$\alpha\beta$-based Play-outs in Monte-Carlo Tree Search

Mark H.M. Winands   Yngvi Björnsson

***Abstract*— Monte-Carlo Tree Search (MCTS) is a recent paradigm for game-tree search, which gradually builds a game-tree in a best-first fashion based on the results of randomized simulation play-outs. The performance of such an approach is highly dependent on both the total number of simulation play-outs and their quality. The two metrics are, however, typically inversely correlated — improving the quality of the play-outs generally involves adding knowledge that requires extra computation, thus allowing fewer play-outs to be performed per time unit. The general practice in MCTS seems to be more towards using relatively knowledge-light play-out strategies for the benefit of getting additional simulations done.**

In this paper we show, for the game Lines of Action (LOA), that this is not necessarily the best strategy. The newest version of our simulation-based LOA program, MC-LOA$_{\alpha\beta}$, uses a selective 2-ply $\alpha\beta$-search at *each* step in its play-outs for choosing a move. Even though this reduces the number of simulations by more than a factor of two, the new version outperforms previous versions by a large margin — achieving a winning score of approximately 60%.

I. Introduction

For decades $\alpha\beta$ search [1] has been the standard approach used by programs for playing two-person zero-sum games such as Chess and Checkers (and many others) [2], [3]. In the early days deep search was not possible because of limited computational power, so heuristic knowledge was widely used to prune the search tree. The limited lookahead search typically investigated only a subset of the possible moves in each position, chosen selectively based on promise. With increased computational power the search gradually became more brute-force in nature, typically investigating all moves, although not necessarily to the same depth [4]. Over the years many search enhancements, including for controlling how deeply different moves are investigated, have been proposed for this framework that further enhance its effectiveness. The best tradeoff between using a fast search and incorporating informative heuristic knowledge for search guidance [5], [6] is constantly shifting based on new advancements in both hardware and software.

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 [7].

In recent years a new paradigm for game-tree search has emerged, so-called Monte-Carlo Tree Search (MCTS) [8], [9]. 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 [10], [11], [12], 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 computer Go, but other domains include General Game Playing [13], Amazons [14] and Hex [15].

The right tradeoff between search and knowledge equally applies to MCTS. The more informed we make each simulation play-out the slower it gets. On the one hand, this decreases the total number of simulations we can run in an allotted time, but on the other hand the result of each simulation is potentially more accurate. The former degrades the decision quality of MCTS whereas the latter improves it, so the question is where the right balance lies. The trend seems to be in favor of fast simulation play-outs where moves are chosen based on only computationally light knowledge [16], [17], although recently, adding heuristic knowledge at the cost of slowing down the simulation play-outs has proved beneficial in some games. This approach has been particularly successful in Lines of Action (LOA) [18], 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). In 2008, we showed that backpropagating game-theoretic values improved our MCTS LOA playing program MC-LOA considerably [19]. In 2009, we used a selective 1-ply search equipped with a sophisticated evaluation function for choosing the moves in the play-out of MC-LOA [20]. Such a 1-ply lookahead equates to what is often referred to as *greedy* search. That version of the program played at the same level as the $\alpha\beta$ program MIA, the best LOA playing entity in the world.

In this paper we further extend on previous results by using a 2-ply $\alpha\beta$-search for choosing the moves during the play-outs. To reduce the search overhead, special provisions must be taken in selectively choosing moves to fully expand. We evaluate the new version, MC-LOA$_{\alpha\beta}$, both on tactical test-suites and in tournament matches. Although the $\alpha\beta$-search slows down the simulation runs considerably, it improves the program's overall playing strength significantly.

The article is organized as follows. In Section II we explain the rules of LOA. Section III discusses MCTS and its implementation in our LOA program. In Section IV we describe how to enhance MCTS with $\alpha\beta$ search. The enhancement is empirically evaluated in Section V. Finally,

Mark Winands is a member of the Games and AI Group, Department of Knowledge Engineering, Faculty of Humanities and Sciences, Maastricht University, Maastricht, The Netherlands. Email: m.winands@maastrichtuniversity.nl; Yngvi Björnsson is a member of the School of Computer Science, Reykjavík University, Reykjavík, Iceland. Email: yngvi@ru.is

