lines of play that can end the game abruptly (e.g. checkmate in chess) typically lend themselves better to minimax-based backup rules than simulation averaging. It is thus clear that chess-like games, which are both highly tactical and where pieces can be shuffled (endlessly) back and forth without much progress, present a challenge for MCTS.

In this article we describe a MCTS program for playing the game Lines of Action (LOA) [13]. It uses an improved MCTS variant that outperforms the world's best $\alpha\beta$-based LOA program. This is an important milestone for MCTS, because up until now the traditional game-tree search approach has been considered to be better suited for LOA, which is a highly-tactical slow-progression game featuring 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 [14], BING [15], YL [16], and MONA [16], are all $\alpha\beta$ based.

To achieve this success MCTS had to be enhanced in several ways. The enhancements occurred in steps over the last couple of years. First, to be able to more effectively handle highly-tactical lines of play leading to untimely wins or losses, MCTS was augmented such that it can prove the game-theoretical value of nodes in a search tree, where applicable [17]. Secondly, to avoid aimlessly moving pieces back and forth, the program uses simulation strategies that have been enriched in various ways with useful domain knowledge. The informed strategies result in simulations that are both more focused and can vary in length depending on the progress made [18]. Finally, by carrying useful tree information around as the game advances and by fine-tuning various search-control parameters

Mark Winands and Jahn-Takeshi Saito are with the Games and AI Group, Department of Knowledge Engineering, Faculty of Humanities and Sciences, Maastricht University, The Netherlands. (e-mail: {m.winands,j.saito}@maastrichtuniversity.nl)

Yngvi Björnsson is with the School of Computer Science, Reykjavík University, Reykjavík, Iceland (e-mail yngvi@ru.is).

further performance gains are achieved. Collectively these enhancements result in a MCTS variant that outperforms $\alpha\beta$.

The article is organized as follows. In the next section we explain the rules of LOA and the role it plays in artificial-intelligence game research. In Section III we discuss MCTS and its implementation in our LOA program. In Sections IV and V we introduce our game-theoretical MCTS variant and the improved simulation strategies, respectively. We empirically evaluate the MCTS-based LOA program in Section VI and match it against its $\alpha\beta$-based counterpart. Finally, in Section VII we conclude and give an outlook on future research.

II. Lines of Action

Lines of Action (LOA) is a two-person zero-sum game with perfect information; it is a chess-like game (i.e. with pieces that move and can be captured) played on an 8×8 board, albeit with a connection-based goal. LOA was invented by Claude Soucie around 1960. Sid Sackson [13] described the game in his first edition of *A Gamut of Games*.

A. The Rules

LOA is played on an 8×8 board by two sides, Black and White. Each side has twelve (checker) pieces at its disposal. Game play is specified by the following rules:[1]

1) 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 (see Fig. 1(a)).
2) The players alternately move a piece, starting with Black.
3) 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 (see Fig. 1(b)).
4) A player may jump over its own pieces.
5) A player may not jump over the opponent's pieces, but can capture them by landing on them.
6) 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. Connected pieces are on squares that are adjacent, either orthogonally or diagonally (e.g., see Fig. 1(c)). A single piece is a connected unit.
7) In the case of simultaneous connection, the game is drawn.
8) A player that cannot move must pass.
9) If a position with the same player to move occurs for the third time, the game is drawn.

In Fig. 1(b) the possible moves of the black piece on **d3** (using the same coordinate system as in chess) are shown by arrows. The piece cannot move to **f1** because its path is blocked by an opposing piece. The move to **h7** is not allowed because the square is occupied by a black piece.

[1]These are the rules used at the Computer Olympiads and at the MSO World Championships. In some books, magazines or tournaments, there may be a slight variation on rules 2, 7, 8, and 9.

B. Characteristics

The game has an average branching factor of approximately 29 and an average game length of around 44 ply [14]. The game-tree complexity is estimated to be about 10^{64} and the state space complexity 10^{23} [19].

The game is thus comparable to Othello with respect to complexity [20]. Given the current state-of-the-art computer techniques LOA is not solvable by brute-force methods any time soon. A scaled-down 6×6 version was solved by Winands in 2008 [21].

Since most terminal positions have still more than 10 pieces remaining on the board [22], endgame databases are not effectively applicable in LOA (a 10 piece database would require approximately 10 terabytes to store). Apart from endgame databases not being applicable, the same search techniques and enhancements commonly found in chess-playing programs are generally effective in LOA, such as transposition table [23], [24], killer moves [25], adaptive null-move [26], [27], and multi-cut [28], [29].

C. The Role of LOA in AI Game Research

Around 1975 LOA received its first credentials as an AI research topic. Then the first LOA program was written by an unknown author at the Stanford AI laboratory [30]. In the 1980s and 1990s "hobby" programmers wrote several LOA programs, however, all were easily beaten by humans [30]. At the end of the nineties LOA again received increased interest from the games research community.

On the one hand, researchers recognized LOA as a good test domain for their algorithms. For example, Eppstein mentioned evaluation of connectivity of LOA positions as a possible application for his dynamic planar graph techniques [31]. Kocsis successfully applied his time allocation learning algorithms and his new Neural MoveMap move ordering method in LOA [32][33]. Moreover, Björnsson used LOA as an alternative domain (to chess) to verify the merits of his multi-cut pruning method [34]. Donkers used LOA to test opponent-model search [35]. Sakuta *et al.* investigated the application of the killer-tree heuristic and the λ-search method to the endgame of LOA [36]. Hashimoto *et al.* chose LOA as a test domain for their automatic realization-probability search method [37].

One the other hand, researchers concentrated on building strong LOA programs based on both existing and new ideas. For instance, the four programs MIA (Maastricht In Action) [14], Bing [15], YL [16], and Mona [16] are example of strong LOA programs. Since 2000 LOA has been played seven times at the Computer Olympiad, a multi-games event in which all of the participants are computer programs. The strongest LOA programs are considerably stronger than the best human players [38].

III. Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) [1], [2] 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

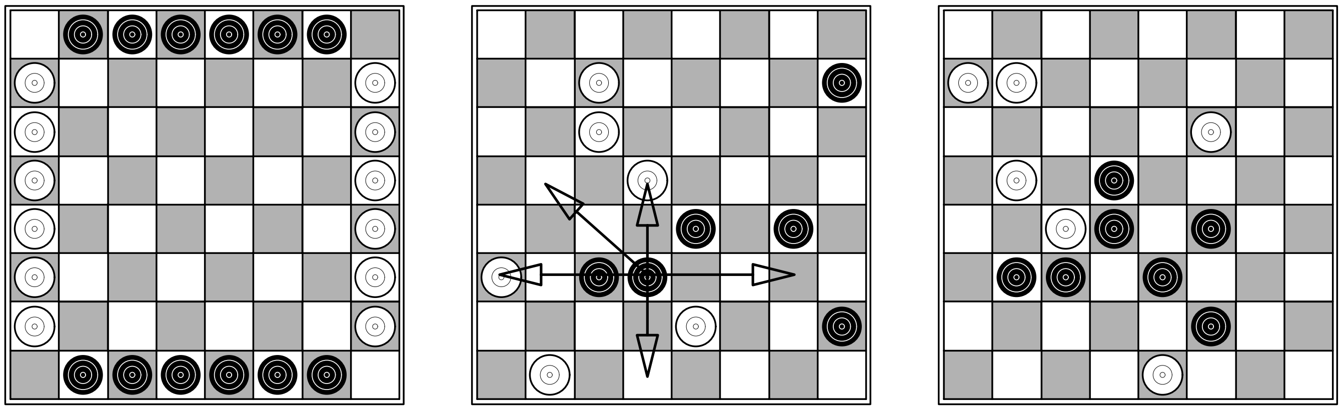

Fig. 1. (a) The initial position. (b) Example of possible moves. (c) A terminal position.

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 [39]. The steps, outlined in Fig. 2, 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, in the *backpropagation step,* R is propagated back along the path from *E* to the root node, adding R to an incrementally computed result average for each action along the way. When time is up, the action played by the program is the child of the root with the highest such average value.

A. The four strategic steps

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

1) Selection: Selection picks a child to be searched based on previous 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 [2], enhanced with Progressive Bias (PB) [39]. PB is a technique to embed domain-knowledge bias into the UCT formula. It is e.g. 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 node p that satisfies Formula 1:

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

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 can be tuned experimentally. $\frac{W \times P_{mc}}{l_i+1}$ is the PB part of the formula. *W* is a constant, which is set manually (here $W = 10$). P_{mc} is the *transition probability* of a move category *mc* [40]. Instead of dividing the PB part by the visit count n_i as done originally [39], it is here divided by the number of losses l_i. In this approach, nodes that do not perform well are not biased too long, whereas nodes that continue to have a high score, continue to be biased. To ensure that we do not divide by 0, a 1 is added in the denominator. Nijssen and Winands [41] tested this approach in the games Focus and Chinese Checkers, showing that PB divided by the number of losses outperformed the default PB in the two-player variants with a winning score of 65% and 58%, respectively. A slight improvement was measured for our MC-LOA program as well.

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 mc is calculated as follows:

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

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 [42]. 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,

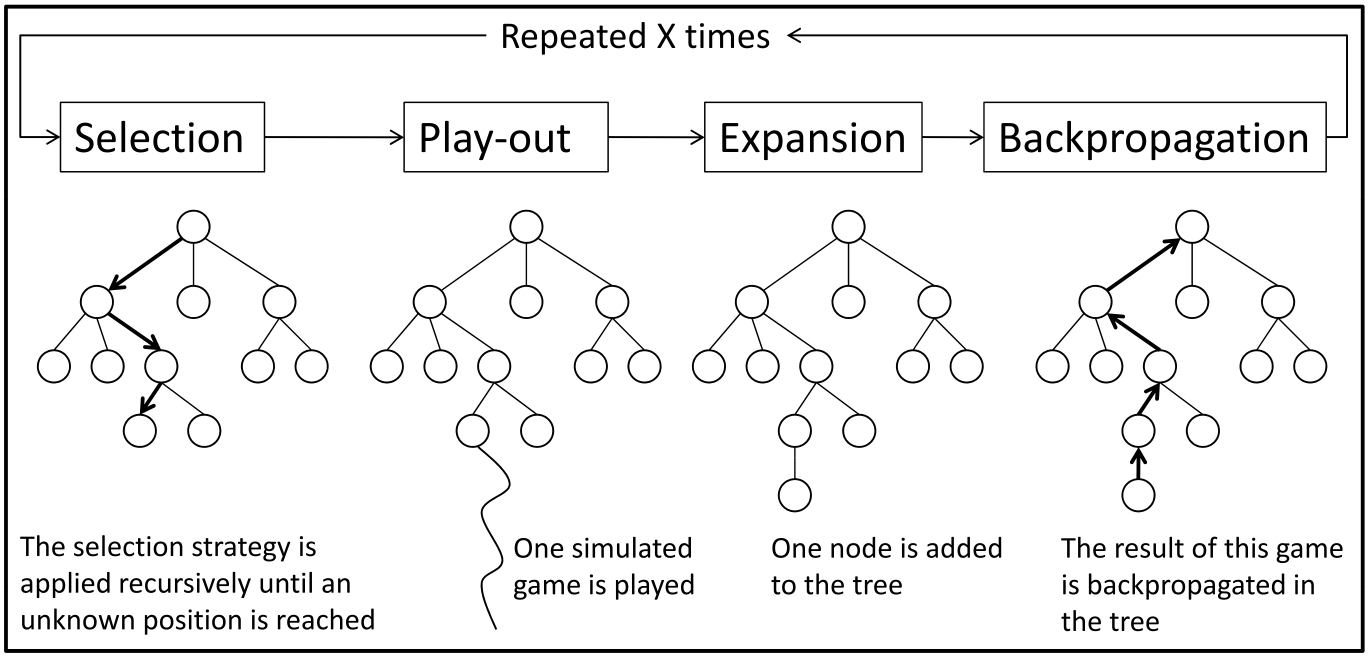
Repeated X times
Selection
Play-out
Expansion
Backpropagation
The selection strategy is applied recursively until an unknown position is reached
One simulated game is played
One node is added to the tree
The result of this game is backpropagated in the tree

Fig. 2. Outline of Monte-Carlo Tree Search (adapted from Chaslot *et al.* [39]).

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.

The aforementioned selection strategy is only applied in nodes with visit count higher than a certain threshold T (here 5) [1]. 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.

2) 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*. Good simulation strategies have the potential to improve the level of play significantly [43]. The main idea is to play interesting moves according to heuristic knowledge. In our MC-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 effectively over. When an evaluation function returns a position assessment that exceeds a certain threshold (i.e., 700 points), which heuristically indicates a decisive advantage, the game is scored as a win. If the evaluation function returns a value that is below a mirror threshold (i.e., -700 points), the game is scored as a loss. For efficiency reasons the evaluation function is called only every 3 plies, determined by trial and error [17].

The idea of early terminations based on an evaluation score is not new. The Amazons program INVADERMC [11] also does so. 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.

