Monte Carlo Game Solver

Tristan Cazenave

LAMSADE, Université Paris-Dauphine, PSL, CNRS, Paris, France
Tristan.Cazenave@dauphine.psl.eu

Abstract. We present a general algorithm to order moves so as to speedup exact game solvers. It uses online learning of playout policies and Monte Carlo Tree Search. The learned policy and the information in the Monte Carlo tree are used to order moves in game solvers. They improve greatly the solving time for multiple games.

1 Introduction

Monte Carlo Tree Search (MCTS) associated to Deep Reinforcement learning has superhuman results in the most difficult perfect information games (Go, Chess and Shogi) [26]. However little has been done to use this kind of algorithms to exactly solve games. We propose to use MCTS associated to reinforcement learning of policies so as to speedup the resolution of various games.

The paper is organized as follows: the second section deals with related work on games. The third section details the move ordering algorithms for various games. The fourth section gives experimental results for these games.

2 Previous Work

In this section we review the different algorithms that have been used to solve games. We then focus on the $\alpha\beta$ solver. As we improve $\alpha\beta$ with MCTS we show the difference to MCTS Solver. We also expose Depth First Proof Number Search as it has solved multiple games. We finish with a description of online policy learning as the resulting policy is used for our move ordering.

2.1 Solving Games

Iterative Deepening Alpha-Beta associated to a heuristic evaluation function and a transposition table is the standard algorithm for playing games such as Chess and Checkers. Iterative Deepening Alpha-Beta has also been used to solve games such as small board Go [30], Renju [29], Amazons endgames [15]. Other researchers have instead used a Depth-first Alpha-Beta without Iterative Deepening and with domain specific algorithms to solve Domineering [28] and Atarigo [3]. The advantage of Iterative Deepening associated to a transposition table for solving games is that it finds the shortest win and that it reuses the information of previous iterations for ordering the moves, thus maximizing the cuts. The heuristics usually used to order moves in combination with Iterative Deepening are: trying first the transposition table move, then trying killer moves and then sorting the remaining moves according to the History Heuristic [23]. The advantage of not using Iterative Deepening is that the iterations before the last one are not performed, saving memory and time, however if bad choices on move ordering happen, the search can waste time in useless parts of the search tree and can also find move sequences longer than necessary.

There are various competing algorithms for solving games [12]. The most simple are Alpha-Beta and Iterative Deepening Alpha-Beta. Other algorithms memorize the search tree in memory and expand it with a best first strategy: Proof Number Search [2], PN^2 [4], Depth-first Proof Number Search (Df-pn) [17], Monte Carlo Tree Search Solver [32,8] and Product Propagation [20].

Games solved with a best first search algorithm include Go-Moku with Proof Number search and Threat Space Search [1], Checkers using various algorithms [24], Fanorona with PN^2 [22], 6×6 Lines of Action with PN^2 [31], 6×5 Breakthrough with parallel PN^2 [21], and 9×9 Hex with parallel Df-pn [18].

Other games such as Awari were solved using retrograde analysis [19]. Note that retrograde analysis was combined with search to solve Checkers and Fanorona.

2.2 $\alpha\beta$ Solver

Iterative Deepening $\alpha\beta$ has long been the best algorithm for multiple games. Most of the Chess engines still use it even if the current best algorithm is MCTS [26].

Depth first $\alpha\beta$ is more simple but it can be better than Iterative Deepening $\alpha\beta$ for solving games since it does not have to explore a large tree before searching the next depth. More over in the case of games with only two outcomes the results are always either Won or Lost and enable immediate cuts when Iterative Deepening $\alpha\beta$ has to deal with unknown values when it reaches depth zero and the state is not terminal.

One interesting property of $\alpha\beta$ is that selection sort becomes an interesting sorting algorithm. It is often useful to only try the best move or a few good moves before reaching a cut. Therefore it is not necessary to sort all the moves at first. Selecting move by move as in selection sorting can be more effective.

