Investigations with Monte Carlo Tree Search for Finding Better Multivariate Horner Schemes

H. Jaap van den Herik[1](✉), Jan Kuipers[2], Jos A.M. Vermaseren[2], and Aske Plaat[1]

[1] Tilburg Center for Cognition and Communication, Tilburg University, Warandelaan 2, 5037 AB Tilburg, The Netherlands
jaapvandenherik@gmail.com
[2] Nikhef Theory Group, Science Park 105, 1098 XG Amsterdam, The Netherlands

Abstract. After a computer chess program had defeated the human World Champion in 1997, many researchers turned their attention to the oriental game of Go. It turned out that the minimax approach, so successful in chess, did not work in Go. Instead, after some ten years of intensive research, a new method was developed: MCTS (Monte Carlo Tree Search), with promising results. MCTS works by averaging the results of random play-outs. At first glance it is quite surprising that MCTS works so well. However, deeper analysis revealed the reasons.

The success of MCTS in Go caused researchers to apply the method to other domains. In this article we report on experiments with MCTS for finding improved orderings for multivariate Horner schemes, a basic method for evaluating polynomials. We report on initial results, and continue with an investigation into two parameters that guide the MCTS search. Horner's rule turns out to be a fruitful testbed for MCTS, allowing easy experimentation with its parameters. The results reported here provide insight into how and why MCTS works. It will be interesting to see if these insights can be transferred to other domains, for example, back to Go.

Keywords: Artificial intelligence · High energy physics · Horners rule · Monte Carlo Tree Search · Go · Chess

1 Introduction

In 1965, the Soviet mathematician Aleksandr Kronrod called chess the Drosophila Melanogaster of Artificial Intelligence [29]. At that time, chess was a convenient domain that was well suited for experimentation. Moreover, dedicated research programs all over the world created quick progress. In half a century the dream of

Parts of this work have appeared in a keynote speech by the first author at the International Conference on Agents and Artifical Intelligence ICAART 2013 in Barcelona under the title "Connecting Sciences." These parts are reprinted with permission by the publisher.

© Springer-Verlag Berlin Heidelberg 2014
J. Filipe and A. Fred (Eds.): ICAART 2013, CCIS 449, pp. 3–20, 2014.
DOI: 10.1007/978-3-662-44440-5_1

Fig. 1. Example of a Go board.

beating the human world champion was realized. On May 11, 1997 Garry Kasparov, the then highest rated human chess player ever, was defeated by the computer program DEEP BLUE, in a highly publicized six game match in New York.

So, according to some, the AI community lost their Drosophila in 1997, and started looking for a new one. The natural candidate was an even harder game: the oriental game of Go. Go is played on a 19×19 board, see Fig. 1. Its state space is much larger than the chess state space. The number of legal positions reachable from the starting position in Go (the empty board) is estimated to be $\mathcal{O}(10^{171})$ [1], whereas for chess this number is "just" $\mathcal{O}(10^{46})$ [15]. If chess is a game of tactics, then Go is a game of strategy. The standard minimax approach that worked so well for chess (and for other games such as checkers, Awari, and Othello) did not work well for Go, and so Go became the new Drosophila. For decades, computer Go programs played at the level of weak amateur. After 1997, the research effort for computer Go intensified. Initially, progress was slow, but in 2006, a breakthrough happened. The breakthrough and some of its consequences, are the topic of this article.

The remainder of the contribution is structured as follows. First, the techniques that worked so well in chess will be discussed briefly. Second, the new search method that caused the breakthrough in playing strength in Go will be described. Then, a successful MCTS application to Horner's rule of multivariate polynomials will be shown. It turns out that Horner's rule yields a convenient test domain for experimentation with MCTS. We complete the article by an in-depth investigation of the search parameters of MCTS.

A note on terminology. The rule published by William Horner almost two centuries ago is called Horner's *rule*. It is a technique to reduce the work required for the computation of a polynomial in a single variable at a particular value. Finding better variable orderings of multivariate polynomials, in order to then apply Horner's rule repeatedly, is called finding better Horner *schemes*.

2 The Chess Approach

The heart of a chess program consists of two parts: (1) a heuristic evaluation function, and (2) the minimax search function. The purpose of the heuristic evaluation function is to provide an estimate of how good a position looks, and sometimes of its chances of winning the game [17]. In chess this includes items such as the material balance (capturing a pawn is good, capturing a queen is usually very good), mobility, and king safety. The purpose of the search function is to look ahead: if I play this move, then my opponent would do this, and then I would do that, and ..., etc. By searching more deeply than the opponent the computer can find moves that the heuristic evaluation function of the opponent mis-evaluates, and thus the computer can find the better move.

Why does this approach fail in Go? Originally, the main reason given was that the search tree is so large (which is true). In chess, the opening position has 20 legal moves (the average number of moves is 38 [18,22]). In Go, this number is 361 (and thereafter it decreases with one per move). However, soon it turned out that an even larger problem was posed by the construction of a good heuristic evaluation function. In chess, material balance, the most important term in the evaluation function, can be calculated efficiently and happens to be a good first heuristic. In Go, so far no good heuristics have been found. The influence of stones and the life and death of groups are generally considered to be important, but calculating these terms is time consuming, and the quality of the resulting evaluation is a mediocre estimator for the chances of winning a game.

Alternatives. Lacking a good evaluation function and facing the infeasibility of a full-width look-ahead search, most early Go programs used as a first approach the knowledge-based approach: (1) generate a limited number of likely candidate moves, such as corner moves, attack/defend groups, connecting moves, and ladders, and (2) search for the best move in this reduced state space [34]. The Go heuristics used for choosing the candidate moves can be generalized in move patterns, which can be learned from game databases [44,45]. A second approach was to use neural networks, also with limited success [19]. This approach yielded programs that could play a full game that looked passable, but never reached more than weak amateur level.