3) 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 [1]. The added leaf node L corresponds to the first position encountered during the traversal that was not already stored.

4) 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 [1], i.e., $v_L = (\sum_k R_k)/n_L$.

B. Parallelization

The parallel version of our MC-LOA program uses the so-called "single-run" parallelization [44], also called *root parallelization* [45]. 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 [44], [45]. However, root parallelization does not scale well for a larger

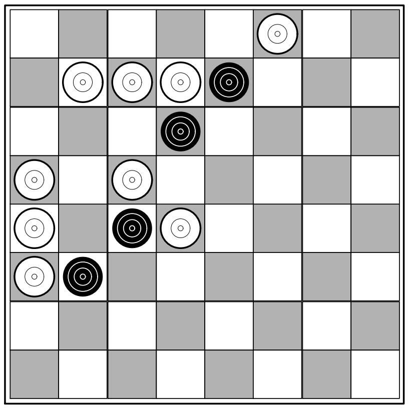

Fig. 3. White to move.

number of threads. An alternative is to use *tree parallelization* [45], which had good results in Computer Go [46], [47]. This method uses one shared tree from which several simulated games are played simultaneously [45].

IV. Monte-Carlo Tree Search Solver

Although MCTS is unable to *prove* the game-theoretical value, in the long run MCTS equipped with the UCT formula is able to *converge* to the game-theoretical value. For example, in endgame positions in fixed termination games like Go or Amazons, MCTS is often able to find the optimal move relatively fast [48], [49]. But in a tactical game like LOA, where the main line towards the winning position is typically narrow with many non-progressing alternatives, 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. 3 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 or longer, it selects the correct move. For a reference, we remark that in this position it takes $\alpha\beta$ less than a second to select the best move and prove the win.

We thus designed a new MCTS variant called MCTS-Solver, which is able to prove the game-theoretical value of a position. The backpropagation and selection steps were modified for this variant, as well as the procedure for choosing the final move to play.

A. Backpropagation

The play-out step returns the values $\{1, 0, -1\}$ for simulations ending in a win, draw, or loss, respectively. In regular MCTS the same is true for terminal positions occurring in the search tree (built by the MCTS expansion step). In the MCTS-Solver, terminal win and loss positions occurring in the tree are handled differently.[2] The win and loss terminal positions are instead assigns ∞ or $-\infty$, respectively. A special provision is then taken when backing such proven values up the tree. There are three cases to consider as shown in Fig. 4 (we use the negamax formulation, alternating signs between levels). First, when a simulation backs up a proven loss ($-\infty$) from a child c to a parent p, the parent node p becomes, and is labelled as, a proven win (∞), that is, the position is won for the player at p because the move played leads to a win (left backup diagram in the figure). When backing up a proven win (∞) from c to p, one must, however, also look at the other children of p to determine p's value. In the second case, when all child nodes of p are also a proven win (∞), then the value of p becomes a proven loss (-∞), because all moves lead to a position lost for p (middle backup diagram in the figure). However, the third case occurs if there exists at least one child with a value different value from a proven win. Then we cannot label p as a proven loss. Instead p gets updates as if a simulation win (instead of a proven win) were being backed up from node c (right backup diagram in the figure; v and u indicate non-proven values). Non-proven values are backed up as in regular MCTS.

[2]Draws are generally more problematic to prove than wins and losses, however, because draws happen only in exceptional cases in LOA we took the decision not to handle them for efficiency reasons.

B. Selection

As seen in the previous subsection, a node can have a proven game-theoretical value of ∞ or $-\infty$. The question arises how these game-theoretical values affect the selection strategy. When entering a node with such a proven value, that value can simply be returned without any selection taking place. A more interesting case is when the node itself has a non-proven value but some of its children have.

Assume that one or more moves of node p are proven to lead to the loss for the player to move in p. It is tempting to discard them in the selection step based on the argument that one would never pick them. However, this can lead to overestimating the value of node p, especially when moves are pseudo-randomly selected by the simulation strategy. For example, in Fig. 5 we have three one-ply subtrees. Leaf nodes *B* and *C* are proven to be a loss (for player to move in *A*), indicated by $-\infty$; the numbers below the other leaves are the *expected* pay-off values (also from the perspective of the player to move in *A*). 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.

Conversely, 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. 5. 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 trials and error, the most effective selection is performed in the following way. In case Formula (1) is applied, moves leading to a loss for the player will never be selected.

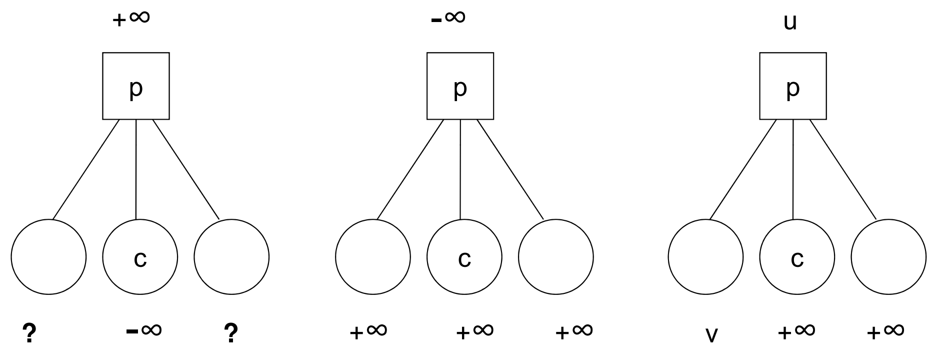

Fig. 4. Backup of proven values.

For nodes that instead select moves according to a simulation strategy, that is nodes having the visit count below the pre-set threshold, moves leading to a loss *can* be selected.

One additional improvement is to perform a 1-ply lookahead at leaf nodes (i.e., where the visit count equals one) [17]. We check whether they lead to a direct win for the player to move. If there is such a move, we can skip the play-out, label the node as a win, and start the backpropagation step. If it were not for such a lookahead, it could take many simulations before a child leading to a mate-in-one is selected and the node proven.

C. 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, is does not matter too much which of the approaches is used given that a sufficient amount of simulations for each root move has been played. However, for MCTS-Solver it does somewhat 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* [39]. 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. For example, if two moves have the same value, we would prefer the one explored less often. The rational has to do with the derivative of their value: because of the imbalance in the number of simulations, either the value of the move more explored must have been dropping, or the value of the one less explored increasing; in both cases the one less explored is to be favored.

