Advances in computational methods for Quantum Field Theory calculations

Ruijl, B.J.G.

EXPRESSION SIMPLIFICATION 2

Monte Carlo integration is the preferred way to compute complicated cross sections of scattering processes [46–49]. The disadvantage of this method is that it converges slowly and thus requires a large number of samples. Sampling large multivariate polynomials is very time consuming, which makes Monte Carlo methods slow. In this chapter we investigate these challenges by answering

> **RQ1**: *To what extent can the number of arithmetic operations of large multivariate polynomials be reduced?*

The expressions that arise from Quantum Chromodynamics (QCD) are polynomials in many variables. The number of variables could range from a few to several hundreds. Analogously, the number of terms range from ten thousand terms to millions of terms [50]. In some extreme cases, the executable code that performs the evaluation of the expression could take up a few gigabytes. If we are able to reduce the number of operations required to evaluate these expressions, sampling becomes faster. As a result, Monte Carlo integrators can obtain precise results much faster.

We describe two methods to reduce the number of operations. The first is Horner's rule for multivariate polynomials, which is extracting variables outside brackets [51] (sec. 2.1.1). For multivariate expressions the order of these variables is called a *Horner scheme*. The second is called Common Subexpression Elimination (CSEE) [52], which is performed after the Horner scheme has been applied (sec. 2.1.2). We will investigate four methods (named H1 to H4) of finding a Horner scheme that yields a near-minimal number of operations after (1) the Horner scheme and (2) CSEE have been applied. The first three are based on tree search algorithms, and the fourth is based on local search algorithms.

H1. Following recent successes in expression simplification, we investigate tree search methods such as Monte Carlo Tree Search (MCTS), using Upper Confidence bounds applied to Trees (UCT) as best-child criterion [53] (sec. 2.3). However, UCT is not straightforward, as (1) it introduces an exploration-exploitation constant C_p that must be tuned, and (2) it does little exploration at the bottom of the tree.

H2. The second method is Nested Monte Carlo Search (NMCS) [54], described in sec. 2.3.2. NMCS does not not have the two issues of MCTS+UCT. However, since our evaluation function is quite expensive (3 seconds for one of our benchmark polynomials), NMCS performs (too) many evaluations to find a path in the tree, rendering it unsuitable for our simplification task.

H3. We make a modification to UCT (sec. 2.3.3), which we call Simulated Annealing - UCT (SA-UCT). SA-UCT introduces a dynamic exploration-exploitation parameter $T(i)$ that decreases linearly with the iteration number i. SA-UCT causes

a gradual shift from exploration at the start of the simulation to exploitation at the end. As a consequence, the final iterations will be used for exploitation, improving their solution quality. Additionally, more branches reach the final states, resulting in more exploration at the bottom of the tree. Moreover, we show that the tuning of C_p has become easier, since the region with appropriate values for C_p has increased by at least a tenfold [14]. The main contribution of SA-UCT is that this simplification of tuning allows for the results of our MCTS approach to be obtained much faster.

H4. We study local search methods (sec. 2.4). We find that the state space of Horner schemes is ideally suited for local search methods, since it is relatively flat and contains few local minima. This allows us to simplify expressions without expensive tuning of parameters. Our final algorithm is a variant of Stochastic Hill Climbing, which requires about ten times fewer samples than SA-UCT for similar results.

Our final Stochastic Hill Climbing algorithm is able to reduce the computation time of numerical integration from weeks to days or even hours. The methods have been implemented in Form and are used by at least one other research group.

The remainder of this chapter is structured as follows. First, we discuss methods for expression simplification in section 2.1. In section 2.2 we discuss the experimental setup. Next, we discuss tree search methods (method H1, H2, H3) in section 2.3. In section 2.4 we examine local search methods (method H4). We discuss performance and results in section 2.5. Finally, we present the chapter conclusion in section 2.6.

2.1 Horner schemes and common subexpression elimination

Expression simplification is a widely studied problem. Some examples are Horner schemes [52], common subexpression elimination (CSEE) [55], partial syntactic factorisation [56] and Breuer's growth algorithm [57]. Much research is put into simplifications using more algebraic properties, such as factorisation, especially because of its interest for cryptographic research [58, 59]. Simplification methods that depend on factorisation have the major problem of being notoriously slow. Horner schemes and CSEE do not require sophisticated mathematics: only the commutative and associative properties of the operators are used. The expressions we are considering often have more than twenty variables and more than a hundred thousand terms [50]. In this regime, computationally expensive methods are infeasible. Therefore, we consider using basic methods such as Horner schemes and CSEE.

