Ensemble UCT Needs High Exploitation

S. Ali Mirsoleimani[1,2], Aske Plaat[1], Jaap van den Herik[1] and Jos Vermaseren[2]
[1]*Leiden Centre of Data Science, Leiden University*
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
[2]*Nikhef Theory Group, Nikhef*
Science Park 105, 1098 XG Amsterdam, The Netherlands

Keywords: Monte Carlo tree search, Ensemble Search, Parallelism, Exploration-exploitation trade-off

Abstract: Recent results have shown that the MCTS algorithm (a new, adaptive, randomized optimization algorithm) is effective in a remarkably diverse set of applications in Artificial Intelligence, Operations Research, and High Energy Physics. MCTS can find good solutions without domain dependent heuristics, using the UCT formula to balance exploitation and exploration. It has been suggested that the optimum in the exploitation-exploration balance differs for different search tree sizes: small search trees needs more exploitation; large search trees need more exploration. Small search trees occur in variations of MCTS, such as parallel and ensemble approaches. This paper investigates the possibility of improving the performance of Ensemble UCT by increasing the level of exploitation. As the search trees become smaller we achieve an improved performance. The results are important for improving the performance of large scale parallelism of MCTS.

1 INTRODUCTION

Since its inception in 2006 (Coulom, 2006), the Monte Carlo Tree Search (MCTS) algorithm has gained much interest among optimization researchers. MCTS is a sampling algorithm that uses search results to guide itself through the search space, obviating the need for domain-dependent heuristics. Starting with the game of Go, an oriental board game, MCTS has achieved performance breakthroughs in domains ranging from planning and scheduling to high energy physics (Chaslot et al., 2008a; Kuipers et al., 2013; Ruijl et al., 2014). The success of MCTS depends on the balance between exploitation (look in areas which appear to be promising) and exploration (look in areas that have not been well sampled yet). The most popular algorithm in the MCTS family which addresses this dilemma is the Upper Confidence Bound for Trees (UCT) (Kocsis and Szepesvári, 2006).

As with most sampling algorithms, one way to improve the quality of the result is to increase the number of samples and thus enlarge the size of the MCTS tree. However, constructing a single large search tree with t samples or playouts is a time consuming process. A solution for this problem is to create a group of n smaller trees that each have t/n playouts and search these in parallel. This approach is used in root parallelism (Chaslot et al., 2008a) and in Ensemble UCT (Fern and Lewis, 2011). In both, root parallelism and Ensemble UCT, multiple independent UCT instances are constructed. At the end of the search process, the statistics of all trees are combined to yield the final result (Browne et al., 2012). However, there is contradictory evidence on the success of Ensemble UCT (Browne et al., 2012). On the one hand, Chaslot et al. found that, for Go, Ensemble UCT (with n trees of t/n playouts each) outperforms a plain UCT (with t playouts) (Chaslot et al., 2008a). On the other hand, Fern and Lewis were not able to reproduce this result in other domains (Fern and Lewis, 2011), they found situations where a plain UCT outperformed Ensemble UCT given the same total number of playouts.

As already mentioned, the success of MCTS depends on the exploitation-exploration balance. Previous work by Kuipers et al. has argued that when the tree size is small, more exploitation should be chosen, and with larger tree sizes, high exploration is suitable (Kuipers et al., 2013). The main contribution of this paper is that we show that this idea can be used in Ensemble UCT to improve its performance.

The remainder of this paper is structured as follows: in section 2 the required background information is briefly discussed. Section 3 discusses related work. Section 4 gives the experimental setup, together with the experimental results. Finally, a conclusion is given in Section 5.

```
function UCTSEARCH(r,m)
    i ← 1
    for i ≤ m do
        n ← select(r)
        n ← expand(n)
        Δ ←playout(n)
        backup(n,Δ)
    end for
    return
end function
```

Figure 1: The general MCTS algorithm.

2 BACKGROUND

Below we provide some background information on MCTS (Section 2.1), Ensemble UCT (Section 2.2), and the game of Hex (Section 2.3).