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. 3, MCTS-Solver is able to select the best move and prove the win for the position depicted in less than a second, or in the same time frame as $\alpha\beta$. As noted earlier, it takes standard MCTS over a minute to pick the winning move.

D. Pseudo Code for MCTS-Solver

A C-like pseudo code of MCTS-Solver is provided in Fig. 6. The algorithm is constructed similar to negamax in the context of minimax search. `select(Node N)` is the selection function as discussed in Subsection IV-B, 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.

V. Improved 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*.

A. 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 gets very lopsided, an evaluation function can return a quite trustworthy score, more so than even elaborate simulation strategies. The game can thus be (relatively) safely terminated both earlier and with a more accurate score than if continuing the simulation (which might e.g. fail to deliver the win). This is somewhat analogous to the ”mercy-rule” in computer Go [50]. We use the MIA 4.5 evaluation function [51] for this purpose. When the evaluation function gives a value that exceeds a certain threshold, the game is scored as a win. Conversely, if the evaluation function gives a value that is below the negated threshold, the game is scored as a loss.

Our initial MCTS-based LOA program described in [17], used a threshold value of 1000 points, chosen conservatively as (by observation) such a high value, with only a few exceptions, represents an eventual win. Such a conservative choice of a threshold is not necessarily optimal. It might be a better choice to use a more aggressive cut-off threshold even though being occasionally wrong. The added number of simulations because of even earlier terminations of lopsided positions might more than offset the errors introduced by the occasional erroneous

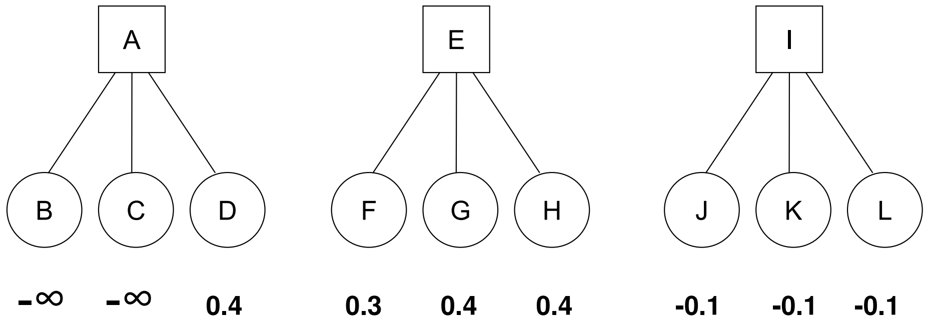

Fig. 5. Monte-Carlo Subtrees.

```
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. 6. Pseudo code for MCTS-Solver

termination decisions. In our improved evaluation cut-off strategy we determine this tradeoff empirically (see experimental section), leading to a substantially more aggressive threshold settings, i.e., 700 points. As before, the termination strategy is applied only in the play-out step. For efficiency reasons the evaluation function is called only every 3 plies, starting at the second ply (thus at 2, 5, 8, 11 etc.). Differences in odd vs. even ply evaluations observed in some LOA programs are not too important here, because they are typically relatively small compared to the large threshold value, as well as they are (partially) offset in the evaluation function of our LOA program by having a side-to-move bonus [14].

B. 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 problem.

In the *Corrective* strategy we use the evaluation function to further 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 get their 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. 7. The effectiveness of the algorithm will be partially determined by how efficiently game positions and moves are evaluated. For a reference, in our MCTS LOA program, using this strategy, evaluating positions consumes around 30% of the program's total execution time (somewhat more than the combined make/undo move operations), whereas determining a weight for a move category takes around 5% of the total execution time.

C. 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

```
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.getMCWeight(board);
        scoreSum += m.score;
    }