978-1-4577-0011-8/11/$26.00 ©2011 IEEE

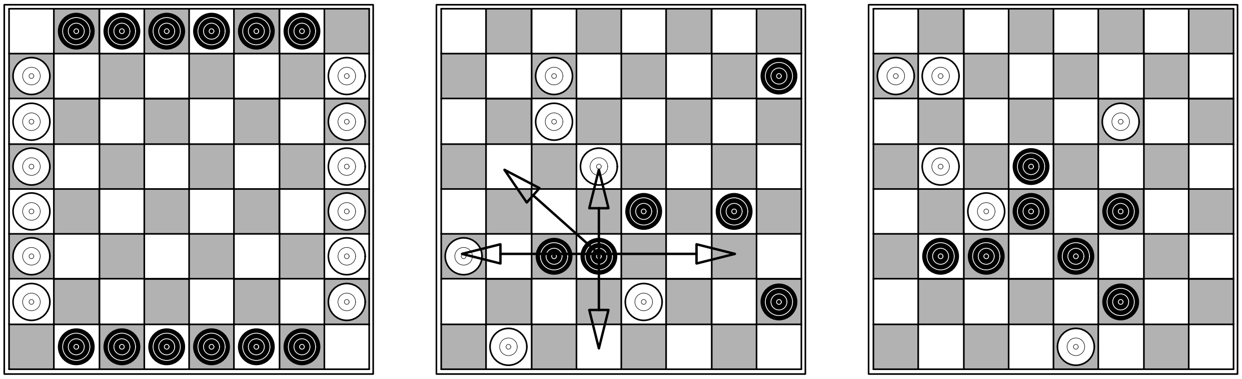

Fig. 1. (a) The initial position. (b) Example of possible moves. (c) A terminal position.

in Section VI 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 [21] described the game in his first edition of *A Gamut of Games*. The game has over the years been played in competitions both at the Mind Games Olympiad and on various sites on the world-wide web, gathering a community of expert human players. The strongest contemporary LOA programs have reached a super-human strength [22].

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 Figure 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 Figure 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 Figure 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 Figure 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.

III. Monte-Carlo Tree Search

In this section we discuss how we applied MCTS in LOA so far [18]. First, Subsection III-A gives an overview of MCTS. Next, Subsection III-B explains the four MCTS steps.

A. Overview

Monte-Carlo Tree Search (MCTS) [8], [9] is a best-first search method that does not require a positional evaluation function. It is based on a randomized exploration of the search space. Using the results of previous explorations, the algorithm gradually builds up a game tree in memory, and successively becomes better at accurately estimating the values of the most promising moves.

MCTS consists of four strategic steps, repeated as long as there is time left [23]. The steps, outlined in Figure 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 move along the way. When time is up, the move played by the program is the child of the root with the highest average value (or the most frequently visited child node, or some variation thereof [23]).

MCTS is unable to *prove* the game-theoretic value. However, in the long run MCTS equipped with the UCT formula

[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.

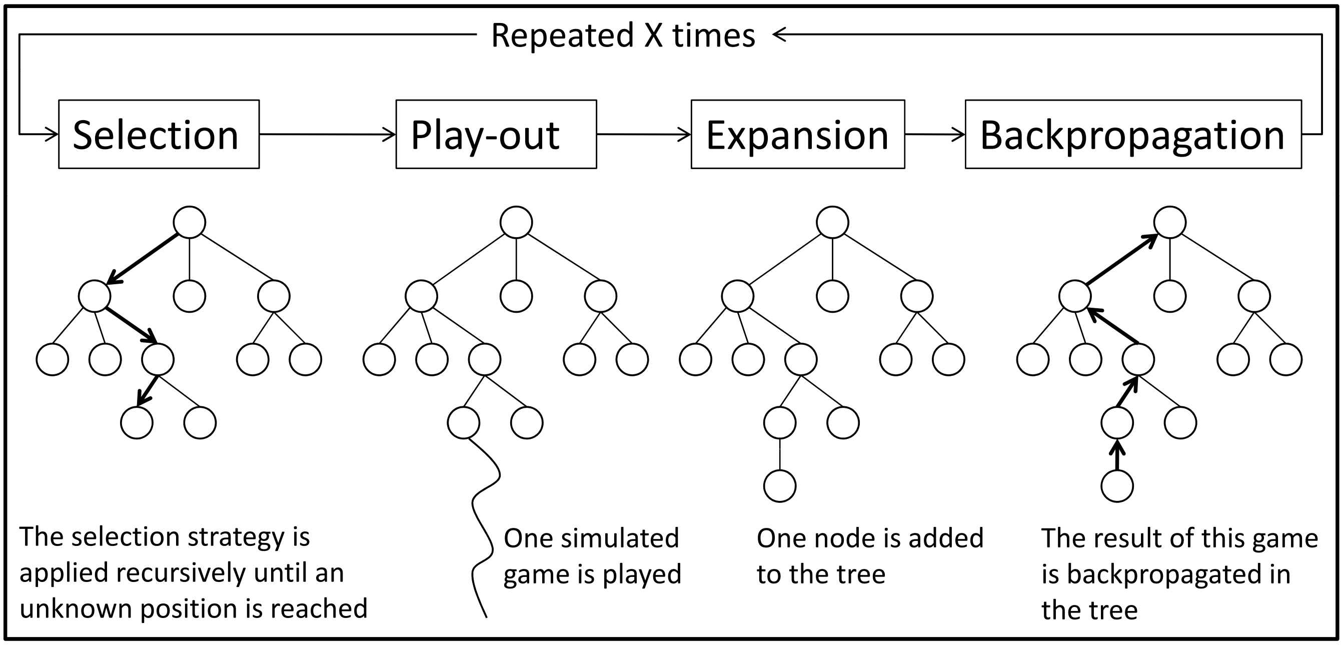
Repeated X times
Selection
Play-out
Expansion
Backpropagation
The selection strategy is applied recursively until an unknown position is reached
One simulated game is played
One node is added to the tree
The result of this game is backpropagated in the tree