2.3 MCTS Solver

MCTS has already been used as a game solver [32]. The principle is to mark as solved the subtrees that have an exact result. As the method uses playouts it has to go through the tree at each playout and it revisits many times the same states doing costly calculations to choose the move to try according to the bandit. Moreover in order to solve a game a large game tree has to be kept in memory.

The work on MCTS Solver has later been extended to games with multiple outcomes [8].

2.4 Depth First Proof Number Search

Proof Number Search is a best first algorithm that keeps the search tree in memory so as to expand the most informative leaf [2]. In order to solve the memory problem

Algorithm 1 The $\alpha\beta$ algorithm for solving games.

```
Function $\alpha\beta$ ($s$,$depth$,$\alpha$,$\beta$)
    if isTerminal ($s$) or $depth = 0$ then
        return Evaluation ($s$)
    end if
    if $s$ has an entry $t$ in the transposition table then
        if the result of $t$ is exact then
            return $t.res$
        end if
        put $t.move$ as the first legal move
    end if
    for $move$ in legal moves for $s$ do
        $s_1$ = play ($s$, $move$)
        $eval$ = -$\alpha\beta$($s_1$,$depth - 1$,-$\beta$,-$\alpha$)
        if $eval > \alpha$ then
            $\alpha = eval$
        end if
        if $\alpha \geq \beta$ then
            update the transposition table
            return $\beta$
        end if
    end for
    update the transposition table
    return $\alpha$
```

of Proof Number Search, the PN^2 algorithm has been used [4]. PN^2 uses a secondary Proof Number Search at each leaf of the main Proof Number Search tree, thus enabling the square of the total number of nodes of the main search tree to be explored. More recent developments of Proof Number Search focus on Depth-First Proof Number search (DFPN) [17]. The principle is to use a transposition table and recursive depth first search to efficiently search the game tree and solve the memory problems of Proof Number Search. DFPN can be parallelized to improve the solving time [13]. It has been improved for the game of Hex using a trained neural network [10]. It can be applied to many problems, including recently Chemical Synthesis Planning [14].

2.5 Online Policy Learning

Playout Policy Adaptation with Move Features (PPAF) has been applied to many games [7].

An important detail of the playout algorithm is the code function. In PPAF the same move can have different codes that depend on the presence of features associated to the move. For example in Breakthrough the code also takes into account whether the arriving square is empty or contains an opponent pawn.

The principle of the learning algorithm is to add 1.0 to the weight of the moves played by the winner of the playout. It also decreases the weights of the moves not played by the winner of the playout by a value proportional to the exponential of the weight. This algorithm is given in algorithm 2.

Algorithm 2 The PPAF adapt algorithm

```
1: Function adapt ($winner$, $board$, $player$, $playout$, $policy$)
2:     $polp \leftarrow policy$
3:     for $move$ in $playout$ do
4:         if $winner$ = $player$ then
5:             $polp$ [code($move$)] $\leftarrow$ $polp$ [code($move$)] + $\alpha$
6:             $z \leftarrow 0.0$
7:             for $m$ in possible moves on $board$ do
8:                 $z \leftarrow z$ + exp ($policy$ [code($m$)])
9:             end for
10:            for $m$ in possible moves on $board$ do
11:                $polp$ [code($m$)] $\leftarrow$ $polp$ [code($m$)] - $\alpha * \frac{exp(policy[code(m)])}{z}$
12:            end for
13:        end if
14:        play ($board$, $move$)
15:        $player \leftarrow$ opponent ($player$)
16:    end for
17:    $policy \leftarrow polp$
```

2.6 GRAVE

The principle of the All Moves As First heuristic (AMAF) is to compute statistics where all the moves of the playout are taken into account for the average of the moves. The principle of RAVE is to start from AMAF statistics when there are only a few playouts because the AMAF statistics are then more precise than the UCT statistics and to progressively switch to UCT statistics when more playouts are available. The formula found using a mathematical analysis [11] is:

$$\beta_m \leftarrow \frac{pAMAF_m}{pAMAF_m + p_m + bias \times pAMAF_m \times p_m}$$

$$argmax_m((1.0 - \beta_m) \times mean_m + \beta_m \times AMAF_m)$$

Where $pAMAF_m$ is the number of playouts that contain move m, p_m is the number of playouts that start with move m, $bias$ is a parameter to be tuned, $mean_m$ is the average of the playouts that start with move m and $AMAF_m$ is the average of the playouts that contain move m.