2.1 Monte Carlo Tree Search

The main building block of the MCTS algorithm is the search tree, where each node of the tree represents a game position. The algorithm constructs the search tree incrementally, expanding one node in each iteration. Each iteration has four steps (Chaslot et al., 2008b). (1) In the selection step, beginning at the root of the tree, child nodes are selected successively according to a selection criterion, until a leaf node is reached. (2) In the expansion step, unless the selected leaf node ends the game, a random unexplored child of the leaf node is added to the tree. (3) In the simulation step (also called playout step), the rest of the path to a final state is completed by playing random moves. At the end a score Δ is obtained that signifies the score of the chosen path through the state space. (4) In the backpropagation step (also called backup step), the Δ value is propagated back through the traversed path in the tree, which updates the average score (win rate) of a node. The number of times that each node in this path is visited is incremented by one. Figure 1 shows the general MCTS algorithm. In many MCTS implementations the UCT algorithm is chosen as the selection criterion (Kocsis and Szepesvári, 2006).

2.1.1 The UCT Algorithm

The UCT algorithm provides a solution for the problem of exploitation (look into existing promising areas) and exploration (look for new promising areas) in the selection phase of the MCTS algorithm (Kocsis and Szepesvári, 2006). A child node j is selected to maximize:

$$UCT(j) = \overline{X}_j + C_p\sqrt{\frac{\ln(n)}{n_j}} \qquad (1)$$

where $\overline{X}_j = \frac{w_j}{n_j}$, w_j is the number of wins in child j, n_j is the number of times child j has been visited, n is the number of times the parent node has been visited, and $C_p \geq 0$ is a constant. The first term in UCT equation is for exploitation and the second one is for exploration. The level of exploration of the UCT algorithm can be adjusted by the C_p constant. (High C_p means more exploration.)

2.1.2 Root Parallelism

Originally, root parallelism was considered as an UCT algorithm, viz. UCT in parallel. In root parallelism (Chaslot et al., 2008a) each thread is assumed to build simultaneously a private and independent MCTS search tree with a unique random seed. When root parallelism wants to select the next move to play, one of the threads collects the number of visits and the number of wins in the upper-most nodes of all trees and then computes for both (visits and wins) the total sum for each child (Chaslot et al., 2008a). Thereafter, it selects a move based on one of the possible policies. Figure 2 shows root parallelism. However, nowadays we have noted that UCT with root parallelism is not algorithmically equivalent to plain UCT, but is equivalent to Ensemble UCT (Browne et al., 2012).

2.2 Ensemble UCT

Ensemble UCT is given its place in the overview article by (Browne et al., 2012). Table 1 shows different possible configurations for Ensemble UCT. Each configuration has its own benefits. The total number of playouts is t and the size of ensemble (number of trees inside the ensemble) is n. It is supposed that n processors are available which is equal to the ensemble size. Figure 3 shows the pseudo-code of Ensemble UCT.

The first line of the table shows the situation where Ensemble UCT has $n \cdot t$ playouts in total while UCT

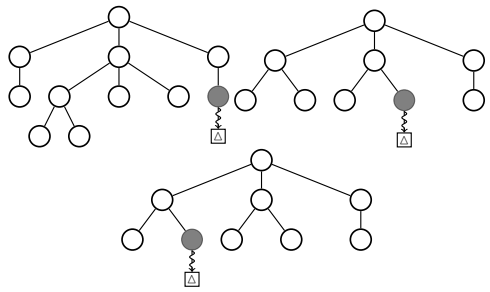

Figure 2: Different independent UCT trees are used in root parallelism.

Table 1: Different possible configurations for Ensemble UCT. The ensemble size is n.

Number of playouts			Playout speedup		Strength speedup
UCT	Ensemble UCT		n cores	1 core	
	Each tree	Total			
t	t	$n \cdot t$	1	$\frac{1}{n}$	Yes
t	$\frac{t}{n}$	t	n	1	?

has only t playouts. In this case, there would be no speedup in a parallel execution of the ensemble approach on n cores, but the larger search effort would presumably result in a better search result. We call this use of parallelism *Strength speedup*.

The second line of Table 1 shows a different possible configuration for Ensemble UCT. In this case, the total number of playouts for both UCT and Ensemble UCT is equal to t. Thus, each core searches a smaller tree of size $\frac{t}{n}$. The search will be n times faster (the ideal case). We call this use of parallelism *Playout speedup*. It is important to note that in this configuration both approaches take the same amount of time on a single core. However, there is still the question whether we can reach any *Strength speedup*. This question will be answered in Section 4.2.2.