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

Fig. 7. Pseudo code for the Corrective strategy

```
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. 8. Pseudo code for the Greedy strategy

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. 8.

D. *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 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. Finding the right balance between exploitation and exploration, however, remains one of the main challenges in simulation-based search. Whereas the mixed strategy proposed here does a good job in our test-domain, more work is still needed for the approach to be applied to other game domains in a principled way.

VI. Experiments

In this section we evaluate the performance of the improved MCTS LOA player, both via self-play and against the world's strongest $\alpha\beta$-based LOA program, MIA 4.5 (as well as some of its earlier ancestors).

We will refer to the MCTS player as MC-LOA. It can be instantiated using the various combinations of enhancements introduced in earlier sections. We use a three-tuple $(solver, threshold, strategy)$ to represent the parameter setting used in each particular player instance, where $solver \in \{on, off\}$, $threshold \in [0, \infty]$, and $strategy \in \{default, corrective, greedy, mixed\}$. For example, in the following experiments the most common instantiation, referring to the best setting we found, is MC-LOA$_{(on, 700, mixed)}$, that is, the solver is enabled, the simulation cut-off threshold is set to 700, and the mixed simulation strategy is used.

To determine the relative playing strength of two programs we play a match between them consisting of many games (to establish a statistical significance). In the following experiments each match data point represents the result of 1,000 games (unless otherwise specified), with both colors played equally. A standardized set of 100 three-ply starting positions [16] is used, with a small random factor in the evaluation function preventing games from being repeated. The thinking time is 5 seconds per move (unless otherwise specified). All experiments were performed on an AMD Opteron 2.2 GHz computer.

In the next subsection we briefly describe MIA 4.5. Then, in turn, we empirically evaluate the simulation strategies, the solver, and then additional tuning enhancements.

A. *MIA (Maastricht In Action)*

MIA is a world-class LOA program, which won the LOA tournament at the eighth (2003), ninth (2004), and eleventh (2006) Computer Olympiad. Over its lifespan of 10 years it has gradually been improved and has for years now been generally accepted as the best LOA-playing entity in the world. All our

TABLE I
THREE DIFFERENT OPPONENTS PLAYING AGAINST MC-LOA$_{(on,\mathbf{t},default)}$ (WIN %)

Threshold (t)	0	100	200	400	600	700	800	1000	1200	1400	∞
MC-LOA$_{(on,\infty,def)}$	15.1 ± 2.2	14.9 ± 2.2	15.6 ± 2.2	11.8 ± 2.0	11.2 ± 1.9	9.9 ± 1.9	8.7 ± 1.7	5.7 ± 1.4	4.7 ± 1.3	$\mathbf{2.2 \pm 0.9}$	-
MIA III	24.6 ± 2.7	24.6 ± 2.7	22.8 ± 2.6	23.2 ± 2.6	$\mathbf{21.8 \pm 2.6}$	25.4 ± 2.7	24.8 ± 2.7	33.5 ± 2.9	39.8 ± 3.0	51.1 ± 3.1	99.8 ± 0.3
MIA 4.5	79.4 ± 2.6	79.5 ± 2.6	77.6 ± 2.6	76.9 ± 2.6	72.0 ± 2.7	$\mathbf{71.7 \pm 2.8}$	75.4 ± 2.7	78.1 ± 2.6	83.5 ± 2.3	86.8 ± 2.1	99.9 ± 0.2
Avg. Game Len.	2.00	2.67	3.56	5.85	8.45	9.83	11.17	13.94	16.65	19.18	53.70
Games per Sec.	10074	9242	8422	6618	5260	4659	4211	3507	2995	2611	2060

experiments were performed using the latest version of the program, called MIA 4.5. The program is written in Java.[3]

MIA performs an $\alpha\beta$ depth-first iterative-deepening search in the Enhanced-Realization-Probability-Search (ERPS) framework [42]. A *two-deep* transposition table [23] is applied to prune a subtree or to narrow the $\alpha\beta$ window. At all interior nodes that are more than 2 plies away from the leaves, it generates all moves to perform Enhanced Transposition Cutoffs (ETC) [24]. Next, a null-move [26] is performed adaptively [27]. Then, an enhanced multi-cut is performed [28], [29]. For move ordering, the move stored in the transposition table (if applicable) is always tried first, followed by two killer moves [25]. These are the last two moves that were best, or at least caused a cutoff, at the given depth. Thereafter follow: (1) capture moves going to the inner area (the central 4×4 board) and (2) capture moves going to the middle area (the 6×6 rim). All the remaining moves are ordered decreasingly according to the relative history heuristic [52]. At the leaf nodes of the regular search, a quiescence search is performed to get more accurate evaluations. For additional details on the search engine and the evaluation function used in MIA, we refer to the Ph.D. thesis *Informed Search in Complex Games* [14].

ERPS is applied in MIA in the following way. First, moves are classified as captures or non-captures. Next, moves are further sub-classified based on the origin and destination of the move's from and to 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 in the game according to this classification.

B. Evaluation Cut-Off Threshold

The first set of experiments was designed to determine a good cut-off threshold for the Evaluation Cut-Off strategy. MC-LOA$_{(on,t,default)}$ with different cut-off threshold values for t was matched against three other programs: MC-LOA$_{(on,\infty,default)}$ (essentially never terminating simulations early), MIA 4.5, and finally, to get a more variety of opponents, an older version of MIA called MIA III, which uses a somewhat less sophisticated evaluation function. In this experiment, the thinking time was set to 1 second per move.

The results are given in Table I, showing the winning percentage of the players against MC-LOA$_{(on,t,default)}$ using various thresholds. The best threshold setting t against each of the players is the one that minimizes their winning percentage (shown in bold). Based on this we chose a threshold $t = 700$ for our default player, as a compromise between the three different optimal thresholds, with more weight put on the thresholds performing well against the $\alpha\beta$-based opponents. It is worth to note that the value $t = 700$ performs significantly better against these opponents than the value of $t = 1000$ used by the MCTS-based LOA program described in [17].

Several other things of interest can be read from the table. First, it can clearly be seen how important a termination threshold is for MCTS-based LOA programs, as a player without one, as the first line shows, stands a little chance. Second, it is interesting to contrast how well the two MIA versions perform. The MC-LOA$_{(on,t,default)}$ program handily beats MIA III when using appropriate cut-off thresholds, but is not able to match the strong MIA 4.5 program. The MIA 4.5 evaluation function is apparently much stronger than (the already strong) older one, and showcases the importance of a good evaluation function in the game of LOA. Finally, the last two rows of the table give us some insights into how the threshold value affects the average simulation length and number of simulations per time unit, respectively.

TABLE II