Fig. 2. Outline of Monte-Carlo Tree Search (adapted from Chaslot *et al.* [23]).

[9] *converges* to the game-theoretic value. For instance, in endgame positions in fixed termination games like Go or Amazons, MCTS is often able to find the optimal move relatively quickly [24], [25]. 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 quickly enough to their game-theoretic value. We use therefore a newly proposed variant called Monte-Carlo Tree Search Solver (MCTS-Solver) [19] in our MC-LOA program, which is able to prove the game-theoretic value of a position. The backpropagation and selection mechanisms have been modified for this variant.

B. 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 [9], enhanced with Progressive Bias (PB) [23]. 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:

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

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* [26]. Instead of dividing the PB part by the visit count n_i as done originally [23], it is here divided divide it by $\sqrt{l_i + 1}$, where l_i is the number of losses [18]. 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 (cf. [27]).

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

The aforementioned selection strategy is only applied in nodes with visit count higher than a certain threshold T (here 5) [8]. If the node has been visited fewer times than this threshold, the next move is selected according to the Corrective strategy [20]. The move categories together with their transition probabilities are used to select the moves pseudo-randomly. We use the MIA 4.5 evaluation function [29] 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).

One additional improvement is to perform a 1-ply lookahead at leaf nodes (i.e., where the visit count equals one) [19]. 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 back-propagation 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.

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 [16]. The main idea is to play interesting moves according to heuristic knowledge.

In MC-LOA the following strategy is implemented [20]. At the first position entered in the play-out step, the Corrective strategy is applied. For the remainder of the play-out the Greedy strategy is applied [20]. In this strategy the MIA 4.5 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. When a move leads to a position with an evaluation over a preset threshold (i.e., 700 points [20]), the play-out is stopped and scored as a win. The remaining moves, which are not heuristically evaluated, are checked for a mate. Finally, if a selected move would lead to a position where heuristic evaluation function gives a value below a mirror threshold (i.e., -700 points); the play-out is scored as a loss.

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 [8]. 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 [8], i.e., $v_L = (\sum_k R_k)/n_L$.

In addition to backpropagating the values $\{1,0,-1\}$, game-theoretic values ∞ or $-\infty$ can be propagated [19]. The search assigns ∞ or $-\infty$ to a won or lost terminal position for the player to move in the tree, respectively. Backpropagating proven values in the tree is performed similar to regular negamax. Assume a simulation is run that ends in the game-tree at a node with a proven game-theoretic value. When backing such a proven value up the tree, there are several cases to consider. First, if the selected move (child) of a node returns ∞, the node is a win. That is, to prove that a node is a win, it suffices to prove that one child of that node is a win. Second, in the case that the selected child of a node returns $-\infty$, all the node's children must be checked. Now one of two possibilities can occur. Either all have values of $-\infty$, in which case the node is a loss. That is, to prove that a node is a loss, we must prove that all its children lead to a loss. Alternatively, one or more children of the node have a non-loss value, in which case we cannot prove the loss. The value the simulation is backing up is and remains a loss, however, it is not proven in the current position. Therefore, we still back up a loss value, but not a proven one. That is, -1 is now backpropagated. The node will thus simply be updated according to the regular backpropagation strategy for non-proven nodes as described previously.