2.3 The Game of Hex

Hex is a game with a board of hexagonal cells (Arneson et al., 2010). Each player is represented by a color (Black or White). Players take turns by placing a stone of their color on a cell of the board. The goal for each player is to create a connected chain of stones between the opposing sides of the board marked by their colors. The first player to complete this path wins the game.

In our implementation of the Hex game, a fast disjoint-set data structure is used to determine the connected stones. Using this data structure we have an efficient representation of the board position (Galil and Italiano, 1991).

$n \leftarrow$ ensemble size or number of trees
$t \leftarrow$ total number of playouts
function ENSEMBLEUCT(s,t,n)
  $m \leftarrow t/n$
  $i \leftarrow 1$
  for $i \leq n$ **do**
    $r[i] \leftarrow$ create an independent root node with state s
  end for
  $i \leftarrow 1$
  for $i \leq n$ **do**
    execute UCTSearch($r[i]$,m)
  end for
  collect from all trees the number of wins and visits to the root's children. Then compute the total sum of visits and wins for each child and store it to a new root r'.
  return child with $argmax\, w_j/n_j$ $j \in$ children of r'
end function

Figure 3: The pseudo-code of Ensemble UCT.

3 RELATED WORK

From the introduction we know that (Chaslot et al., 2008a) provided evidence that, for Go, root parallelism with n instances of $\frac{t}{n}$ iterations each outperforms plain UCT with t iterations, i.e., root parallelism (being a form of Ensemble UCT) outperforms plain UCT given the same total number of iterations. However, in other domains, (Fern and Lewis, 2011) did not find this result.

(Soejima et al., 2010) also analyzed the performance of root parallelism in detail. They found that a majority voting scheme gives better performance than the conventional approach of playing the move with the greatest total number of visits across all trees. They suggested that the findings in (Chaslot et al., 2008a) are explained by the fact that root parallelism performs a shallower search, making it easier for UCT to escape from local optima than the deeper search performed by plain UCT. In root parallelism each process does not build a search tree larger than the sequential UCT. Moreover, each process has a local tree that contains characteristics which differs from tree to tree. Recently, (Teytaud and Dehos, 2015) proposed a new idea by distinguishing between tactical behavior and strategic behavior. They transferred the RAVE (Rapid Action Value Estimate) ideas as developed by (Gelly and Silver, 2007), from the selection phase to the simulation phase. This implies that influencing the tree policy is changed into also influencing the Monte-Carlo policy.

Fern and Lewis thoroughly investigated an Ensemble UCT approach in which multiple instances of UCT were run independently. Their root statistics were combined to yield the final result (Fern and Lewis, 2011). So, our task is to explain the differ-

ences in their work and that by (Chaslot et al., 2008a).

4 EMPIRICAL STUDY

In this section, the experimental setup is described and then the experimental results are presented.

4.1 Experimental Setup

The Hex board is represented by a disjoint-set. This data structure has three operations *MakeSet*, *Find* and *Union*. In the best case, the amortized time per operation is $O(\alpha(n))$. The value of $\alpha(n)$ is less than 5 for all remotely practical values of n (Galil and Italiano, 1991).

In Ensemble UCT, each tree performs a completely independent UCT search with a different random seed. To determine the next move to play, the number of wins and visits of the root's children of all trees are collected. For each child the total sum of wins and the total sum of visits are computed. The child with the largest number of wins/visits is selected.

The plain UCT algorithm and Ensemble UCT are implemented in C++. In order to make our experiments as realistic as possible, we use a custom developed game playing program for the game of Hex (Mirsoleimani et al., 2014; Mirsoleimani et al., 2015). This program is highly optimized, and reaches a speed of more than 40,000 playouts per second per core on a 2,4 GHz Intel Xeon processor. The source code of the program is available online. [1]

As Hex is a 2-player game, the playing strength of Ensemble UCT is measured by playing versus a plain UCT with the same number of playouts. We expect to see an improvement for Ensemble UCT playing strength against plain UCT by choosing 0.1 as the value of C_p (high exploitation) when the number of playouts is small. In our experiments, the value of C_p is set to 1.0 for plain UCT (high exploration). Note that for the purpose of this research it is not important to find the optimal value of C_p, but just to to show the difference in effect on the performance.