2.1.1 *Horner Schemes*

Horner's rule reduces the number of multiplications in an expression by lifting variables outside brackets [51, 52, 60]. For multivariate expressions Horner's rule

can be applied sequentially, once for each variable. The order of this sequence is called the *Horner scheme*. Take for example:

$$x^2z + x^3y + x^3yz \to x^2(z + x(y(1 + z)) \, . \tag{4}$$

Here, first the variable x is extracted (i.e., x^2 and x) and second, y. The number of multiplications is now reduced from 8 to 4. However, the order x, y is chosen arbitrarily. One could also try the order y, x:

$$x^2z + x^3y + x^3yz \to x^2z + y(x^3(1 + z)) \, , \tag{5}$$

for which the number of multiplications is 6. Evidently, this is a sub-optimal Horner scheme. There are $n!$ orders of extracting variables, where n is the number of variables, and it turns out that the problem of selecting an optimal ordering is NP-hard [60].

A heuristic that works reasonably well is selecting variables according to how frequently a term with such a variable occurs ("occurrence order") [53]. A counter-example that shows that occurrence order is not always optimal is

$$x^{50}y + x^{40} + y + yz \, , \tag{6}$$

where extracting the most occurring variable y first causes the x^{50} and x^{40} to end up in different subparts of the polynomial, preventing their common terms from being extracted. We note that ordering the variables according to its highest power or to the sum of its powers in all the terms leads to other counter-examples.

2.1.2 *Common subexpression elimination*

The number of operations can be reduced further by applying common subexpression elimination (CSEE). This method is well known from the fields of compiler construction [55] and computer chess [61], where it is applied to much smaller expressions or subtrees than what we are considering here. Figure 2 shows an example of a common subexpression in a tree representation of an expression. The shaded expression $b(a + e)$ appears twice, and its removal means removing one superfluous addition and one multiplication.

CSEE is able to reduce both the number of multiplications and the number of additions, whereas Horner schemes are only able to reduce the number of multiplications.

2.1.3 *The evaluation function*

Writing an efficient evaluation function is important, since this function gets called many times. It consists of two parts: (1) applying the Horner scheme to an expression, and (2) removing common subexpressions. The Horner scheme is applied to the expression in FORM's internal format, which is a linear representation of a polynomial.

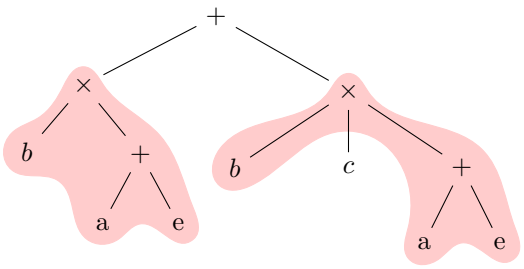


Figure 2: A common subexpression (shaded) in an associative and commutative tree representation.

In order to compute subexpressions efficiently, we transform the expression from a linear internal format to a tree. While building the expression tree (similar to figure 2), we store a hash of the branch that starts at the current node. Since the tree is built from the bottom up, we combine the hashes of the two subnodes. When the tree is completely built, we try to find common subexpressions. We keep track of each subtree we have come across in a hashtable. If a node is found, we skip exploring the subnodes, since it is a common subexpression. If we have not found the node before, we add the appropriate number of operations, add the node to the hashtable (the hash for the entire subtree is easily and quickly retrieved from the node itself), and continue to its children. A top-down approach has the benefit that entire subtrees can be easily identified as common subexpressions and double searches can be prevented. This process outweighs the time it takes to build the tree[1].

We apply two improvements to increase the number of common subexpressions: (1) when we extract the Horner scheme variable, we also extract a greatest common divisor (gcd). From our measurements, we have seen that this exposes more common subexpressions. (2) We store common subexpressions for exponentiations as well:

$$x^{20} = (x^{10})^2 = ((x^5)^2)^2 = x(((x^2)^2)^2)^2 \,. \tag{7}$$

2.1.4 *Interplay*

We note that there is an interplay between Horner schemes and CSEE: a certain "optimal" Horner scheme may reduce the number of multiplications the most, but may expose fewer common subexpressions than a "mediocre" Horner scheme. Thus, we need to find a way to obtain a Horner scheme that reduces the number of operations the most after both Horner and CSEE have been applied.