IV. $\alpha\beta$ Search in the Play-out Step

In most abstract board games there is a delicate tradeoff between search and knowledge, as has been studied for $\alpha\beta$ search [5], [6], including in LOA [22]. There is also such a tradeoff in MCTS. Adding heuristic knowledge to the play-off strategy increases the accuracy and reliability of each play-out. However, if the heuristic knowledge is too computationally expensive, the number of play-outs per second decreases, offsetting the benefits [30]. For MCTS a consensus until recently seemed to be that the most beneficial tradeoff is achieved by choosing the moves in the play-outs based on some computationally light knowledge [16], [17]. However, in MC-LOA [18] choosing a move based on a selective 1-ply search equipped with a static evaluation function (i.e., the Greedy strategy) was shown to perform better than drawing the moves based only on light knowledge items (i.e., the move categories). Moreover, Lorentz [31] improved his MCTS program for the of Havannah by checking in the beginning of the play-out whether the opponent has a mate in one.

In here we move even further in this direction, now proposing to apply a 2-ply selective minimax-based search to choose the moves in the play-out step. However, to reduce the computational overhead we do this selectively, by fully evaluating only the first 7 moves at the root node and by looking at only the first 5 moves at the second ply. This specific configuration was obtained by trial-and-error. We observed that the playing strength of MC-LOA drops significantly if the number of moves to be considered is set too high. Next, at the root of the search we additionally check all moves for a terminal condition, that is, if they lead to an immediate win. The $\alpha\beta$ mechanism [1] is applied to prune the branches irrelevant for the search tree. The pseudo-code for this selective search is depicted in Figure 3. It is called on each turn in a play-out to decide on the move to play; the move at the root of the sab search that leads to the best search value is chosen (details not shown). The code is a regular (fail-soft) $\alpha\beta$ search augmented such that: a) only the first x moves are fully explored by the search; (b) the next y moves are checked for a terminal condition only; (c) the remaining moves are ignored altogether. The arrays n_eval and n_look, indexed by the search ply, are used to decide how many moves fall into each category.

The success of $\alpha\beta$ search is strongly dependent on the move ordering [32]. In the 2-ply search, we always first try two killer moves [33]. These are the last two moves that were best, or at least caused a cutoff, at the given depth. Moreover, if the 2-ply search is completed, we store the killer moves for that specific level in the play-out. In such a way there are always killer moves available for the $\alpha\beta$-search. Next, the move categories (III-B.1) together with their weights are used to order the remaining moves. The details of the move ordering are not shown in the aforementioned pseudo-code, but rather abstracted away in the $getMoves$ method. Finally, we use an aspiration window [32] when invoking the search to prune even more branches. The window is based on the thresholds configuration -600 and 600 that are used to stop the play-out.[2]

In the early game, the default version of MC-LOA runs at 5,100 sps (simulations per second) on a AMD Opteron 2.2 GHz, while the version equipped with $\alpha\beta$-search runs at 2,200 sps. If we consider the fact that LOA has an average branching of 30, a decrease of a factor 2.3 in sps is quite reasonable. Finally, we remark that if we would not have included killer moves in the $\alpha\beta$-search, the program would have slowed down by additional 10%.

V. Experiments

In this section we evaluate empirically the addition of a 2-ply $\alpha\beta$ search in the play-out step of MC-LOA, via self-play and against the world's strongest $\alpha\beta$-based LOA program MIA 4.5. The tactical performance on endgame positions is evaluated as well. We refer to the $\alpha\beta$-based MCTS player as MC-LOA$_{\alpha\beta}$. All experiments in this section were performed on an AMD Opteron 2.2 GHz computer.

```
// At first ply, fully evaluate first 7 moves,
// and check all remaining ones for a terminal
// cond.  At second ply, fully evaluate first
// 5 moves, and do not check any additional
// moves for a terminal condition.
n_eval[] = { 7, 5 };
n_look[] = { INF, 0 };

sab( pos, ply, d, alpha, beta )
{
   if ( d <= 0 || pos.isTerminal( ) ) {
      return pos.evaluate( );
   }
   best = alpha;
   stop = false;
   n =  pos.getMoves( moves ) ;
   for ( i = 0; i < n && !stop; ++i ) {
      v = best;
      pos.make( moves[i] );
      if ( i < n_eval[ply] ) {
         // Search and evaluate move.
         v = -sab(pos,ply+1,d-1,-beta,-best);
      }
      else if (i < n_eval[ply]+n_look[ply]) {
         // Check whether a move leads to an
         // immediate terminal position, in
         // particular a winning one, which
         // causes a cutoff.
         if ( pos.isTerminal( ) ) {
            v = pos.evaluate( );
         }
      }
      else {
       // Do not explore more moves.
         stop = true;
      }
      pos.unmake( moves[i] );
      if ( v > best ) {
         best = v;
         if ( best >= beta ) {
            stop = true;  // cutoff
         }
      }
   }
   return best;
}
```