GRAVE is a simple but effective improvement of RAVE. GRAVE takes into account the AMAF statistics of the last state in the tree that has been visited more than a fixed number of times instead of always taking into account the AMAF statistics of the current state. In this way the states with few playouts use more meaningful AMAF statistics. GRAVE has very good results in General Game Playing [5,27].

3 Move Ordering for Different Games

We describe the general tools used for move ordering then their adaptation to different games.

3.1 Outline

In order to collect useful information to order moves we use a combination of the GRAVE algorithm [6] and of the PPAF algorithm. Once the Monte Carlo search with GRAVE and PPAF is finished we use the transposition table of the Monte Carlo search to order the moves, putting first the most simulated ones. When outside the transposition table we use the weights learned by PPAF to order the moves. The algorithm used to score the moves so as to order them is given in algorithm 3.

Algorithm 3 The Monte Carlo Move Ordering function

```
1:  Function orderMC ($board$, $code$)
2:      $score \leftarrow policy[code]$
3:      **if** $board$ has an entry $t$ in the MCTS TT **then**
4:          **if** $t.nbPlayouts > 100$ **then**
5:              **for** $move$ in legal moves for $board$ **do**
6:                  **if** $t.nbPlayouts[move] > 0$ **then**
7:                      **if** $code(move) = code$ **then**
8:                          $score \leftarrow t.nbPlayouts[move]$
9:                      **end if**
10:                 **end if**
11:             **end for**
12:         **end if**
13:     **end if**
14:     **return** $1000000000 - 1000 \times score$
```

3.2 Atarigo

Atarigo is a simplification of the game of Go. The first player to capture has won. It is a game often used to teach Go to beginners. Still it is an interesting games and tactics can be hard to master.

The algorithm for move ordering is given in algorithm 4. It always puts first a capture move since it wins the game. If no such move exist it always plays a move that saves a one liberty string since it is a forced move to avoid losing. Then it favors moves on liberties of opponent strings that have few liberties provided the move has itself sufficient liberties. If none of these are available it returns the evaluation by the Monte Carlo ordering function.

The code associated to a move is calculated using the colors of the four intersections next to the move.

Algorithm 4 The function to order moves at Atarigo

```
Function order ($board$, $move$)
    $minOrder \leftarrow 361$
    **for** $i$ in adjacents to $move$ **do**
        **if** $i$ is an opponent stone **then**
            $n \leftarrow$ number of liberties of $i$
            **if** $n = 1$ **then**
                **return** 0
            **end if**
            $nb \leftarrow n - 4 \times nbEmptyAdjacent(move)$
            **if** $nb < minOrder$ **then**
                $minOrder \leftarrow nb$
            **end if**
        **end if**
    **end for**
    **if** $move$ escapes an atari **then**
        **return** 1
    **end if**
    **if** $minOrder = 361$ **then**
        **if** MonteCarloMoveOrdering **then**
            **return** $orderMC(board, code(move))$
        **end if**
        **return** $20 - nbEmptyAdjacent(move)$
    **end if**
    **return** $minOrder$
```

3.3 Nogo

Nogo is the misere version of Atarigo [9]. It was introduced at the 2011 Combinatorial Game Theory Workshop in Banff. The first player to capture has lost. It is usually played on small boards. In Banff there was a tournament for programs and Bob Hearn won the tournament using Monte-Carlo Tree Search.

We did not find simple heuristics to order moves at Nogo. So the standard algorithm uses no heuristic and the MC algorithms sort moves according to algorithm 3.

3.4 Go

Go was solved for rectangular boards up to size 7×4 by the MIGOS program [30]. The algorithm used was an iterative deepening $\alpha\beta$ with transposition table. We use no heuristic to sort moves at Go and completely rely on algorithm 3 to order moves.