ROUND-ROBIN TOURNAMENT RESULTS MATCHING DIFFERENT SIMULATION STRATEGIES, MC-LOA$_{(on,700,\mathbf{e})}$ (WIN %)

Strategy	Default	Corrective	Greedy	Mixed
Default	-	44.25 ± 3.1	59.80 ± 3.0	32.55 ± 2.9
Corrective	55.75 ± 3.1	-	67.40 ± 2.9	36.20 ± 3.0
Greedy	40.20 ± 3.0	32.60 ± 2.9	-	16.70 ± 2.3
Mixed	67.45 ± 2.9	63.80 ± 3.0	83.30 ± 2.3	-

C. Simulation Strategies

In the second set of experiments we quantify the performance of the *Corrective*, *Greedy*, and *Mixed* simulation strategies introduced in Sect. V, as well as that of a default strategy (where the three aforementioned strategies are all disabled). All the strategies, including the default one, use the threshold setting of $t = 700$ determined in the previous subsection. For this experiment, the thinking time was set to 1 second per move.

The result of a round-robin tournament is given in Table II. Somewhat surprisingly, the heavily evaluation-function based Greedy strategy is the weakest of the four, including the default one. The Corrective strategy is better than both the default and the Greedy strategy. But, the Mixed strategy, the combination of Corrective and Greedy, outperforms all the others convincingly. This shows that the evaluation function

[3]A Java program executable and test sets can be found at: http://www.personeel.unimaas.nl/m-winands/loa/.

can be directly used for selecting moves as done by Greedy, but not at the start of a simulation. The first moves should rather be highly randomized.

D. Solver

Having determined the most promising settings for the simulation strategies, we now evaluate the solver's effectiveness in combination with these strategies. The tactical performance of MC-LOA$_{(on,700,mixed)}$ was contrasted to that of the highly sophisticated variable-depth $\alpha\beta$ search of MIA 4.5 (default), as well as to a non-variable-depth search (classic). The classic variant, unlike the default one, does not use ERPS, null-move search nor multi-cut. We measure the effort it takes the programs to solve selected endgame positions in terms of both nodes and CPU time. For MC-LOA, all children at a leaf node evaluated for the termination condition during the search are counted (see subsection IV-B). For the $\alpha\beta$ variants, nodes at depth *i* are counted only during the first iteration that the level is reached. This is how node counting was done in analogous comparisons for other games in [53]. The maximum number of nodes the programs are allowed to search on each problem is 10,000,000. The test-set consists of 488 forced-win LOA positions.[4]

In Table III the results are presented. From the second and third columns we see that MC-LOA$_{(on,700,mixed)}$ outperforms classic $\alpha\beta$ both in terms of positions solved and nodes expanded (although not CPU time). The default $\alpha\beta$ variant, however, outperforms both the others by a large margin in terms of all measures. The node expansions and CPU times are reported only for the subset of positions all three algorithms were able to solve (257 positions) to allow a fairer comparison. Note that it serves no purpose to experiment with MC-LOA without the solver code enabled on the test set, as such a variant is unable to prove any terminal values.

TABLE III
SOLVING PERFORMANCE OF MC-LOA$_{(on,700,mixed)}$ VS. $\alpha\beta$ ON 488 FORCED WIN ENGAME POSITIONS.

Program	488 positions	257 positions	
	# solved	Nodes	Time (ms.)
MC-LOA$_{(on,700,mixed)}$	319	315,900,579	1,373,393
Classic $\alpha\beta$	288	407,975,053	809,866
Default $\alpha\beta$	454	81,349,671	122,640

We can, however, investigate how turning the solver off affects the program's overall playing strength. We do so both for self-play and against MIA 4.5. The results are shown in Table IV. Not only does the MC-LOA program with the solver enabled beat the one with it disabled with almost 54% winning rate, but it also fares much better against MIA 4.5 (scoring close to 47% as opposed to just over 39%). This shows that the ability to prove game-theoretical values of game positions is important in a tactical game like LOA.

[4]The test set is available at www.personeel.unimaas.nl/m-winands/loa/tscg2002a.zip.

TABLE IV
TOURNAMENT RESULTS MC-LOA$_{(\mathbf{s},700,mixed)}$ (WIN %). EACH DATA POINT IS BASED ON A 2000-GAME MATCH.

Strategy	off	on	MIA 4.5
MC-LOA$_{(off,700,mixed)}$	-	46.05 ± 2.2	39.38 ± 2.1
MC-LOA$_{(on,700,mixed)}$	53.95 ± 2.2	-	46.93 ± 2.2
MIA 4.5	60.62 ± 2.1	53.07 ± 2.2	-

E. Parallelization and Tuning Enhancements

The MC-LOA program using the best derived set of parameters, i.e., MC-LOA$_{(on,700,mixed)}$, is performing close to the level of the world-class $\alpha\beta$-based program MIA 4.5, although coming up a little short (47%).

One nice benefit of MCTS is that it can be parallelized quite easily compared to $\alpha\beta$ search. We have a multi-threaded version of our MC-LOA program. For curiosity we matched two- and four-threaded versions of our MC-LOA program against (a single-threaded) MIA 4.5.

TABLE V
PARALLEL MC-LOA$_{(on,700,mixed)}$ VS. MIA 4.5 (WIN %)

Matched Programs	win %
$1 \times$ MC-LOA$_{(on,700,mixed)}$ vs. MIA 4.5	46.93 ± 3.1
$2 \times$ MC-LOA$_{(on,700,mixed)}$ vs. MIA 4.5	56.35 ± 3.1
$4 \times$ MC-LOA$_{(on,700,mixed)}$ vs. MIA 4.5	60.25 ± 3.0

The results are shown in in Table V. We see that the multi-threaded version of MC-LOA handily outperforms the single-threaded MIA 4.5. Unfortunately, there does not exist a multi-threaded version of MIA 4.5 to compare with, as this does not represent a fair comparison. However, to get some idea how a multi-threaded MIA 4.5 might perform we reran the match against the two-threaded MC-LOA, but this time giving MIA 4.5 50% more deliberation 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. Although this type of experiment can give us some insights as of how a multi-threaded MIA 4.5 might perform, nonetheless, based on the experiment's ad-hoc nature we do not feel comfortable drawing firm conjectures about the performance of a hypothetical multi-threaded MIA 4.5 program.

One advantage MIA 4.5 has over its MCTS-based counterpart is having been around for many more years, thus being far more carefully tuned based on years of tournament experience.

To somewhat offset this advantage we took some extra time to further tune our MC-LOA player. By doing the tuning independently afterwards, after having run all the other experiments, we can better demonstrate the potentials such a tuning phase has for improving playing strength. We refer to the more carefully tuned player as MC-LOA-T. Two minor changes were incorporated: (1) between moves, we recycle the relevant part of the MCTS tree [54]; and, (2) instead of dividing the Progressive Bias part by $l_i + 1$ (Formula 1) we divide it by $\sqrt{l_i + 1}$, effectively making the Progressive Bias more relevant.

The result of playing MC-LOA-T against MIA 4.5 is given in Table VI (for a comparison we repeat the result of MC-LOA

TABLE VI
TUNING MC-LOA$_{(on,700,mixed)}$. 2,000-GAME MATCH RESULTS

	win %
MC-LOA$_{(on,700,mixed)}$ vs. MIA 4.5	46.93 ± 2.2
MC-LOA-T$_{(on,700,mixed)}$ vs. MIA 4.5	52.38 ± 2.2