Our experimental results show the percentage of wins for Ensemble UCT with a particular ensemble size and a particular C_p value. They are measured against plain UCT. Each data point represents the average of 200 games with a corresponding 99% confidence interval. Table 2 summarizes how the performance of Ensemble UCT versus plain UCT is evaluated. The concept of *high exploitation for small UCT tree* is significant if Ensemble UCT reaches a win rate of more than 50%. (Section 4.2.2 will shown that this is indeed the case.)

The board size for Hex is 11x11. In our experiments the maximum ensemble size is $2^8 = 256$. Thus, for 2^{17} playouts, when the ensemble size is 1 there are 2^{17} playouts per tree and when the ensemble size is $2^6 = 64$ the number of playouts per tree is 2^{11}. Throughout the experiments the ensemble size is multiplied by a factor of two.

The results were measured on a dual socket machine with 2 Intel *Xeon* E5-2596v2 processors running at 2.40GHz. Each processor has 12 cores, 24 hyperthreads and 30 MB L3 cache. Each physical core has 256KB L2 cache. The pack TurboBoost frequency is 3.2 GHz. The machine has 192GB physical memory. Intel's *icc 14.0.1* compiler is used to compile the program.

4.2 Experimental Results

Below we provide our experimental results. We distinguish them into hidden exploration in Ensemble UCT (4.2.1) and exploitation-exploration trade-off for Ensemble UCT (4.2.2).

4.2.1 Hidden Exploration in Ensemble UCT

It is important to understand that Ensemble UCT has a hidden exploration factor by nature. Two reasons are: (1) each tree in Ensemble UCT is independent, and (2) an ensemble of trees contains more exploration than a single UCT search with the same number of playouts would have. The hidden exploration is because each tree in Ensemble UCT searches in different areas of the search space.

In Figure 4 the difference in exploitation-exploration behavior of the Ensemble UCT and plain UCT is shown in the number of visits that one of the root's children counts when using one of the algorithmic approaches with $C_p = 0$. Both Ensemble UCT (Browne et al., 2012) and plain UCT (Browne et al., 2012) have 80,000 of playouts. In each experiment, a search tree for selecting the first move on an empty board is constructed. Each of the children corresponds to a possible move of an empty Hex board (i.e., 121 moves). Ensemble UCT is more explorative compared to plain UCT if it generates more data points with more distance from the x-axis than plain UCT. In Ensemble UCT the number of playouts is distributed among 8 separate smaller trees. Each of the trees has 10,000 playouts and for each child the number of visits is collected. When the value of C_p is 0, which means the exploration part of UCT formula is turned off, all possible moves in the Ensem-

[1] Source code is available at https://github.com/mirsoleimani/paralleluct/

Table 2: The performance evaluation of Ensemble UCT vs. plain UCT based on win rate.

Approach	Win (%)	Performance vs. plain UCT	Strength Speedup
Ensemble UCT	< 50	Worse than	No
	$= 50$	As good as	No
	> 50	Better than	Yes

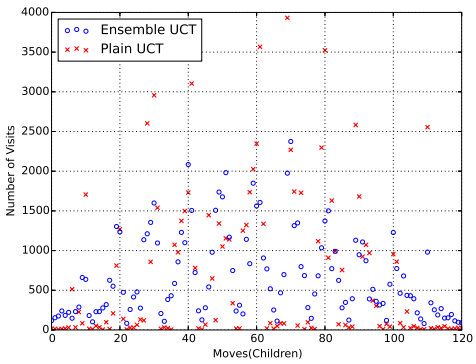

Figure 4: The number of visits for root's children in Ensemble UCT and plain UCT. Each child represents an available move on the empty Hex board with size 11x11. Both Ensemble UCT and plain UCT have 80,000 playouts and $C_p = 0$. In Ensemble UCT, the size of the ensemble is 8.

ble UCT receive at least a few visits. While for plain UCT with 80,000 playouts and $C_p = 0$ there are many of the moves with *no* visits. The data points when using plain UCT are closer to the x-axis compared to Ensemble UCT. However, for Ensemble UCT the peak is 2400, while it is 4000 visits for plain UCT. It means that plain UCT is more exploitative.