3.5 Breakthrough and Knightthrough

Breakthrough is an abstract strategy board game invented by Dan Troyka in 2000. It won the 2001 8x8 Game Design Competition and it is played on Little Golem. The game starts with two rows of pawns on each side of the board. Pawns can capture diagonally and go forward either vertically or diagonally. The first player to reach the opposite row has won. Breakthrough has been solved up to size 6×5 using Job Level Proof Number Search [21]. The code for a move at Breakthrough contains the starting square, the arrival square and whether it is empty or contains an enemy pawn. The ordering gives priority to winning moves, then to moves to prevent a loss, then Monte Carlo Move Ordering.

Misere Breakthrough is the misere version of Breakthrough, the games is lost if a pawn reaches the opposite side. It is also a difficult game and its is more difficult for MCTS algorithms [7]. The code for a move is the same as for Breakthrough and the ordering is Monte Carlo Move Ordering.

Knightthrough emerged as a game invented for the General Game Playing competitions. Pawns are replaced with knights. Misere Knightthrough is the misere version of the game where the goal is to lose. Codes for moves and move ordering are similar to Breakthrough.

3.6 Domineering

Domineering is played on a chess board and two players alternate putting dominoes on the board. The first player puts the dominoes vertically, the second player puts them horizontally. The first player who cannot play loses. In Misere Domineering the first player who cannot play wins. We use no heuristic to sort moves at Domineering and completely rely on algorithm 3 to order moves.

Algorithm 5 The function to order moves at Knightthrough

```
Function order ($board$, $move$)
    if $move$ is a winning move then
        return 0
    end if
    if $move$ captures an opponent piece then
        if capture in the first 3 lines then
            return 1
        end if
    end if
    if destination in the last 3 lines then
        if support(destination) $>$ attack(destination) then
            return 2
        end if
    end if
    if MonteCarloMoveOrdering then
        return $orderMC(board, code(move))$
    end if
    return 100
```

4 Experimental results

The iterative deepening $\alpha\beta$ with a transposition table (ID $\alpha\beta$ TT) is called with a null window since it saves much time compared to calling it with a full window. Other algorithms are called with the full window since they only deal with terminal states values and that the games we solve are either Won or Lost.

A transposition table containing 1 048 575 entries is used for all games. An entry in the transposition table is always replaced by a new one.

An algorithm name finishing with MC denotes the use of Monte Carlo Move Ordering. The times given for MC algorithms include the time for the initial MCTS that learns a policy. The original Proof Number Search algorithm is not included in the experiments since it fails due to being short of memory for complex games. The PN^2 algorithm solves this memory problem and is included in the experiments. The algorithms that do not use MC still do some move ordering but without Monte Carlo. For example in algorithm 4 for Atarigo the MonteCarloMoveOrdering boolean is set to False but the function to order moves is still used.

Table 1 gives the results for Atarigo. For Atarigo 5×5 $\alpha\beta$ TT MC is the best algorithm and is much better than $\alpha\beta$ TT. For Atarigo 6×5 the best algorithm is again $\alpha\beta$ TT MC which is much better than all other algorithms.

Table 2 gives the results for Nogo. Nogo 7×3 is solved in 49.72 seconds by $\alpha\beta$ TT MC with 100 000 playouts. This is 88 times faster than $\alpha\beta$ TT the best algorithm not using MC.

Nogo 5×4 is solved best by $\alpha\beta$ TT MC with 1 000 000 playouts before the $\alpha\beta$ search. It is 21 times faster than ID $\alpha\beta$ TT the best algorithm not using MC.

Table 1: Different algorithms for solving Atarigo.