3 Monte Carlo

In 1993, the mathematician and physicist Bernd Brügmann was intrigued by the use of simulated annealing for solving the traveling salesman problem. If such a basic procedure as randomized local search (also known as Monte Carlo) could find shortest tours, then perhaps it could find good moves in Go? He wrote a 9×9 Go program based on simulated annealing [7]. Crucially, the program did not have a heuristic evaluation function. Instead it played a series of random moves all the way until the end of the game was reached. Then the final position was trivially scored as either a win or a loss. This procedure of randomized play-outs

was repeated many times. The result was averaged and taken to be an estimate of the "heuristic" value of each move. So instead of searching a tree, Brügmann's program searched paths, and instead of using the minimax function to compute the scores, the program took the average of the final scores. The program had no domain knowledge, except not to fill its own territory. Could this program be expected to play anything but meaningless random moves?

Surprisingly, it did. Although it certainly did not play great or even good moves, the moves looked better than random. Brügmann concluded that by just following the rules of the game the average of many thousands of plays yielded better-than-random moves.

At that time, the attempt to connect the sciences of physics and artificial intelligence appeared to be a curiosity. Indeed, the hand-crafted knowledge-based programs still performed significantly better. For the next ten years not much happened with Monte Carlo Go.

Monte Carlo Tree Search. Then, in 2003, Bouzy and Helmstetter reported on further experiments with Monte Carlo playouts, again stressing the advantage of having a program that can play Go moves without the need for a heuristic evaluation function [2,5]. They tried adding a small 2-level minimax tree on top of the random playouts, but this did not improve the performance. In their conclusion they refer to other works that explored statistical search as an alternative to minimax [24,38] and concluded: "Moreover, the results of our Monte Carlo programs against knowledge-based programs on 9×9 boards and the ever-increasing power of computers lead us to think that Monte Carlo approaches are worth considering for computer Go in the future."

They were correct.

Three years later a breakthrough took place by the repeated introduction of MCTS and UCT. Coulom [16] described Monte Carlo evaluations for tree-based search, specifying rules for node selection, expansion, playout, and backup. Chaslot et al. coined the term Monte Carlo Tree Search or MCTS, in a contribution that received the ICGA best publication award in 2008 [10,12]. In 2006 Kocsis and Szepesvari [25] laid the theoretical foundation for a selection rule that balances exploration and exploitation and that is guaranteed to converge to the minimax value. This selection rule is termed UCT, short for **U**pper **C**onfidence bounds for multi-armed bandits [4] applied to **T**rees (see Eq. (4)). Gelly et al. [21] used UCT in a Go program called MoGo, short for Monte Carlo Go, which was instantly successful. MoGo received the ICGA award in 2009. Chaslot et al. [11] also described the application of MCTS in Go, reporting that it outperformed minimax, and mentioned applications beyond Go.

Since 2006 the playing strength of programs improved rapidly to the level of strong amateur/weak master (2-3 dan). The MCTS breakthrough was confirmed when, for the first time, a professional Go player was beaten in a single game. In August 2008 at the 24th Annual Go Congress in Portland, Oregon, MoGo-Titan, running on 800 cores of the Huygens supercomputer in Amsterdam, beat 8P dan professional Kim MyungWan with a 9-stone handicap [14]. Further refinements have increased the playing strength. At the Human versus

Computer Go Competition that was held as part of the IEEE World Congress on Computational Intelligence in June 2012 in Brisbane, Australia, the program ZEN defeated the 9P dan professional Go player Takemiya Masaki with a four-stone handicap (≈5P dan) on the 19×19 board.

The main phases of MCTS are shown in Fig. 2. They are explained briefly below.

After the introduction om MCTS, there has been a large research interest in MCTS. Browne et al. [8] provides an extensive survey, referencing 240 publications.

MCTS Basics. MCTS consists of four main steps: selection, expansion, simulation (playout), and back-propagation (see Fig. 2). The main steps are repeated as long as there is time left. For each step the activities are as follows.

(1) In the selection step the tree is traversed from the root node until we reach a node, where a child is selected that is not part of the tree yet.
(2) Next, in the expansion step the child is added to the tree.
(3) Subsequently, during the simulation 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 Go), 0 in case of a draw, and −1 in case of a win for White.
(4) In the back-propagation step, R is propagated backwards, through the previously traversed nodes. Finally, the move played by the program is the child of the root with the best win/visit count, depending on UCT probability calculations (to be discussed briefly below).

Crucially, the selection rule of MCTS allows balancing of (a) exploitation of parts of the tree that are known to be good (i.e., high win rate) with (b) exploration of parts of the tree that have not yet been explored (i.e., low visit count).

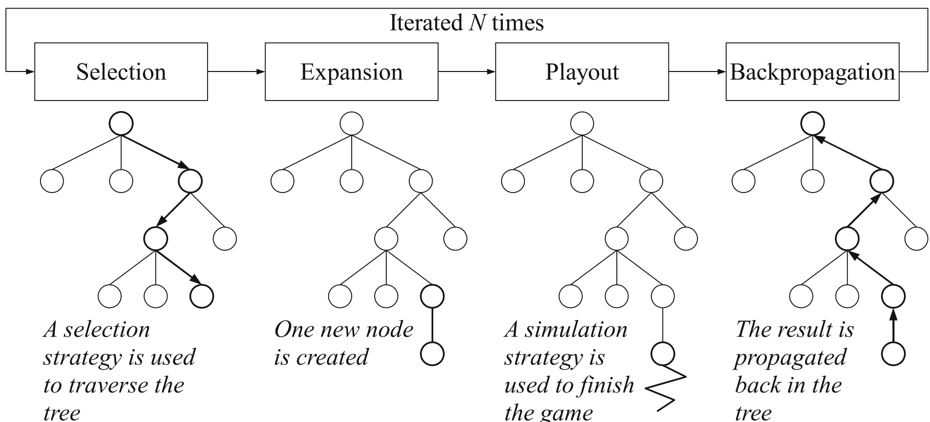

Fig. 2. The basic Monte Carlo Tree Search scheme.

Originally MCTS used moves in the playout phase that were strictly random. However, soon better results were obtained by playing moves that use small (fast) amounts of domain knowledge. Nowadays, many programs use pattern databases for this purpose [21]. The high levels of performance that are currenlty achieved with MCTS depend to a large extent on enhancements of the expansion strategy, simulation phase, and the parallelization techniques. (So, after all, small amounts of domain knowledge are needed, albeit not in the form of a heuristic evaluation function. No expensive influence or life-and-death calculations are used, but fast pattern lookups.)