4.2.2 Exploitation-Exploration trade-off for Ensemble UCT

In Figures 5 and 6, from the left side to the right side of a graph, the ensemble size (number of search trees per ensemble) increases by a factor of two and the number of playouts per tree (tree size) decreases by the same factor. Thus, at the most right hand side of the graph we have the largest ensemble with smallest trees. The total number of playouts always remains the same throughout an experiment for both Ensemble UCT and plain UCT. The value of C_p for plain UCT is always 1.0 which means high exploration.

Figure 5 shows the *relations* between the value of C_p and the ensemble size, when both plain UCT and Ensemble UCT have the same number of total playouts. Moreover, Figure 5 shows the *performance* of Ensemble UCT for different values of C_p. It shows that when $C_p = 1.0$ (highly explorative) Ensemble UCT performs as good as or mostly worse than plain UCT. When Ensemble UCT uses $C_p = 0.1$ (highly exploitative) then for small ensemble sizes (large sub-trees) the performance of Ensemble UCT sharply drops down. By increasing the ensemble size (smaller sub-trees), the performance of Ensemble UCT keeps improving until it becomes as good as or even better than plain UCT.

In order to investigate the effect of enlarging the number of playouts on the performance of Ensemble UCT, the second experiment is conducted using 2^{18} playouts. Figure 6 shows that when for this large number of playouts the value of $C_p = 1.0$ is high (i.e., highly explorative) the performance of Ensemble UCT cannot be better that plain UCT. While for a small value of $C_p = 0.1$ (i.e., highly exploitative) the performance of Ensemble UCT is almost always better than plain UCT after ensemble size is 2^5. Therefore, there is a marginal strength speedup. The potential playout speedup could be up to the ensemble size if sufficient number of processing cores is available.

5 CONCLUSION

This paper describes an empirical study on Ensemble UCT with different sets of configurations for ensemble size, tree size and exploitation-exploration trade-off. Previous studies on Ensemble UCT/root parallelism provided inconclusive evidence on the effectiveness of Ensemble UCT (Chaslot et al., 2008a; Fern and Lewis, 2011; Browne et al., 2012). Our results suggest that the reason lies in the exploration-exploitation trade-off in relation to the *size of the sub-trees*. Our results provide clear evidence that the performance of Ensemble UCT is improved by selecting higher exploitation for smaller search trees given a fixed time bound or number of simulations.

This work is motivated, in part, by the observation in (Chaslot et al., 2008a) of super-linear speedup in root parallelism. Finding super-linear speedup in two-agent games occurs infrequently. Most studies in parallel game-tree search report a battle against search overhead, communication overhead, and synchronization overhead (see, e.g., (Romein, 2001)). For super-linear speedup to occur, the parallel search must search *fewer* nodes than the sequential search. In

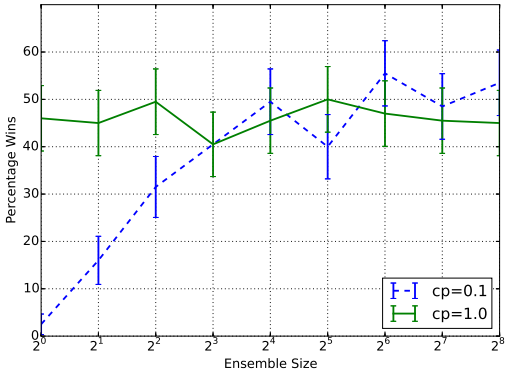

Figure 5: The total number of playouts for both plain UCT and ensemble UCT is $2^{17} = 131072$. The percentage of wins for ensemble UCT is reported. The value of C_p for plain UCT is always 1.0 when playing against Ensemble UCT. To the left few large UCT trees, to the right many small UCT trees.