Fig. 3. Pseudo code for $\alpha\beta$ based move selection in the play-outs

This remainder of this section is organized as follows. First, we briefly explain MIA in Subsection V-A. Next, we match MIA, MC-LOA, and MC-LOA$_{\alpha\beta}$ in a round-robin tournament in Subsection V-B. Finally, in Subsection V-C we evaluate the tactical strength of MC-LOA$_{\alpha\beta}$.

A. MIA (Maastricht In Action)

MIA is a world-class LOA program, which won the LOA tournament at the eighth (2003), ninth (2004), eleventh (2006) and fourteenth (2009) Computer Olympiad. Over its lifespan of 10 years it has gradually been improved and has for years now been generally accepted as the best

[2] These values are more tight as in the default MC-LOA (i.e., -700 and 700), but do not affect the strength of the program (cf. [20]).

LOA-playing entity in the world. All our 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 [28]. A *two-deep* transposition table [34] 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) [35]. Next, a null-move [36] is performed adaptively [37]. Then, an enhanced multi-cut is applied [38], [39]. For move ordering, the move stored in the transposition table (if applicable) is always tried first, followed by two killer moves [33]. 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 [40]. At the leaf nodes of the regular search, a quiescence search is performed to get more accurate evaluations.

B. Round-Robin Experiments

In the first set of experiments we quantify the performance of MIA, MC-LOA, and MC-LOA$_{\alpha\beta}$ in three round-robin tournaments with each a thinking time of 1, 5, and 30 seconds per move. 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, with both colors played equally. A standardized set of 100 3-ply starting positions [22] is used, with a small random factor in the evaluation function preventing games from being repeated.

TABLE I
1 SECOND PER MOVE TOURNAMENT RESULTS (WIN %). EACH DATA POINT IS BASED ON A 1000-GAME MATCH.

	MIA 4.5	MC-LOA	MC-LOA$_{\alpha\beta}$
MIA 4.5	-	55.40 ± 3.1	55.20 ± 3.1
MC-LOA	44.60 ± 3.1	-	56.60 ± 3.1
MC-LOA$_{\alpha\beta}$	44.80 ± 3.1	43.40 ± 3.1	-

TABLE II
5 SECONDS PER MOVE TOURNAMENT RESULTS (WIN %). EACH DATA POINT IS BASED ON A 1000-GAME MATCH.

	MIA 4.5	MC-LOA	MC-LOA$_{\alpha\beta}$
MIA 4.5	-	47.85 ± 3.1	42.40 ± 3.1
MC-LOA	52.15 ± 3.1	-	47.35 ± 3.1
MC-LOA$_{\alpha\beta}$	57.60 ± 3.1	52.65 ± 3.1	-

TABLE III
30 SECONDS PER MOVE TOURNAMENT RESULTS (WIN %). EACH DATA POINT IS BASED ON A 1000-GAME MATCH.

	MIA 4.5	MC-LOA	MC-LOA$_{\alpha\beta}$
MIA 4.5	-	50.95 ± 3.1	40.15 ± 3.0
MC-LOA	49.05 ± 3.1	-	40.55 ± 3.0
MC-LOA$_{\alpha\beta}$	59.85 ± 3.0	59.45 ± 3.0	-