Applications Beyond Go. The striking performance of MCTS in Go has led researchers to apply the algorithm to other domains. Traditionally, best-first algorithms rely on domain knowledge to try the "best" moves first. This domain knowledge is often hard to codify correctly and is expensive to compute. Many researchers have looked for best-first algorithms that could somehow do without domain knowledge [35–37,42]. The ability of MCTS to magically home in on clusters of "bright spots" in the state space without relying on domain knowledge has resulted in a long list of other applications, for example, for proof-number search [40]. In addition, MCTS has been proposed as a new framework for game-AI for video games [13], for the game Settlers of Catan [43], for the game Einstein würfelt nicht [32], for the Voronoi game [6], for Havannah [31], for Amazons [28], and for various single player applications [39,41].

4 Horner's Rule for Multivariate Polynomials

We will now turn our attention to one such application domain: that of finding better variable orderings for applying Horner's rule to evaluate multivariate polynomials efficiently.

One area where finding solutions is important, and where good heuristics are hard to find, is equation solving for high energy physics (HEP). In this field large equations (often very large) are needed to be solved quickly. Standard packages such as Maple and Mathematica are often too slow, and scientists frequently use a specialized high-efficiency package called Form [27].

The research on MCTS in Form was started by attempting to improve the speed of the evaluation of multivariate polynomials. Applying MCTS to this challenge resulted in an unexpected improvement, first reported in [26]. Here we will stress further investigations into parameters that influence the search process.

Polynomial evaluation is a frequently occurring part of equation solving. Minimizing its cost is important. Finding more efficient algorithms for polynomial evaluation is a classic problem in computer science. For single variable polynomials, the classic Horner's rule provides a scheme for producing a computationally efficient form. It is conventionally named after William George Horner (1819) [20], although references to the method go back to works by the mathematicians Qin Jiushao (1247) and Liu Hui (3rd century A.D.). For multivariate polynomials Horner's rule is easily generalized but the order of the variables is

unspecified. Traditionally greedy approaches such as using (one of) the most-occurring variable(s) first are used. This straightforward approach has given remarkably efficient results and finding better approaches has proven difficult [9].

For polynomials in one variable, Horner's rule provides a computationally efficient evaluation form:

$$a(x) = \sum_{i=0}^{n} a_i x^i = a_0 + x(a_1 + x(a_2 + x(\cdots + x \cdot a_n))). \tag{1}$$

The rule makes use of the repeated factorization of the terms of the n-th degree polynomial in x. With this representation a dense polynomial of degree n can be evaluated with n multiplications and n additions, giving an evaluation cost of $2n$, assuming equal cost for multiplication and addition.

For multivariate polynomials Horner's rule must be generalized. To do so one chooses a variable and applies Eq. (1), treating the other variables as constants. Next, another variable is chosen and the same process is applied to the terms within the parentheses. This is repeated until all variables are processed. As a case in point, for the polynomial $a = y - 6x + 8xz + 2x^2yz - 6x^2y^2z + 8x^2y^2z^2$ and the order $x < y < z$ this results in the following expression

$$a = y + x(-6 + 8z + x(y(2z + y(z(-6 + 8z))))). \tag{2}$$

The original expression uses 5 additions and 18 multiplications, while the Horner form uses 5 additions but only 8 multiplications. In general, applying Horner's rule keeps the number of additions constant, but reduces the number of multiplications.

After transforming a polynomial with Horner's rule, the code can be further improved by performing a common subexpression elimination (CSE). In Eq. (2), the subexpression $-6+8z$ appears twice. Eliminating the common subexpression results in the code

$$\begin{aligned} T &= -6 + 8z \\ a &= y + x(T + x(y(2z + y(zT)))), \end{aligned} \tag{3}$$

which uses only 4 additions and 7 multiplications.

Horner's rule reduces the number of multiplications, CSE also reduces the number of additions.

Finding the optimal order of variables for applying Horner's rule is an open problem for all but the smallest polynomials. Different orders impact the cost evaluating the resulting code. Straightforward variants of local search have been proposed in the literature, such as most-occurring variable first, which results in the highest decrease of the cost at that particular step.

MCTS is used to determine an order of the variables that gives efficient Horner schemes in the following way. The root of the search tree represents the situation where no variables are chosen yet. This root node has n children. Each of these children represents a choice for variables in the trailing part of the order, and so on. Therefore, n equals the depth of the node in the search tree. A node at depth d has $n - d$ children: the remaining unchosen variables.

In the simulation step the incomplete order is completed with the remaining variables added randomly. This complete order is then used for applying Horner's rule followed by CSE. The number of operators in this optimized expression is counted. The selection step uses the UCT criterion with as score the number of operators in the original expression divided by the number of operators in the optimized one. This number increases with better orders.

In MCTS the search tree is built in an incremental and asymmetric way; see Fig. 3 for the visualization of a snap shot of an example tree built during an MCTS run. During the search the traversed part of the search tree is kept in memory. For each node MCTS keeps track of the number of times it has been visited and the estimated result of that node. At each step one node is added to the search tree according to a criterion that tells where most likely better results can be found. From that node an outcome is sampled and the results of the node and its parents are updated. This process is illustrated in Fig. 2. We will now again discuss the four steps of MCTS, as we use them for finding Horner orderings.

Selection. During the selection step the node which most urgently needs expansion is selected. Several criteria are proposed, but the easiest and most-used criterion is the UCT criterion [25]:

$$UCT_i = \langle x_i \rangle + 2C_p \sqrt{\frac{2 \log n}{n_i}}. \quad (4)$$

Here $\langle x_i \rangle$ is the average score of child i, n_i is the number of times child i has been visited, and n is the number of times the node itself has been visited. C_p is a problem-dependent constant that should be determined empirically. Starting at the root of the search tree, the most-promising child according to this criterion is selected and this selection process is repeated recursively until a node is reached with unvisited children. The first term of Eq. (4) biases nodes with previous high rewards (exploitation), while the second term selects nodes that have not been visited much (exploration). Balancing exploitation versus exploration is essential for the good performance of MCTS.