Finding appropriate Horner schemes is not a trivial task, for at least three reasons. First, there are no known local heuristics. For the Travelling Salesman Problem (TSP), the distance between two cities can be used as a heuristic [62], and more specialised

1 The tree can be allocated at once, since a close upper bound to the number of nodes is known.

	variables	terms	operations	eval. time (s)
res(7,4)	13	2561	29 163	0.001
res(7,5)	14	11 379	142 711	0.03
res(7,6)	15	43 165	587 880	0.13
res(9,8)	19	4 793 296	83 778 591	25.0
HEP(σ)	15	5716	47 424	0.008
HEP(F_{13})	24	105 058	1 068 153	0.4
HEP(F_{24})	31	836 009	7 722 027	3.0
HEP(b)	107	193 767	1 817 520	2.0

Table 1: The number of variables, terms, operations, and the evaluation time of applying a single Horner scheme and CSEE in seconds, for our eight (unoptimised) benchmark expressions. The time measurement is performed on a 2.4 GHz Xeon computer. All expressions fit in memory (192 GB).

heuristics are able to solve symmetric TSP instances with thousands of cities (a historic example is a TSP with 7397 cities [63, 64]). Second, the Horner scheme is applied to an expression. This means that the scheme has a particular context: the nth entry applies to the subexpressions that are created after the first $n-1$ entries in the Horner scheme have been applied to the expression. Third, the evaluation of a Horner scheme and CSEE is slow: for some benchmark expressions the evaluation took multiple seconds on a 2.4 GHz computer (see table 1). Since the evaluation is so slow, we have to find an optimisation algorithm that performs well with only a limited number of samples. Our attempted parallelisation of the evaluation function was unsuccessful, since the Horner scheme evaluation function is too fine-grained.

The time it takes to apply a Horner scheme is directly related to the number of variables and the number of terms in the expression. The common subexpression elimination time scales linearly with the number of operations. The difficulty of finding a good Horner scheme is related to (1) the size of the permutation space, i.e., related to the number of variables, but also to (2) the distribution of the variables in the terms. The composition of the variables affects the flatness of the state space and the occurrence of saddle points and local minima, as we shall see in section 2.4.3.

2.2 EXPERIMENTAL SETUP

We use eight large benchmark expressions, four from mathematics and four from real-world High Energy Physics (HEP) calculations. In table 1 statistics for the expressions are displayed. We show the number of variables, terms, operations, and the evaluation time of applying a Horner scheme and CSEE.

The expressions called res(7,4), res(7,5), res(7,6), and res(9,8) are resolvents and are defined by $\text{res}(m,n) = \text{res}_x(\sum_{i=0}^{m} a_i x^i, \sum_{i=0}^{n} b_i x^i)$, as described in [56]. The number of variables is $m+n+2$. The polynomial res(9,8) is the largest polynomial we have tested and has been included to test the boundaries of our hardware.

The High Energy Physics expressions represent scattering processes for the future International Linear Collider, a likely successor to the Large Hadron Collider [50]. A standard method of calculating the probability of certain collision events is by using perturbation theory. As a result, for each order of perturbations, additional expressions are calculated as corrections to previous orders of precision. The HEP polynomials of table 1 are second-order corrections to various processes.

HEP(σ) describes parts of the process $e^+e^- \to \mu^+\mu^-\gamma$, namely the collision of an electron and positron that creates a muon, an anti-muon, and a photon.

HEP(F_{13}), HEP(F_{24}), and HEP(b) are obtained from the process $e^+e^- \to \mu^+\mu^- u\bar{u}$, namely the collision of an electron and positron that creates a muon, anti-muon, an up-quark, and an up-antiquark. The results can be used to obtain next-generation precision measurements for electron-positron scattering [50].

These four HEP polynomials represent classes of polynomials with approximately the same behaviour.

2.3 TREE SEARCH METHODS

In this section we investigate three tree search methods. First, we review MCTS combined with UCT and discuss some issues in section 2.3.1. Second, we consider Nested Monte Carlo Search (NMCS) in section 2.3.2. Third, we construct a new best-child criterion called SA-UCT in section 2.3.3.

2.3.1 *Monte Carlo Tree Search*

Recently, Monte Carlo Tree Search has been shown to yield good quality Horner schemes [53]. We will describe its characteristics, so that we can see if we can improve its performance.