In Tables I, II, and III the results of the tournaments are given for searches with a thinking time of 1, 5, and 30 seconds per move, respectively. Both the winning percentage and a 95% confidence interval (using a standard two-tailed Student's *t*-test) are given for each data point. In Table I, we see that for a short time setting MC-LOA$_{\alpha\beta}$ is weaker than MIA or MC-LOA. However, as the time controls increase the relative performance of MC-LOA$_{\alpha\beta}$ increases. Table II shows that for 5 seconds per move MC-LOA$_{\alpha\beta}$ plays on equal footing with MC-LOA, and defeats MIA in approximately 58% of the games. For 30 seconds per move, Table III shows that the performance gap widens even further, with MC-LOA$_{\alpha\beta}$ wining against MIA and MC-LOA almost 60% of the games. MC-LOA, on the other hand, is not able to gain from the increased time controls, still having a winning percentage around 50% against MIA on both the 5 and 30 second setting. With increased time controls the more accurate play-outs of MC-LOA$_{\alpha\beta}$ do more than outweigh the computational overhead involved. Although MC-LOA$_{\alpha\beta}$ generates on average around 2.3 times fewer simulations than MC-LOA, it still performs much better. With future increases in hardware speed the result suggests that this tradeoff will even further bias in MC-LOA$_{\alpha\beta}$'s favor.

Based on the results we may conclude the following. (1) Given sufficient time per move, performing small $\alpha\beta$ guided play-offs offers a better tradeoff in LOA, thus further improving MCTS-based programs. (2) MCTS using such an enhancement convincingly outperforms even the best $\alpha\beta$-based programs in LOA.

C. Tactical strength of MC-LOA$_{\alpha\beta}$

In [18], it was shown that the $\alpha\beta$ search of MIA was more than 10 times quicker in solving endgame positions than the MCTS search of MC-LOA. In the next series of experiments we investigate whether adding $\alpha\beta$ in the play-out step would improve the tactical strength of MC-LOA. The tactical performance of MC-LOA$_{\alpha\beta}$ was contrasted to that of MC-LOA. 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 III-B.1). The maximum number of nodes the programs are allowed to search on each problem is 20,000,000. The test set consists of 488 forced-win LOA positions.[4]

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

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

Table IV presents the results. The second column shows that MC-LOA$_{\alpha\beta}$ was able to solve 3 more positions than MC-LOA. In the third and fourth column the number of nodes and the time consumed are given for the subset of 373 positions that both programs were able to solve. We observe that the performance of MC-LOA$_{\alpha\beta}$ compared to MC-LOA is somewhat disappointing. MC-LOA$_{\alpha\beta}$ explores approximately 5% more nodes and consumes almost 20% more CPU time than MC-LOA. From this result it is clear that the improved strength of the MC-LOA$_{\alpha\beta}$ is not because of improved endgame play, which in LOA is typically the most tactical phase of the game. The improved playing strength seems more likely to be a result of improved positional play in the opening and middle game.

TABLE IV

COMPARING THE SEARCH ALGORITHMS ON 488 TEST POSITIONS

Algorithm	# of positions solved	373 positions	
	(out of 488)	Total nodes	Total time (ms.)
MC-LOA$_{\alpha\beta}$	391	884,503,705	4,877,737
MC-LOA	388	846,007,567	4,106,377

VI. Conclusion and Future Research

In this paper we described the application of $\alpha\beta$ search in the LOA-playing MCTS program MC-LOA. The new version, MC-LOA$_{\alpha\beta}$, applies a selective 2-ply $\alpha\beta$ search to choose the moves during the play-out. This $\alpha\beta$ search uses enhancements such as killer moves and aspiration windows to reduce the overhead.

Round-robin experiments against MIA and MC-LOA revealed that MC-LOA$_{\alpha\beta}$ performed better with increasing search time. For example, at a time setting of 30 seconds a move, MC-LOA$_{\alpha\beta}$ was able to defeat both opponents in approximately 60% of the games. On a test set of 488 LOA endgame positions MC-LOA$_{\alpha\beta}$ did not perform better in solving them than MC-LOA. This experiment suggests that the improvement in playing strength is due to better positional play in the opening and middle game, rather than improved tactical abilities in the endgame phase.

The main conclusion is that given sufficient time per move performing small $\alpha\beta$ searches in the play-out can improve the performance of a MCTS program significantly. We only experimented with this approach in the game of LOA, however, as there is nothing explicitly game specific with the approach, we believe that similar trends could also be seen in many other games. The exact tradeoff between search and knowledge will though differ from one game to the next (and from one program to another). For example, in our test domain the overhead of performing a 3-ply (or more) $\alpha\beta$ search decreased the strength of MC-LOA$_{\alpha\beta}$ drastically. This clear phase transition between 2- and 3-ply search is though not unlikely to shift with further advancements in both hardware and software. For example, with the advance of multi-core machines many more simulations are possible than before, potentially reaching the point of diminishing returns, in which case one avenue of further improvements would be through more knowledge-rich simulations. As future work we plan to investigate such issues as well as experimenting with alternative game domains.