Expansion. The selection step finishes in a node with unvisited children. In the expansion step one of these children is added to the tree.

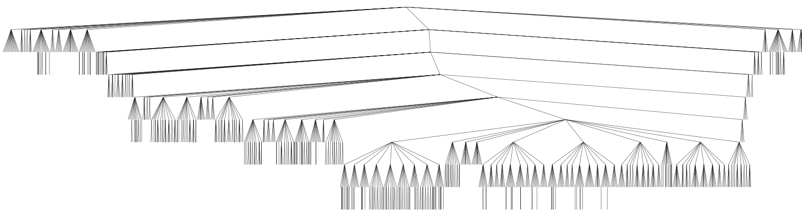

Fig. 3. Example of how an MCTS search expands the tree asymmettrically. Taken from a search for a Horner scheme.

Simulation. In the simulation step a single possible outcome is simulated starting from the node that has just been added to the tree. The simulation can consist of generating a fully random path starting from this node to a terminal outcome. In most applications more advanced programs add some known heuristics to the simulation, reducing the randomness. The latter typically works better if specific knowledge of the problem is available. In our MCTS implementation a fully random simulation is used. (We use domain-specific enhancements, such as CSE, but these are not search heuristics that influence the way MCTS traverses the search space.)

Backpropagation. In the backpropagation step the results of the simulation are added to the tree, specifically to the path of nodes from the newly added node to the root. Their average results and visit count are updated.

The MCTS cycle is repeated a fixed number of times or until the computational resources are exhausted. After that the best result found is returned.

Sensitivity to C_p and N. The performance of MCTS-Horner followed by CSE has been tested by implementing it in Form [26,27]. MCTS-Horner was tested on a variety of different multivariate polynomials, against the currently best algorithms. For each test-polynomial MCTS found better variable orders, typically with half the number of operators than the expressions generated by previous algorithms. The results are reported in detail in [26].

The experiments showed that the effectiveness of MCTS depends heavily on the choice for the exploitation/exploration constant C_p of Eq. (4) and on the number of tree expansions (N). In the remainder of this section we will investigate the sensitivity of the performance of MCTS-Horner to these two parameters.

When C_p is small, MCTS favors parts of the tree that have been visited before because the average score was good ("exploitation"). When C_p is large, MCTS favors parts of the tree that have not been visited before ("exploration").

Finding better variable orderings for Horner's rule is an application domain that allows relatively quick experimentation. To gain insight into the sensitivity of the performance in relation to C_p and to the number of expansions a series of scatter plots have been created.

The results of MCTS followed by CSE, with different numbers for tree expansions N as a function of C_p are given in Fig. 4 for a large polynomial from high energy physics, called HEP(σ). This polynomial has 5717 terms and 15 variables. The formula is typical for formulas that are automatically produced in particle reactions calculations; these formulas need to be processed further by a Monte Carlo integration program.

The number of operations of the resulting expression is plotted on the y-axis of each graph. The lower this value, the better the algorithm performs. The lowest value found for this polynomial by MCTS+CSE is an expression with slightly more than 4000 multiplication and addition operations. This minimum is achieved in the case of $N = 3000$ tree expansions for a value of C_p with $0.7 \lesssim C_p \lesssim 1.2$. Dots above this minimum represent a sub-optimal search result.

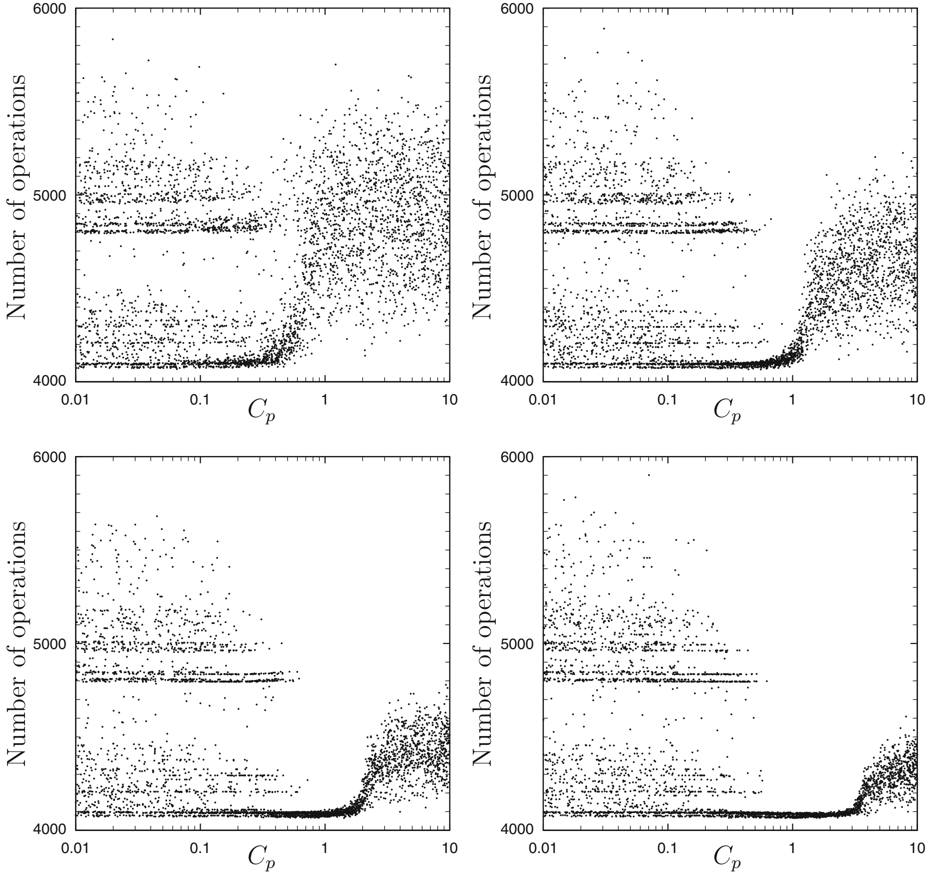

Fig. 4. Four scatter plots for $N = 300$, 1000, 3000, 10000 points per MCTS run. Each plot represents the average of 4000 randomized runs, for the HEP(σ) polynomial (see text).