Size	5×5	
Result	Won	
	Move count	Time
PN^2	14 784 088 742	37 901.56 s.
ID $\alpha\beta$ TT	$>$ 35 540 000 000	$>$ 86 400.00 s.
$\alpha\beta$ TT	$>$ 37 660 000 000	$>$ 86 400.00 s.
ID $\alpha\beta$ TT MC	62 800 334	126.84 s.
$\alpha\beta$ TT MC	**3 956 049**	**12.79 s.**

Size	6×5	
Result	Won	
	Move count	Time
PN^2	$>$ 33 150 000 000	$>$ 86 400.00 s.
ID $\alpha\beta$ TT	$>$ 37 190 000 000	$>$ 86 400.00 s.
$\alpha\beta$ TT	$>$ 7 090 000 000	$>$ 44 505.91 s.
ID $\alpha\beta$ TT MC	12 713 931 627	27 298.35 s.
$\alpha\beta$ TT MC	**329 780 434**	**787.17 s.**

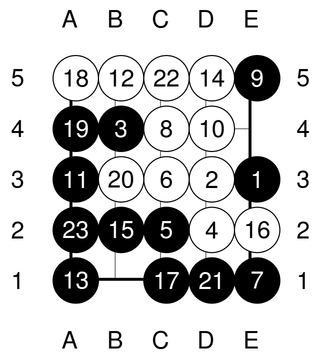
Table 2: Different algorithms for solving Nogo.

Size	7×3	
Result	Won	
	Move count	Time
PN^2	$>$ 80 390 000 000	$>$ 86 400.00 s.
ID $\alpha\beta$ TT	10 921 978 839	12 261.64 s.
$\alpha\beta$ TT	3 742 927 598	4 412.21 s.
ID $\alpha\beta$ TT MC	1 927 635 856	2 648.91 s.
$\alpha\beta$ TT MC	**35 178 886**	**49.72 s.**

Size	5×4	
Result	Won	
	Move count	Time
PN^2	$>$ 101 140 000 000	$>$ 86 400.00 s.
ID $\alpha\beta$ TT	1 394 182 870	1 573.72 s.
$\alpha\beta$ TT	1 446 922 704	1 675.64 s.
ID $\alpha\beta$ TT MC	73 387 083	134.26 s.
$\alpha\beta$ TT MC	**33 850 535**	**74.77 s.**

$\alpha\beta$ TT MC with 10 000 000 playouts solves Nogo 5×5 in 61 430.88 seconds and 46 092 056 485 moves. Nogo 5×5 was first solved in 2013 [25]. The solution we found is given in figure 1.

Fig. 1: Solution of Nogo 5×5.



As it is the first time results about solving Nogo are given we recapitulate in table 3 the winner for various sizes. A one means a first player win and a two a second player win.

	1	2	3	4	5	6	7	8	9	10
1	2	1	1	2	1	1	1	1	1	1
2	1	1	2	2	1	1	1	1	2	2
3	1	2	1	2	1	1	1	1		
4	2	2	2	2	1	1				
5	1	1	1	1	1					
6	1	1	1	1						
7	1	1	1							
8	1	1	1							
9	1	2								
10	1	2								

Table 3: Winner for Nogo boards of various sizes

Table 4: Different algorithms for solving Go.

Size	3×3	
Result	Won	
	Move count	Time
PN^2	246 394	3.72 s.
ID $\alpha\beta$ TT	840 707	11.34 s.
$\alpha\beta$ TT	420 265	11.50 s.
ID $\alpha\beta$ TT MC	375 414	5.62 s.
$\alpha\beta$ TT MC	**6 104**	**0.16 s.**

Size	4×3	
Result	Won	
	Move count	Time
PN^2	43 202 038	619.98 s.
ID $\alpha\beta$ TT	39 590 950	515.71 s.
$\alpha\beta$ TT	107 815 563	1 977.86 s.
ID $\alpha\beta$ TT MC	22 382 730	348.08 s.
$\alpha\beta$ TT MC	**4 296 893**	**96.63 s.**

Table 4 gives the results for Go. Playouts and depth first $\alpha\beta$ can last a very long time in Go since stones are captured and if random play occurs the goban can become almost empty again a number of times before the superko rules forbids states. In order to avoid very long and useless games an artificial limit on the number of moves allowed in a game was set to twice the size of the board. This is not entirely satisfactory since one can imagine weird cases where the limit is not enough. The problem does not appear in the other games we have solved since they converge to a terminal state before a fixed number of moves. The trick we use to address the problem is to send back an evaluation of zero if the search reaches the limit. When searching for a win with a null window this is equivalent to a loss and when searching for a loss it is equivalent to a win. Therefore if the search finds a win it does not rely on the problematic states. The 3×3 board was solved with a komi of 8.5, the 4×3 board was solved with a komi of 3.5.