Acknowledgments

This research is financed by the Netherlands Organisation for Scientific Research in the framework of the project Comparison and Development of Techniques for Embedding Search-Control Knowledge into Monte-Carlo Tree Search, grant number 040.11.203, as well as the Icelandic Centre for Research (RANNIS).

References

[1] D. E. Knuth and R. W. Moore, “An analysis of alpha-beta pruning,” *Artificial Intelligence*, vol. 6, no. 4, pp. 293–326, 1975.

[2] F. Hsu, *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton, NJ, USA: Princeton University Press, 2002.

[3] J. Schaeffer, *One Jump Ahead: Computer Perfection at Checkers*, 2nd ed. New York, NY, USA: Springer, 2009.

[4] T. A. Marsland and Y. Björnsson, “Variable-depth search,” in *Advances in Computer Games 9*, H. J. van den Herik and B. Monien, Eds. Universiteit Maastricht, Maastricht, The Netherlands, 2001, pp. 9–24.

[5] H. J. Berliner, G. Goetsch, M. S. Campbell, and C. Ebeling, “Measuring the performance potential of chess programs,” *Artificial Intelligence*, vol. 43, no. 1, pp. 7–20, 1990.

[6] A. Junghanns and J. Schaeffer, “Search versus knowledge in game-playing programs revisited,” in *IJCAI-97*, 1997, pp. 692–697.

[7] M. Müller, “Computer Go,” *Artificial Intelligence*, vol. 134, no. 1-2, pp. 145–179, 2002.

[8] 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. Berlin Heidelberg, Germany: Springer-Verlag, 2007, pp. 72–83.

[9] 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.

[10] 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.

[11] 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.

[12] B. Brügmann, “Monte Carlo Go,” Physics Department, Syracuse University, Tech. Rep., 1993.

[13] 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.

[14] R. J. 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. Berlin Heidelberg, Germany: Springer, 2008, pp. 13–24.

[15] B. Arneson, R. B. Hayward, and P. Henderson, “Monte Carlo Tree Search in Hex,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 251–258, 2010.

[16] 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.

[17] K.-H. Chen and P. Zhang, “Monte-Carlo Go with knowledge-guided simulations,” *ICGA Journal*, vol. 31, no. 2, pp. 67–76, 2008.

[18] M. H. M. Winands, Y. Björnsson, and J.-T. Saito, “Monte Carlo Tree Search in Lines of Action,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 239–250, 2010.

[19] 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. Berlin Heidelberg, Germany: Springer, 2008, pp. 25–36.

[20] 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. Berlin Heidelberg, Germany: Springer, 2010, pp. 33–44.

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

[22] 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.

[23] 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.

[24] P. Zhang and K.-H. Chen, "Monte Carlo Go capturing tactic search," *New Mathematics and Natural Computation*, vol. 4, no. 3, pp. 359–367, 2008.

[25] 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.

[26] 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.

[27] J. A. M. Nijssen and M. H. M. Winands, "Enhancements for multi-player Monte-Carlo Tree Search," in *Computers and Games (CG 2010)*, ser. Lecture Notes in Computer Science (LNCS), H. J. van den Herik, H. Iida, and A. Plaat, Eds., vol. 6151. Berlin Heidelberg, Germany: Springer, 2011, pp. 238–249.

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

[29] 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.

[30] G. M. J.-B. Chaslot, "Monte-carlo tree search," Ph.D. dissertation, Maastricht University, Maastricht, The Netherlands, 2010.

[31] R. J. Lorentz, "Improving monte-carlo tree search in Havannah," in *Computers and Games (CG 2010)*, ser. Lecture Notes in Computer Science (LNCS), H. J. van den Herik, H. Iida, and A. Plaat, Eds., vol. 6151. Berlin Heidelberg, Germany: Springer, 2011, pp. 105–115.

[32] T. A. Marsland, "A review of game-tree pruning," *ICCA Journal*, vol. 9, no. 1, pp. 3–19, 1986.

[33] 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.

[34] 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.

[35] 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.

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

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

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

[39] 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.

[40] 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.