For small values of the numbers of tree expansions MCTS cannot find a good answer. With $N = 100$ expansions the graph looks almost random (graph not shown). Then, as we move to 300 tree expansions per data point (left upper panel of Fig. 4), some clearer structure starts to emerge, with a minimum emerging at $C_p \approx 0.6$. With more tree expansions (see the other three panels of Fig. 4) the picture becomes clearer, and the value for C_p for which the best answers are found becomes higher, the picture appears to shift to the right. For really low numbers of tree expansions (see again upper left panel of Fig. 4) there is no discernible advantage of setting the exploitation/exploration parameter at a certain value. For slightly larger numbers of tree expansion, but still low (see upper right panel) MCTS needs to exploit each good result that it obtains. As the number of tree expansions grows larger (the two lower panels of Fig. 4) MCTS achieves

better results when its selection policy is more explorative. It can afford to look beyond the narrow tunnel of exploitation, to try a few explorations beyond the path that is known to be good, and to try to get out of local optima. For the graphs with tree expansions of 3000 and 10000 the range of good results for C_p becomes wider, indicating that the choice between exploitation/exploration becomes less critical.

For small values of C_p, such that MCTS behaves exploitatively, the method gets trapped in one of the local minima as can be seen from scattered dots that form "lines" in the left-hand sides of the four panels in Fig. 4. For large values of C_p, such that MCTS behaves exploratively, many of the searches do not lead to the global minimum found as can be seen from the cloud of points on the right-hand side of the four panels. For intermediate values of $C_p \approx 1$ MCTS balances well between exploitation and exploration and finds almost always an ordering for applying Horner's rule that is very close to the best one known to us.

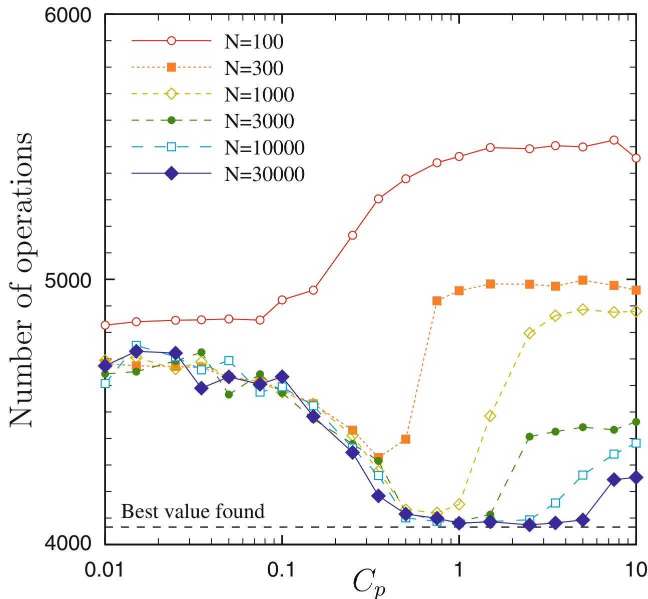

Fig. 5. Results for MCTS Horner orders as function of the exploitation/exploration constant C_p and of the number of tree expansions N. For $N = 3000$ (green line/solid bullets) the optimum for C_p is $C_p \approx 1$.

Results. The results of the test with HEP(σ) for different numbers of tree expansions are shown in Fig. 5, reproduced from [26]. For small numbers of tree expansions low values for the constant C_p should be chosen (less than 0.5). The search is then mainly in exploitation mode. MCTS quickly searches deep in the tree, most probably around a local minimum. This local minimum is

explored quite well, but the global minimum is likely to be missed. With higher numbers of tree expansions a value for C_p in the range $[0.5, 2]$ seems suitable. This range gives a good balance between exploring the whole search tree and exploiting the promising nodes. Very high values of C_p appear to be a bad choice in general, nodes that appeared to be good previously are not exploited anymore so frequently.

Here we note that these values hold for HEP(σ), and that different polynomials give different optimal values for C_p and N. Below we report on investigations with other polynomials.

Varying the Number of Tree Expansions. Returning to Fig. 4, let us now look closer at what happens when we vary the number of tree expansions N. In Fig. 4 we see scatter plots for 4 different values of N: 300, 1000, 3000, and 10000 expansions.

On the right side (larger values of C_p) of each plot we see a rather diffuse distribution. When C_p is large, exploration is dominant, which means that at each time we try a random (new) branch, knowledge about the quality of previously visited branches is more or less ignored. On the left side there is quite some structure. Here we give a large weight to exploitation: we prefer to go to the previously visited branches with the best results. Branches that previously had a poor result will never be visited again. This means that there is a large chance that we end up in a local minimum. The plots show indeed several of those (the horizontal bands). When there is a decent balance between exploration and exploitation it becomes likely that the program will find a good minimum. The more points we use the better the chance that we hit a branch that is good enough so that the weight of exploitation will be big enough to have the program return there. Hence, we see that for more points the value of C_p can become larger. We see also that on the right side of the plots using more evaluations gives a better smallest value. This is to be expected on the basis of statistics. In the limit, where we ask for more evaluations than there are leafs in the tree, we would obtain the best value.

Clearly the optimum is that we tune the value of C_p in such a way that for a minimum number of expansions we are still almost guaranteed to obtain the best result. This depends however very much on the problem. In the case of the formula of Fig. 4 this would be $C_p = 0.7$.