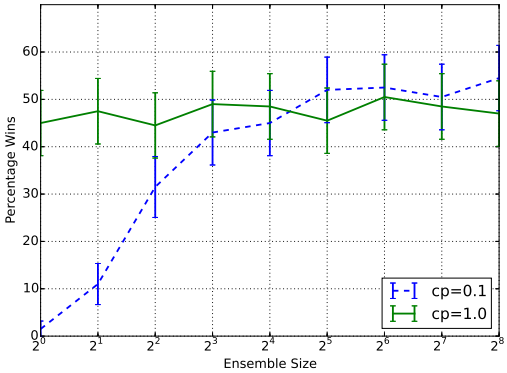

Figure 6: The total number of playouts for both plain UCT and ensemble UCT is $2^{18} = 262144$. The percentage of wins for ensemble UCT is reported. The value of C_p for plain UCT is always 1.0 when playing against Ensemble UCT. To the left few large UCT trees, to the right many small UCT trees.

most algorithms, parallelizations suffer because parts of the tree are searched with less information than is available in the sequential search, causing *more* nodes to be expanded. This study has shown how the remarkable situation in which the parallel search tree is smaller than the sequential search tree can indeed occur in MCTS. The ensemble of the independent (parallel) sub-trees can be smaller than the monolithic total tree. When C_p is chosen low (i.e., exploitative) the Ensemble search runs efficiently, where the monolithic plain UCT search is less efficient (see Figures 5 and 6).

For future work, we will explore other parts of the parameter space, to find optimal C_p settings for different combinations of tree size and ensemble size. Also, we will study the effect in different domains. Even more important will be the study on the effect of C_p in tree parallelism (Chaslot et al., 2008a).

ACKNOWLEDGEMENTS

This work is supported in part by the ERC Advanced Grant no. 320651, “HEPGAME.”

REFERENCES

Arneson, B., Hayward, R. B., and Henderson, P. (2010). Monte Carlo Tree Search in Hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43.

Chaslot, G., Winands, M., and van den Herik, J. (2008a). Parallel Monte-Carlo Tree Search. In *the 6th Internatioal Conference on Computers and Games*, volume 5131, pages 60–71. Springer Berlin Heidelberg.

Chaslot, G. M. J. B., Winands, M. H. M., van den Herik, J., Uiterwijk, J. W. H. M., and Bouzy, B. (2008b). Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357.

Coulom, R. (2006). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Proceedings of the 5th International Conference on Computers and Games*, volume 4630 of *CG'06*, pages 72–83. Springer-Verlag.

Fern, A. and Lewis, P. (2011). Ensemble Monte-Carlo Planning: An Empirical Study. In *ICAPS*, pages 58–65.

Galil, Z. and Italiano, G. F. (1991). Data Structures and Algorithms for Disjoint Set Union Problems. *ACM Comput. Surv.*, 23(3):319–344.

Gelly, S. and Silver, D. (2007). Combining online and offline knowledge in UCT. In *the 24th International Conference on Machine Learning*, pages 273–280, New York, USA. ACM Press.

Kocsis, L. and Szepesvári, C. (2006). *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg.

Kuipers, J., Plaat, A., Vermaseren, J., and van den Herik, J. (2013). Improving Multivariate Horner

Schemes with Monte Carlo Tree Search. *Computer Physics Communications*, 184(11):2391–2395.

Mirsoleimani, S. A., Plaat, A., van den Herik, J., and Vermaseren, J. (2015). Parallel Monte Carlo Tree Search from Multi-core to Many-core Processors. In *ISPA 2015 : The 13th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 77–83, Helsinki.

Mirsoleimani, S. A., Plaat, A., Vermaseren, J., and van den Herik, J. (2014). Performance analysis of a 240 thread tournament level MCTS Go program on the Intel Xeon Phi. In *The 2014 European Simulation and Modeling Conference (ESM'2014)*, pages 88–94, Porto, Portugal. Eurosis.

Romein, J. W. (2001). *Multigame – An Environment for Distributed Game-Tree Search*. PhD thesis, Vrije Universiteit.

Ruijl, B., Vermaseren, J., Plaat, A., and van den Herik, J. (2014). Combining Simulated Annealing and Monte Carlo Tree Search for Expression Simplification. *Proceedings of ICAART Conference 2014*, 1(1):724–731.

Soejima, Y., Kishimoto, A., and Watanabe, O. (2010). Evaluating Root Parallelization in Go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):278–287.

Teytaud, F. and Dehos, J. (2015). One the Tactical and Strategic Behaviour of MCTS When Biasing Random Simulations. *ICCA Journal*, 38(2):67–80.