Table 5: Different algorithms for solving Breakthrough.

Size	5×5	
Result	Lost	
	Move count	Time
PN^2	$>$ 38 780 000 000	$>$ 86 400.00 s.
ID $\alpha\beta$ TT	13 083 392 799	33 590.59 s.
$\alpha\beta$ TT	19 163 127 770	43 406.79 s.
ID $\alpha\beta$ TT MC	3 866 853 361	11 319.39 s.
$\alpha\beta$ TT MC	**3 499 173 137**	**9 243.66 s.**

Table 5 gives the results for Breakthrough. Using MC improves much the solving time. $\alpha\beta$ with MC uses seven times less nodes than the previous algorithm that solved Breakthrough 5×5 without patterns (i.e. parallel PN^2 with 64 clients [21]). Using endgame patterns divides by seven the number of required nodes for parallel PN^2.

Table 6: Different algorithms for solving Misere Breakthrough.

Size	4×5	
Result	Lost	
	Move count	Time
PN^2	$>$ 42 630 000 000	$>$ 86 400 s.
ID $\alpha\beta$ TT	$>$ 43 350 000 000	$>$ 86 400 s.
$\alpha\beta$ TT	$>$ 42 910 000 000	$>$ 86 400 s.
ID $\alpha\beta$ TT MC	1 540 153 635	3 661.50 s.
$\alpha\beta$ TT MC	**447 879 697**	**1 055.32 s.**

Table 6 gives the results for Misere Breakthrough. $\alpha\beta$ TT MC is the best algorithm and is much better than all non MC algorithms.

Table 7: Different algorithms for solving Knightthrough.

Size	6×6	
Result	Won	
	Move count	Time
PN^2	$>$ 33 110 000 000	$>$ 86 400 s.
ID $\alpha\beta$ TT	1 153 730 169	4 894.69 s.
$\alpha\beta$ TT	2 284 038 427	6 541.08 s.
ID $\alpha\beta$ TT MC	**17 747 503**	**102.60 s.**
$\alpha\beta$ TT MC	528 783 129	1 699.01 s.
Size	7×6	
Result	Won	
	Move count	Time
PN^2	$>$ 30 090 000 000	$>$ 86 400 s.
ID $\alpha\beta$ TT	$>$ 17 500 000 000	$>$ 86 400 s.
$\alpha\beta$ TT	$>$ 29 980 000 000	$>$ 86 400 s.
ID $\alpha\beta$ TT MC	**2 540 383 012**	**13 716.36 s.**
$\alpha\beta$ TT MC	6 650 804 159	23 958.04 s.

The results for Knightthrough are in table 7. ID $\alpha\beta$ TT MC is the best algorithm and far better than algorithms not using MC. This is the first time Knightthrough 7×6 is solved.

Table 8: Different algorithms for solving Misere Knightthrough.

Size	5×5	
Result	Lost	
	Move count	Time
PN^2	$>$ 45 290 000 000	$>$ 86 400 s.
ID $\alpha\beta$ TT	$>$ 52 640 000 000	$>$ 86 400 s.
$\alpha\beta$ TT	$>$ 56 230 000 000	$>$ 86 400 s.
ID $\alpha\beta$ TT MC	$>$ 41 840 000 000	$>$ 86 400 s.
$\alpha\beta$ TT MC	**20 375 687 163**	**42 425.41 s.**

Table 8 gives the results for Misere Knightthrough. Misere Knightthrough 5×5 is solved in 20 375 687 163 moves and 42 425.41 seconds by $\alpha\beta$ TT MC. This is the first time Misere Knightthrough 5×5 is solved. Misere Knightthrough 5×5 is much more difficult to solve than Knightthrough 5×5 which is solved in seconds by ID $\alpha\beta$ TT MC. This is due to Misere Knightthrough being a waiting game with longer games than Knightthrough.