Monte Carlo Tree Search (MCTS) is a tree search method that has been successful in games such as Go, Hex, and other applications with a large state space [65, 66]. It works by selectively building a tree, expanding only branches it deems worthwhile to explore. MCTS consists of four steps, which are displayed in figure 3. The first step (3a) is the selection step, where a leaf or a not fully expanded node is selected according to some criterion (see below). Our choice is node z. In the expansion step (3b), a random unexplored child of the selected node is added to the tree (node y). In the simulation step (3(c)), the rest of the path to a final node is completed using random child selection. Finally a score Δ is obtained that signifies the score of the chosen path through the state space. In the backpropagation step (3d), this value is propagated back through the tree, which affects the average score (winrate) of a node (see below). The tree is built iteratively by repeating the four steps.

In the game of Go, each node represents a player move and in the expansion phase the game is played out, in basic implementations, by random moves. In the best performing implementations heuristics and pattern knowledge are used to

complement a random playout [65]. The final score is 1 if the game is won, and 0 if the game is lost. The entire tree is built, ultimately, to select the best first move.

For our purposes, we need to build a complete Horner scheme, variable by variable. As such, each node will represent a variable and the depth of a node in the tree represents the position in the Horner scheme. Thus, in figure 3(c) the partial Horner scheme is x,z,y and the rest of the scheme is filled in randomly with unused variables. The score of a path in our case, is the improvement of the path on the number of operations: the original number of operations divided by the number of operations after the Horner scheme and CSEE have been applied. We note that for our purposes the *entire* Horner scheme is important and not just the first variable.

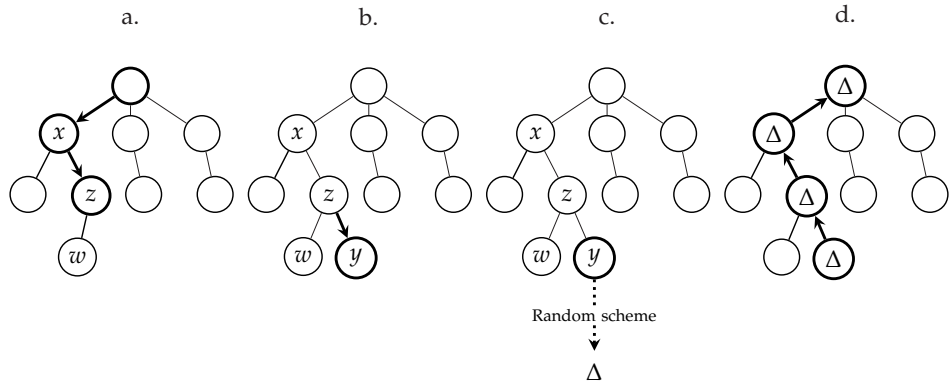


Figure 3: An overview of the four phases of MCTS: selection (a), expansion (b), simulation (c), and backpropagation (d). The selection of a not fully expanded node is done using the best child criterion (in our case UCT). Δ is the number of operations left in the final expression, after the Horner scheme and CSEE have been applied. See also [67].

In many MCTS implementations UCT (formula 8) is chosen as the selection criterion [67, 68]: [2]

$$\underset{\text{children } c \text{ of } s}{\operatorname{argmax}} \; \bar{x}(c) + 2C_p\sqrt{\frac{2\ln n(s)}{n(c)}}\,, \tag{8}$$

where c is a child node of node s, $\bar{x}(c)$ the average score of node c, $n(c)$ the number of times the node c has been visited, C_p the exploration-exploitation constant, and argmax the function that selects the child with the maximum value. This formula balances exploitation, i.e., picking terms with a high average score, and exploration, i.e., selecting nodes where the child has not been visited often compared to the parent. The C_p constant determines how strong the exploration term is: for high C_p the focus will be on exploration, and for low C_p the focus will be on exploitation.

There are two issues with the current form of the MCTS algorithm. The first was already mentioned and involves the tuning of the C_p parameter. Sometimes

2 The factor two outside and inside the square root of the UCT formula are a convention.

the region of C_p that yields good values is small, thus it may be computationally expensive to find an appropriate C_p. We return to this issue in section 2.3.3. Now we focus on the second issue: due to the natural asymmetry of trees, there is more exploration at nodes close to the root compared to the nodes deeper in the tree. Moreover, only a few branches are fully expanded to the bottom. Consequently, the final variables in the scheme will be filled out with the variables of a random playout. No optimisation is done at the end of these branches. As a result, if a very specific order of moves at the end of the tree gives a better outcome, this solution will not be found by MCTS. The issue can be partially reduced by adding a new parameter that specifies whether the Horner scheme should be constructed in reverse, so that the variables selected near the root of the tree are actually the last to be extracted [14, 53].