Repeating Runs of MCTS when C_p is Low. If we reconsider Fig. 4, i.e., we take a layman's look, we notice that at the left sides of the panels the distributions are nearly identical, independent of the number of tree expansions N. What does it mean? How can we influence the observed result? A new approach reads as follows. If, instead of 3000 expansions in a single run, we take, say, 3 times 1000 expansions and take the best result of those, the left side of the graphs is expected to become more favorable. The idea of repeated runs has been implemented in Form and the result is illustrated in Fig. 6. N is the number of tree expansions in a single MCTS run. R is the number of MCTS runs. The idea is best formulated by taking $N \times R$ as constant. In the experiments, we noticed a number of curious issues. We mention three of them explicitly. (1) When each

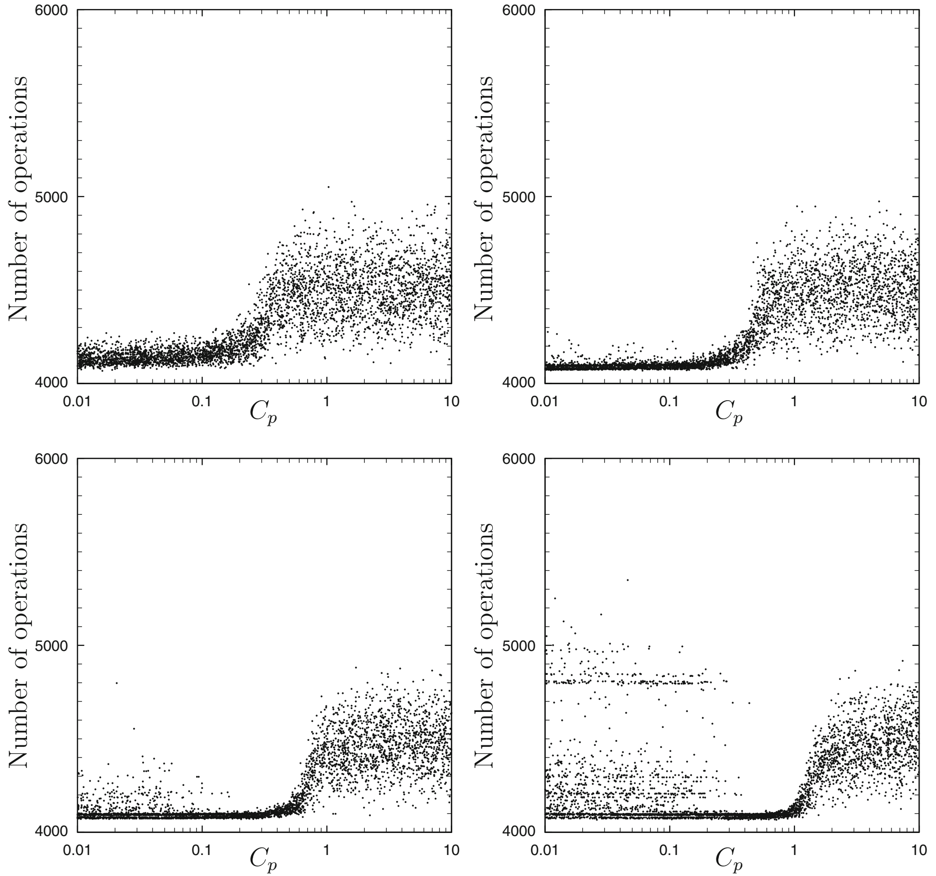

Fig. 6. Experiment for $N \times R$ constant. The polynomial HEP(σ) with 30 runs of 100 expansions, 18 runs of 167 expansions, 10 runs of 300 expansions, and 3 runs of 1000 expansions respectively. For comparison, the graph with a single run of $N = 3000$ can be found in Fig. 4, left bottom.

run has too few points, we do not find a suitable local minimum. (2) When a run has too few points the results revert to that of the almost random branches for large values of C_p. (3) The multiple runs cause us to lose the sharp minimum near $C_p = 0.7$, because we do not have anymore a correlated search of the tree. However, if we have no idea what would be a good value for C_p (i.e., we do not know where to start) it seems appropriate to select a value that is small and make multiple runs provided that the number of expansions N is sufficiently large for finding a reasonable local minimum in a branch of the tree.

Our next question then is: "What is a good value for the number of tree expansions per run?" Below we investigate and answer this question with the help of Fig. 7. We select a small value for C_p (0.01) and make runs for several values of the total number of tree expansions (with $N \times R = 1000, 3000, 5000$).

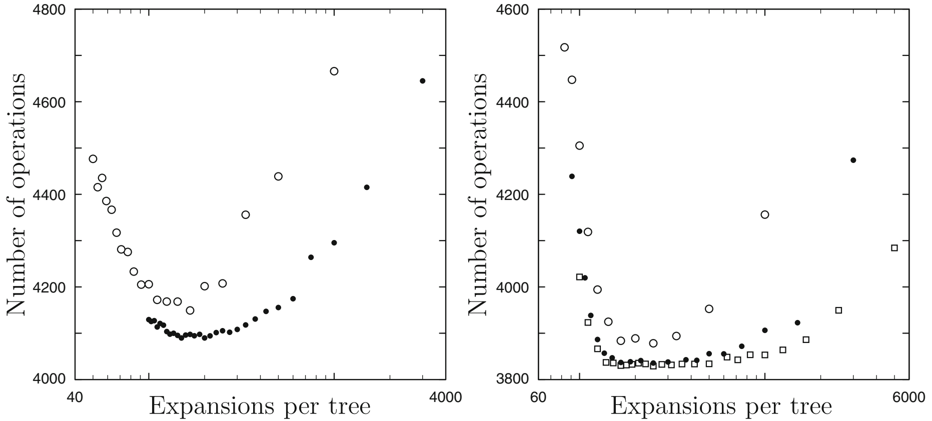

Fig. 7. The effect of repeated MCTS searches for low values of $C_p = 0.01$. The *product* of $N \times R$ (number of expansions times number of runs) is kept constant (1000 for the open circles, 3000 for the black circlesand 5000 for the open squares). The data points are averaged by running the simulations 50 times. The left graph is for the HEP(σ) formula and the right graph is for the 7-4 resultant.

The calculations in the left graph are for the formula HEP(σ) and in the right graph for another polynomial, which is the 7-4 resultant from [30]. The 7-4 resultant has 2562 terms and 13 variables. The minima for HEP(σ) coincide more or less around 165 expansions per tree. We believe that the number of expansions is correlated with the square of the number of variables. The reasoning is as follows. To saturate the nodes around a single path roughly takes $\frac{1}{2}n(n+1)$ expansions. The remaining expansions are used to search around this path and are apparently sufficient to find a local minimum. Returning to the right top panel of Fig. 6, it was selected with 18 runs of 167 expansions per tree (with the minimum of 165 expansions per tree in mind). For the formula involved this seems to be the optimum if one does not know about the value $C_p = 0.7$ or if one cannot run with a sufficient number of expansions to make use of its properties.