Table 9: Different algorithms for solving Domineering.

Size	7×7	
Result	Won	
	Move count	Time
PN^2	$>$ 41 270 000 000	$>$ 86 400 s.
ID $\alpha\beta$ TT	18 958 604 687	35 196.62 s.
$\alpha\beta$ TT	197 471 137	376.23 s.
ID $\alpha\beta$ TT MC	2 342 641 133	5 282.06 s.
$\alpha\beta$ TT MC	**29 803 373**	**123.76 s.**

Table 9 gives the results for Domineering. The best algorithm is $\alpha\beta$ TT MC which is 3 times faster than $\alpha\beta$ TT without MC.

Table 10 gives the results for Misere Domineering. The best algorithm is $\alpha\beta$ TT MC which is much better than all non MC algorithms.

5 Conclusion

For the games we solved, Misere Games are more difficult to solve than normal games. In Misere Games the player waits and tries to force the opponent to play a losing move. This makes the game longer and reduces the number of winning sequences and winning moves.

Monte Carlo Move Ordering improves much the speed of $\alpha\beta$ with transposition table compare to depth first $\alpha\beta$ and Iterative Deepening $\alpha\beta$ with transposition table

Table 10: Different algorithms for solving Misere Domineering.

Size	7×7	
Result	Won	
	Move count	Time
PN^2	$>$ 44 560 000 000	$>$ 86 400 s.
ID $\alpha\beta$ TT	$>$ 49 290 000 000	$>$ 86 400 s.
$\alpha\beta$ TT	$>$ 49 580 000 000	$>$ 86 400 s.
ID $\alpha\beta$ TT MC	7 013 298 932	14 936.03 s.
$\alpha\beta$ TT MC	**72 728 678**	**212.25 s.**

but without Monte Carlo Move Ordering. The experimental results show significant improvements for nine different games.

In future work we plan to parallelize the algorithms and apply them to other problems. It would also be interesting to test if improved move ordering due to Monte Carlo Move Ordering would improve other popular solving algorithms such as DFPN. The ultimate goal with this kind of algorithms could be to solve exactly the game of Chess which is possible provided we have a very strong move ordering algorithm [16].

Acknowledgment

This work was supported in part by the French government under management of Agence Nationale de la Recherche as part of the "Investissements d'avenir" program, reference ANR19-P3IA-0001 (PRAIRIE 3IA Institute).

References

1. Allis, L.V., van den Herik, H.J., Huntjens, M.P.H.: Go-Moku solved by new search techniques. Computational Intelligence **12**, 7–23 (1996)
2. Allis, L.V., van der Meulen, M., van den Herik, H.J.: Proof-Number Search. Artificial Intelligence **66**(1), 91–124 (1994)
3. Boissac, F., Cazenave, T.: De nouvelles heuristiques de recherche appliquées à la résolution d'Atarigo. In: Intelligence artificielle et jeux. pp. 127–141. Hermes Science (2006)
4. Breuker, D.M.: Memory versus Search in Games. Ph.D. thesis, Universiteit Maastricht (1998)
5. Browne, C., Stephenson, M., Piette, É., Soemers, D.J.: A practical introduction to the ludii general game system. In: Advances in Computer Games. Springer (2019)
6. Cazenave, T.: Generalized rapid action value estimation. In: IJCAI 2015. pp. 754–760 (2015)
7. Cazenave, T.: Playout policy adaptation with move features. Theor. Comput. Sci. **644**, 43–52 (2016)
8. Cazenave, T., Saffidine, A.: Score bounded Monte-Carlo tree search. In: Computers and Games, LNCS, vol. 6515, pp. 93–104. Springer-Verlag (2011)
9. Chou, C.W., Teytaud, O., Yen, S.J.: Revisiting Monte-Carlo tree search on a normal form game: NoGo. In: Applications of Evolutionary Computation, LNCS, vol. 6624, pp. 73–82 (2011)