vs MIA 4.5). By relatively little tuning effort we were able to elevate the program's score against MIA 4.5 by more than five percentage points. Now, instead of being slightly behind, the better tuned variant outperforms MIA 4.5 and, although the winning margin is small, it is nonetheless statistically significant using a confidence margin of 95%.

This is an important milestone for MCTS because the traditional game-tree search approach has been considered to be the better suited for playing LOA. We are in the early stages of tuning our MC-LOA player, and with added experience we believe that there are still more strength improvements to be had.

VII. Conclusion and Future Research

In this article we described MC-LOA, a MCTS-based program for playing the game of LOA. The program uses a highly effective MCTS variant that has been imbued with numerous enhancements.

First, the simulations were augmented such that game-theoretical win and loss values could be proved when encountered in the search tree. This required modifications to the backpropagation and selection steps of MCTS, as well as the procedure for picking the final move to play. Secondly, the program uses simulation strategies enriched with useful domain knowledge in various new ways. Modifications were made to both the selection and the play-out steps. The informed strategies resulted in simulations that were more focused. In particular, a mixed strategy of exploring more early on and playing more greedily later on in a simulation seemed to work best. Finally, by carrying useful tree information around as the game advances and by fine-tuning various search-control parameters further performance gains were possible. Collectively these enhancements resulted in a MCTS variant that outperforms even the world's best $\alpha\beta$-based LOA player.

This success is remarkable, because not only is the game of LOA highly tactical, but also slowly progressing. Both these characteristics have traditionally been considered particularly problematic for MCTS-based players. This work thus represents an important milestone for MCTS.

As for future research directions we plan to further work on enhancing the new simulation strategies, e.g., by combining them in more elaborate ways. Of interest too is to use the MCTS and $\alpha\beta$ algorithms in combination in the LOA program, as the latter is still superior in endgame play. Also of a general interest, although not practical in LOA, is to improve the ability of MCTS to prove ties. Finally, we are still in early stages of tuning our MC-LOA player, and with added experience (e.g., from tournament play against other strong LOA programs) we believe that there are still further strength improvements possible.

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, and by a grant from The Icelandic Centre for Research (RANNIS).

References

[1] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *Computers and Games (CG 2006)*, ser. Lecture Notes in Computer Science (LNCS), H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, Eds., vol. 4630. Springer-Verlag, Heidelberg, Germany, 2007, pp. 72–83.

[2] L. Kocsis and C. Szepesvári, "Bandit Based Monte-Carlo Planning," in *Machine Learning: ECML 2006*, ser. Lecture Notes in Artificial Intelligence, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., vol. 4212, 2006, pp. 282–293.

[3] B. Abramson, "Expected-outcome: A general model of static evaluation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 2, pp. 182–193, 1990.

[4] B. Bouzy and B. Helmstetter, "Monte-Carlo Go Developments," in *Advances in Computer Games 10: Many Games, Many Challenges*, H. J. van den Herik, H. Iida, and E. A. Heinz, Eds. Kluwer Academic Publishers, Boston, MA, USA, 2003, pp. 159–174.

[5] B. Brügmann, "Monte Carlo Go," Physics Department, Syracuse University, Tech. Rep., 1993.

[6] B. Sheppard, "World-championship-caliber Scrabble," *Artificial Intelligence*, vol. 134, no. 1–2, pp. 241–275, 2002.

[7] M. Buro, J. R. Long, T. Furtak, and N. R. Sturtevant, "Improving state evaluation, inference, and search in trick-based card games," in *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence*, C. Boutilier, Ed., Pasadena, CA, USA, 2009, pp. 1407–1413.

[8] H. Finnsson and Y. Björnsson, "Simulation-based approach to General Game Playing," in *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008*, D. Fox and C. Gomes, Eds. AAAI Press, 2008, pp. 259–264.

[9] T. Cazenave and J. Borsboom, "Golois Wins Phantom Go Tournament," *ICGA Journal*, vol. 30, no. 3, pp. 165–166, 2007.

[10] T. Cazenave and A. Saffidine, "Utilisation de la recherche arborescente Monte-Carlo au Hex," *Revue d'Intelligence Artificielle*, vol. 23, no. 2–3, pp. 183–202, 2009, in French.

[11] R. Lorentz, "Amazons discover Monte-Carlo," in *Computers and Games (CG 2008)*, ser. Lecture Notes in Computer Science (LNCS), H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, Eds., vol. 5131, 2008, pp. 13–24.

[12] H. Finnsson and Y. Björnsson, "Simulation control in General Game Playing agents," in *GIGA'09 The IJCAI Workshop on General Game Playing*, July 2009.

[13] S. Sackson, *A Gamut of Games*. Random House, New York, NY, USA, 1969.

[14] M. H. M. Winands, "Informed search in complex games," Ph.D. dissertation, Maastricht University, Maastricht, The Netherlands, 2004.

[15] B. Helmstetter and T. Cazenave, "Architecture d'un programme de Lines of Action," in *Intelligence Artificielle et Jeux*, T. Cazenave, Ed. Hermes Science, 2006, pp. 117–126, in French.

[16] D. Billings and Y. Björnsson, "Search and knowledge in Lines of Action," in *Advances in Computer Games 10: Many Games, Many Challenges*, H. J. van den Herik, H. Iida, and E. A. Heinz, Eds. Kluwer Academic Publishers, Boston, MA, USA, 2003, pp. 231–248.

[17] M. H. M. Winands, Y. Björnsson, and J.-T. Saito, "Monte-Carlo Tree Search Solver," in *Computers and Games (CG 2008)*, ser. Lecture Notes in Computer Science (LNCS), H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, Eds., vol. 5131. Springer, 2008, pp. 25–36.

[18] M. H. M. Winands and Y. Björnsson, "Evaluation function based Monte-Carlo LOA," in *Advances in Computer Games Conference (ACG 2009)*, ser. Lecture Notes in Computer Science (LNCS), H. J. van den Herik and P. Spronck, Eds., vol. 6048. Springer, 2010, pp. 33–44.

[19] M. H. M. Winands, J. W. H. M. Uiterwijk, and H. J. van den Herik, "The quad heuristic in Lines of Action," *ICGA Journal*, vol. 24, no. 1, pp. 3–15, 2001.

[20] L. V. Allis, H. J. van den Herik, and I. S. Herschberg, "Which games will survive?" in *Heuristic Programming in Artificial Intelligence 2: the Second Computer Olympiad*, D. N. L. Levy and D. F. Beal, Eds. Ellis Horwood, Chichester, England, 1991, pp. 232–243.

[21] M. H. M. Winands, "6 × 6 LOA is Solved," *ICGA Journal*, vol. 31, no. 3, pp. 234–238, 2008.

[22] ——, "Analysis and implementation of Lines of Action," Master's thesis, Maastricht University, Maastricht, The Netherlands, 2000.

[23] D. M. Breuker, J. W. H. M. Uiterwijk, and H. J. van den Herik, "Replacement schemes and two-level tables," *ICCA Journal*, vol. 19, no. 3, pp. 175–180, 1996.

[24] J. Schaeffer and A. Plaat, "New advances in alpha-beta searching," in *Proceedings of the 1996 ACM 24th Annual Conference on Computer Science*. ACM Press, New York, NY, USA, 1996, pp. 124–130.