We have also made a few runs for the 7-5 and 7-6 resultants (also taken from [30]) and find minima around 250 and 300 respectively.[1] The results suggest that if the number of variables is in the range of 13 to 15 an appropriate value for the number of expansions is 200–250. This number will then be multiplied by the number of runs of MCTS to obtain an indication for the total number of tree expansions.

For reasons of comparison, we remark that similar studies of other physics formulas with more variables ($\mathcal{O}(30)$) show larger optimal values for the number of expansions per run and less pronounced local minima. Yet, also here, many

[1] The 7-5 resultant has 11380 terms and 14 variables, the 7-6 resultant has 43166 terms and 15 variables.

smaller runs may produce better results than a single large run, provided that the runs have more than a given minimum of tree expansions.

Future Work. This investigation into the sensitivity of (1) the number of tree expansions N, (2) the exploration/exploitation parameter C_P, and (3) the number of reruns of MCTS R has yielded interesting insights into the relationships between these parameters and the effect on the efficiency of MCTS in finding better variable orderings for multivariate polynomials to apply Horner's rule. We have used a limited number of polynomials for our experiments. In future work we will address the effect of different polynomials. In addition, it will be interesting to see if similar results can be obtained for other application domains, in particular for the game of Go.

5 Discussion

From the beginning of AI in 1950, chess has been called the Drosophila of AI. It was the testbed of choice. Many of the findings from decades of computer chess research have found their way to other fields, such as protein sequencing, natural language processing, machine learning, and high performance search [23]. After Deep Blue had defeated Garry Kasparov, research attention shifted to Go.

For Go, no good heuristic evaluation function seems to exist. Therefore, a different search paradigm was invented: MCTS. The two most prevailing characteristics are: no more minimax, and no need for a heuristic evaluation function. Instead, MCTS uses (1) the average of random playouts to guide the search, and (2) by balancing between exploration and exploitation, it appears to be able to detect by itself which areas of the search tree contain the green leaves, and which branches are dead wood. Having a "self-guided" (best-first) search, without the need for a domain-dependent heuristic, can be highly useful. For many other application domains the construction of a heuristic evaluation function is an obstacle, too. Therefore we expect that there are many other domains that could benefit from the MCTS technology, and, indeed, many other applications have already been found how to adapt MCTS to fit their characteristics (see, for example, [6,13,28,31,32,40,41,43]). In this paper one such adaptation has been discussed, viz. with Horner schemes. Finding better variable orders for applying the classic Horner's rule algorithm is an exciting first result [26], allowing easy investigation of two search parameters. It will be interesting to find out whether similar results can be found in MCTS as applied in Go programs, and other application domains.

References

1. Allis, V.: Searching for Solutions in Games and Artificial Intelligence. (Ph.D. thesis), University of Limburg, Maastricht, The Netherlands (1994)
2. Althöfer, I.: The origin of dynamic komi. ICGA J. **35**(1), 31–34 (2012)

3. Aoyama, T., Hayakawa, M., Kinoshita, T., Nio, M.: Tenth-Order QED Lepton Anomalous Magnetic Moment – Eighth-Order Vertices Containing a Second-Order Vacuum Polarization. e-Print: arXiv:1110.2826 [hep-ph] (2011)
4. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Mach. Learn. **47**(2), 235–256 (2002)
5. Bouzy, B., Helmstetter, B.: Monte-Carlo Go developments. In: van den Herik, H.J., Iida, H., Heinz, E.A. (eds.) ACG-10. IFIP, vol. 135, pp. 159–174. Springer, Boston (2003)
6. Bouzy, B., Métivier, M., Pellier, D.: MCTS experiments on the voronoi game. In: van den Herik, H.J., Plaat, A. (eds.) ACG 2011. LNCS, vol. 7168, pp. 96–107. Springer, Heidelberg (2012)
7. Brügmann, B.: Monte-Carlo Go. In: AAAI Fall symposium on Games: Playing, Planning, and Learning (1993). http://www.cgl.ucsf.edu/go/Programs/Gobble.html
8. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of Monte Carlo Tree Search Methods. IEEE Trans. Comput. Intell. AI Games **4**(1), 1–43 (2012)
9. Ceberio, M., Kreinovich, V.: Greedy algorithms for optimizing multivariate Horner schemes. ACM SIGSAM Bull. **38**, 8–15 (2004)
10. Chaslot, G., Saito, J.-T., Bouzy, B., Uiterwijk, J.W.H.M., van den Herik, H.J.: Monte-Carlo strategies for computer Go. In: Proceedings of the 18th BeNeLux Conference on Articial Intelligence, pp. 83–90 (2006)
11. Chaslot, G.M.J.-B., de Jong, S., Saito, J.-T., Uiterwijk, J.W.H.M.: Monte-Carlo tree search in production management problems. In: Proceedings of the BeNeLux Conference on Artificial Intelligence, Namur, Belgium, pp. 91–98 (2006)
12. Chaslot, G.M.J-B., Winands, M.H.M., Uiterwijk, J.W.H.M., van den Herik, H.J., Bouzy, B.: Progressive strategies for Monte-Carlo tree search. In: Wang, P., et al. (eds.) Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007), pp. 655–661. World Scientific Publishing Co., Pte. Ltd. (2007); New Mathematics and Natural Computation, vol. 4(3), pp. 343–357 (2008)
13. Chaslot, G.M.J-B., Bakkes, S., Szita, I., Spronck, P.: Monte-Carlo tree search: a new framework for game AI. In: Mateas, M., Darken, C. (eds.) Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference. AAAI Press, Menlo Park (2008)
14. Chaslot, G.M.J.-B., Hoock, J.-B., Rimmel, A., Teytaud, O., Lee, C.-S., Wang, M.-H., Tsai, S.-R., Hsu, S.-C.: Human-computer go revolution 2008. ICGA J. **31**(3), 179–185 (2008)
15. Chinchalkar, S.: An upper bound for the number of reachable positions. ICCA J. **19**(3), 181–182 (1996)
16. Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo tree search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M.J. (eds.) CG 2006. LNCS, vol. 4630, pp. 72–83. Springer, Heidelberg (2007)
17. Donkers, J.H.L.M., van den Herik, H.J., Uiterwijk, J.W.H.M.: Selecting evaluation functions in opponent model search. Theoret. Comput. Sci. (TCS) **349**(2), 245–267 (2005)
18. de Groot, A.D.: Het denken van den schaker, Ph. D. thesis in dutch (1946); translated in 1965 as “Thought and Choice in chess”, Mouton Publishers, The Hague (2nd edn. 1978). Freely available as e-book from Google (1946)
19. Enzenberger, M.: Evaluation in go by a neural network using soft segmentation. In: van den Herik, H.J., Iida, H., Heinz, E.A. (eds.) Advances in Computer Games. IFIP, vol. 135, pp. 97–108. Springer, Boston (2003)