10. Gao, C., Müller, M., Hayward, R.: Focused depth-first proof number search using convolutional neural networks for the game of hex. In: (IJCAI 2017). pp. 3668–3674 (2017)
11. Gelly, S., Silver, D.: Monte-carlo tree search and rapid action value estimation in computer go. Artif. Intell. **175**(11), 1856–1875 (2011)
12. van den Herik, H.J., Uiterwijk, J.W.H.M., van Rijswijck, J.: Games solved: Now and in the future. Artif. Intell. **134**(1-2), 277–311 (2002)
13. Hoki, K., Kaneko, T., Kishimoto, A., Ito, T.: Parallel dovetailing and its application to depth-first proof-number search. ICGA Journal **36**(1), 22–36 (2013)
14. Kishimoto, A., Buesser, B., Chen, B., Botea, A.: Depth-first proof-number search with heuristic edge cost and application to chemical synthesis planning. In: Advances in Neural Information Processing Systems. pp. 7224–7234 (2019)
15. Kloetzer, J., Iida, H., Bouzy, B.: A comparative study of solvers in amazons endgames. In: IEEE Symposium On Computational Intelligence and Games, 2008. CIG'08. pp. 378–384. IEEE (2008)
16. Lemoine, J., Viennot, S.: Il n'est pas impossible de résoudre le jeu d'échecs. 1024 – Bulletin de la société informatique de France **6** (Juillet 2015)
17. Nagai, A.: Df-pn Algorithm for Searching AND/OR Trees and its Applications. Ph.D. thesis, The University of Tokyo (2002)
18. Pawlewicz, J., Hayward, R.B.: Scalable parallel DFPN search. In: Computers and Games - 8th International Conference, CG 2013, Yokohama, Japan, August 13-15, 2013, Revised Selected Papers. pp. 138–150 (2013)
19. Romein, J.W., Bal, H.E.: Solving awari with parallel retrograde analysis. IEEE Computer **36**(10), 26–33 (2003)
20. Saffidine, A., Cazenave, T.: Developments on product propagation. In: CG 2013. pp. 100–109 (2013)
21. Saffidine, A., Jouandeau, N., Cazenave, T.: Solving breakthrough with race patterns and job-level proof number search. In: ACG 2011. pp. 196–207 (2011)
22. Schadd, M.P.D., Winands, M.H.M., Uiterwijk, J.W.H.M., Herik, H.J.v.d., Bergsma, M.H.J.: Best play in fanorona leads to draw. New Mathematics and Natural Computation **4**(03), 369–387 (2008)
23. Schaeffer, J.: The history heuristic and alpha-beta search enhancements in practice. IEEE Transactions on Pattern Analysis and Machine Intelligence **11**(11), 1203–1212 (1989)
24. Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., Sutphen, S.: Checkers is solved. Science **317**(5844), 1518–1522 (2007)
25. She, P.: The Design and Study of NoGo Program. Master's thesis, National Chiao Tung University, Taiwan (2013)
26. Silver, D., al.: A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science **362**(6419), 1140–1144 (2018)
27. Sironi, C.F.: Monte-Carlo Tree Search for Artificial General Intelligence in Games. Ph.D. thesis, Maastricht University (2019)
28. Uiterwijk, J.W.: 11×11 domineering is solved: The first player wins. In: International Conference on Computers and Games. pp. 129–136. Springer (2016)
29. Wágner, J., Virág, I.: Solving renju. ICGA Journal **24**(1), 30–35 (2001)
30. van der Werf, E.C., Winands, M.H.: Solving go for rectangular boards. ICGA Journal **32**(2), 77–88 (2009)
31. Winands, M.H.: 6×6 LOA is solved. ICGA Journal **31**(4), 234–238 (2008)
32. Winands, M.H., Björnsson, Y., Saito, J.T.: Monte-carlo tree search solver. In: International Conference on Computers and Games. pp. 25–36. Springer (2008)