[25] S. Akl and M. Newborn, "The principal continuation and the killer heuristic," in *1977 ACM Annual Conference Proceedings*. ACM Press, New York, NY, USA, 1977, pp. 466–473.

[26] C. Donninger, "Null move and deep search: Selective-search heuristics for obtuse chess programs," *ICCA Journal*, vol. 16, no. 3, pp. 137–143, 1993.

[27] E. A. Heinz, "Adaptive null-move pruning," *ICCA Journal*, vol. 22, no. 3, pp. 123–132, 1999.

[28] Y. Björnsson and T. A. Marsland, "Risk managament in game-tree pruning," *Information Sciences*, vol. 122, no. 1, pp. 23–41, 2001.

[29] 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*, vol. 175, no. 4, pp. 315–329, 2005.

[30] D. Dyer, "Lines of Action homepage," 2000. [Online]. Available: http://www.andromeda.com/people/ddyer/loa/loa.html

[31] D. Eppstein, "Dynamic Connectivity in Digital Images," *Information Processing Letters*, vol. 62, no. 3, pp. 121–126, May 1997.

[32] L. Kocsis, J. W. H. M. Uiterwijk, and H. J. van den Herik, "Learning time allocation using neural networks," in *Computers and Games (CG 2000)*, ser. Lecture Notes in Computer Science (LNCS), T. A. Marsland and I. Frank, Eds., vol. 2063. Berlin, Germany: Springer-Verlag, 2001, pp. 170–185.

[33] L. Kocsis, J. Uiterwijk, and H. van den Herik, "Move ordering using neural networks," in *Engineering of Intelligent Systems*, ser. Lecture Notes in Artificial Intelligence (LNCS), L. Montosori, J. Váncza, and M. Ali, Eds. Springer-Verlag, Berlin, Germany, 2001, vol. 2070, pp. 45–50.

[34] Y. Björnsson, "Selective depth-first game-tree search," Ph.D. dissertation, University of Alberta, Edmonton, Canada, 2002.

[35] H. H. L. M. Donkers, J. W. H. M. Uiterwijk, and H. J. van den Herik, "Admissibility in opponent-model search," *Information Sciences*, vol. 154, no. 3-4, pp. 119–140, 2003.

[36] M. Sakuta, T. Hashimoto, J. Nagashima, J. W. H. M. Uiterwijk, and H. Iida, "Application of the killer-tree heuristic and the lamba-search method to Lines of Action," *Information Sciences*, vol. 154, no. 3–4, pp. 141–155, 2003.

[37] T. Hashimoto, J. Nagashima, M. Sakuta, J. W. H. M. Uiterwijk, and H. Iida, "Automatic realization-probability search," 2003, internal report, Dept. of Computer Science, University of Shizuoka, Hamamatsu, Japan.

[38] "Lines of Action Wikipage." [Online]. Available: http://en.wikipedia.org/wiki/Lines_of_Action

[39] 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*, vol. 4, no. 3, pp. 343–357, 2008.

[40] Y. Tsuruoka, D. Yokoyama, and T. Chikayama, "Game-tree search algorithm based on realization probability," *ICGA Journal*, vol. 25, no. 3, pp. 132–144, 2002.

[41] J. A. M. Nijssen and M. H. M. Winands, "Enhancements for multi-player Monte-Carlo Tree Search," in *Computers and Games (CG 2010)*, H. J. van den Herik, H. Iida, and A. Plaat, Eds., 2010.

[42] M. H. M. Winands and Y. Björnsson, "Enhanced realization probability search," *New Mathematics and Natural Computation*, vol. 4, no. 3, pp. 329–342, 2008.

[43] S. Gelly and D. Silver, "Combining online and offline knowledge in UCT," in *Proceedings of the International Conference on Machine Learning (ICML)*, Z. Ghahramani, Ed. ACM, 2007, pp. 273–280.

[44] T. Cazenave and N. Jouandeau, "On the parallelization of UCT." in *Proceedings of the Computer Games Workshop 2007 (CGW 2007)*, H. J. van den Herik, J. W. H. M. Uiterwijk, M. H. M. Winands, and M. P. D. Schadd, Eds. Universiteit Maastricht, Maastricht, The Netherlands, 2007, pp. 93–101.

[45] G. M. J.-B. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel Monte-Carlo Tree Search," in *Computers and Games (CG 2008)*, ser. Lecture Notes in Computer Science (LNCS), H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, Eds., vol. 5131, 2008, pp. 60–71.

[46] S. Gelly, J.-B. Hoock, A. Rimmel, O. Teytaud, and Y. Kalemkarian, "The parallelization of Monte-Carlo planning - parallelization of MC-planning," in *Proceedings of the Fifth International Conference on Informatics in Control, Automation and Robotics, Intelligent Control Systems and Optimization (ICINCO 2008)*, J. Filipe, J. Andrade-Cetto, and J.-L. Ferrier, Eds. INSTICC Press, 2008, pp. 244–249.

[47] M. Enzenberger and M. Müller, "A lock-free multithreaded Monte-Carlo Tree Search algorithm," in *Advances in Computer Games (ACG 2009)*, ser. Lecture Notes in Computer Science (LNCS), H. J. van den Herik and P. H. M. Spronck, Eds., vol. 6048. Berlin Heidelberg, Germany: Springer-Verlag, 2010, pp. 14–20.

[48] P. Zhang and K. Chen, "Monte-Carlo Go tactic search," in *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, P. Wang *et al.*, Eds. World Scientific Publishing Co. Pte. Ltd., 2007, pp. 662–670.

[49] J. Kloetzer, H. Iida, and B. Bouzy, "A comparative study of solvers in Amazons endgames," in *Computational Intelligence and Games (CIG 2008)*. IEEE, 2008, pp. 378–384.

[50] B. Bouzy, "Old-fashioned computer Go vs Monte-Carlo Go," in *IEEE Symposium on Computational Intelligence in Games (CIG-07)*, 2007, invited tutorial.

[51] 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)*, 2006, pp. 84–91.

[52] M. H. M. Winands, E. C. D. van der Werf, H. J. van den Herik, and J. W. H. M. Uiterwijk, "The relative history heuristic," in *Computers and Games (CG 2004)*, ser. Lecture Notes in Computer Science (LNCS), H. J. van den Herik, Y. Björnsson, and N. S. Netanyahu, Eds., vol. 3846. Berlin, Germany: Springer-Verlag, 2006, pp. 262–272.

[53] L. V. Allis, "Searching for solutions in games and artificial intelligence," Ph.D. dissertation, Rijksuniversiteit Limburg, Maastricht, The Netherlands, 1994.

[54] J. Steinhauer, "Monte-Carlo TwixT," Master's thesis, Maastricht University, Maastricht, The Netherlands, 2010.

Mark Winands received the Ph.D. degree in Artificial Intelligence from the Department of Computer Science, Maastricht University, Maastricht, The Netherlands, in 2004. Currently, he is an Assistant Professor at the Department of Knowledge Engineering, Maastricht University. His research interests include heuristic search, machine learning and games. Dr. Winands regularly serves on program committees of major AI and computer games conferences. Since January 2009, he is a member of the editorial board of the ICGA Journal.

Yngvi Björnsson is an associate professor at the School of Computer Science, Reykjavík University and a director (and co-founder) of the CADIA research lab. He received a Ph.D in computer science from the Department of Computing Science, University of Alberta, Canada, in 2002. His research interests are in heuristic search methods and search-control learning, and the application of such techniques for solving large-scale problems in a wide range of problem domains, including computer games and industrial process optimization.

Jahn-Takeshi Saito received a Master's degree in Computational Linguistics and Artificial Intelligence from University of Osnabrück, Osnabrück, Germany, in 2005. He has been working as a Ph.D. student at the Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands, since 2005. His research is on Proof-Number Search and Monte-Carlo methods applied to board games.