20. Horner, W.G.: A new method of solving numerical equations of all orders, by continuous approximation. Philos. Trans. (R. Soc. Lond.) 109, 308–335 (1819); Reprinted with appraisal in Smith, D.E.: A Source Book in Mathematics, McGraw-Hill (1929); Dover reprint, vol. 2 (1959)
21. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT with patterns in monte-carlo go. Inst. Nat. Rech. Inform. Auto. (INRIA), Paris, Technical report (2006)
22. Hartmann, D.: How to extract relevant knowledge from grandmaster games. Part 1: Grandmaster have insights–the problem is what to incorporate into practical problems. ICCA J. **10**(1), 14–36 (1987)
23. van den Herik, H.J.: Informatica en het Menselijk Blikveld. Inaugural address Rijksuniversiteit Limburg, Maastricht, The Netherlands (1988)
24. Junghanns, A.: Are there practical alternatives to alpha-beta? ICCA J. **21**(1), 14–32 (1998)
25. Kocsis, L., Szepesvàri, C.: Bandit based monte-carlo planning. In: European Conference on Machine Learning, pp. 282–293. Springer, Berlin, Germany (2006)
26. Kuipers, J., Vermaseren, J.A.M., Plaat, A., van den Herik, H.J.: Improving multivariate Horner schemes with Monte Carlo tree search, July 2012. arXiv 1207.7079
27. Kuipers, J., Ueda, T., Vermaseren, J.A.M., Vollinga, J.: FORM version 4.0 (2012) (preprint). arXiv:1203.6543
28. Kloetzer, J.: Monte-Carlo opening books for amazons. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 124–135. Springer, Heidelberg (2011)
29. Landis, E.M., Yaglom, I.M.: About aleksandr semenovich kronrod. Russ. Math. Surv. **56**, 993–1007 (2001)
30. Leiserson, C.E., Li, L., Maza, M.M., Xie, Y.: Efficient evaluation of large polynomials. In: Fukuda, K., Hoeven, J., Joswig, M., Takayama, N. (eds.) ICMS 2010. LNCS, vol. 6327, pp. 342–353. Springer, Heidelberg (2010)
31. Lorentz, R.: Experiments with monte carlo tree search in the game of havannah. ICGA J. **34**(3), 140–149 (2011)
32. Lorentz, R.J.: An MCTS program to play einstein würfelt nicht!. In: van den Herik, H.J., Plaat, A. (eds.) ACG 2011. LNCS, vol. 7168, pp. 52–59. Springer, Heidelberg (2012)
33. Moch, S.-O., Vermaseren, J.A.M., Vogt, A.: Nucl. Phys. B688, B691, 101–134, 129–181 (2004); B724, 3–182 (2005)
34. Müller, M.: Computer Go. Artif. Intell. **134**(1–2), 145–179 (2002)
35. Pearl, J.: Asymptotical properties of minimax trees and game searching procedures. Artif. Intell. **14**(2), 113–138 (1980)
36. Pearl, J.: Heuristics Intelligent Search Strategies for Computer Problem Solving. Addison-WesleyPublishing Co, Reading (1984)
37. Plaat, A., Schaeffer, J., Pijls, W., de Bruin, A.: Best-First Fixed-Depth Minimax Algorithms. Artificial Intelligence **87**(1–2), 255–293 (1996)
38. Rivest, R.: Game-tree searching by min-max approximation. Artif. Intell. **34**(1), 77–96 (1988)
39. Rosin, C.D.: Nested rollout policy adaptation for monte carlo tree search. In: Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, IJCAI-2011, pp. 649–654 (2011)
40. Saito, J.-T., Chaslot, G.M.J.-B., Uiterwijk, J.W.H.M., van den Herik, H.J.: Monte-carlo proof-number search for computer go. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M.J. (eds.) CG 2006. LNCS, vol. 4630, pp. 50–61. Springer, Heidelberg (2007)

41. Schadd, M.P.D., Winands, M.H.M., van den Herik, H.J., Chaslot, G.M.J.-B., Uiterwijk, J.W.H.M.: Single-player monte-carlo tree search. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 1–12. Springer, Heidelberg (2008)
42. Stockman, G.C.: A minimax algorithm better than alpha-beta? Artif. Intell. **12**(2), 179–196 (1979)
43. Szita, I., Chaslot, G., Spronck, P.: Monte-Carlo tree search in settlers of catan. In: van den Herik, H.J., Spronck, P. (eds.) ACG 2009. LNCS, vol. 6048, pp. 21–32. Springer, Heidelberg (2010)
44. van der Werf, E.C.D., van den Herik, H.J., Uiterwijk, J.W.H.M.: Learning to score final positions in the game of Go. Theoret. Comput. Sci. **349**(2), 168–183 (2005)
45. van der Werf, E.C.D., Winands, M.H.M., van den Herik, H.J., Uiterwijk, J.W.H.M.: Learning to predict Life and Death from Go game records. Inf. Sci. **175**(4), 258–272 (2005)