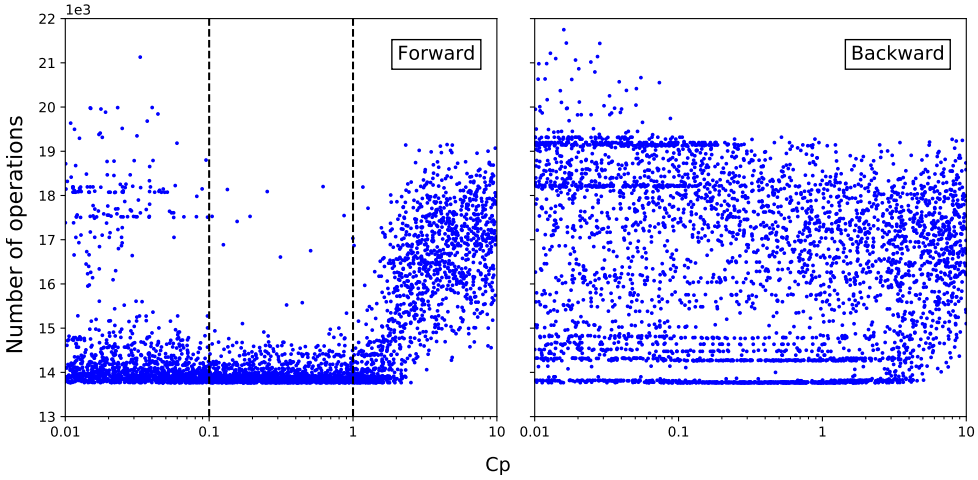


Figure 4: res(7,5): differences between forward (left) and backward (right) Horner schemes, at $N = 1000$ tree updates with SA-UCT. Forward Horner schemes generate a region of C_p where the number of operations is near the global minimum, whereas backward schemes have multiple high-density local minima and a diffuse region.

In figure 4 the difference between a forward and a backward MCTS search with 1000 updates is shown for the polynomial res(7,5) in scatter plot. For the forward construction, we see that there is a region in C_p where the results are good: the lowest measured value is found often. However, the backward scheme does not have a similar range. For other polynomials, it may be better to use the backward scheme, as is the case for HEP(σ) and HEP(F_{13}). Currently, there is no known way to predict whether forward or backward construction should be used. Thus, this introduces an extra parameter to our algorithm.

Even though the scheme direction parameter reduces the problem somewhat, the underlying problem that there is little exploration at the end of the tree still remains.

To overcome the issues of tuning C_p and the lack of exploration, we study a related tree search method called Nested Monte Carlo Search in the next section.

2.3.2 *Nested Monte Carlo Search*

Nested Monte Carlo Search (NMCS) addresses the issue of the exploration bias towards the top of the tree by sampling all children at every level of the tree [54]. In its simplest form, called a level-1 search, a random playout is performed for each child of a node. Next, the child with the best score is selected, and the process is repeated until one of the end states is reached. This method can be generalized to a level k search, where the above process is nested: a level k search chooses the best node from a level $k-1$ search performed on its children. Thus, if the NMCS level is increased, the top of the tree is optimised with greater detail. Even though NMCS makes use of random playouts, it does so at every depth of the tree as the search progresses. Consequently, there is always exploration near the end of the tree.

In figure **??** the results for level 2 NMCS are shown for HEP(σ). The average number of operations is 4189 ± 43 (a standard deviation of 43). To compare the performance of NMCS to that of MCTS, we study the run-time. Since more than 90% of the run-time is spent on the evaluation function, we may compare the number of evaluations instead. A level-2 search for HEP(σ) takes 8500 evaluations. In order to be on a par with MCTS, the score should have been between MCTS 1000 and MCTS 10 000 iterations. However, we see that the score is higher than MCTS with 1000 iterations and thus we may conclude that the performance of NMCS is inferior to MCTS for HEP(σ).

We have performed similar experiments with NMCS on other polynomials, but the resulting average number of operations were always greater than MCTS's. The reason is likely because we select a low level k: a level-1 search selects the best child using one sample per child, a process which is highly influenced by chance. However, there are some performance issues with using a higher k. To analyse these, we investigate the number of evaluations that a level n search requires.

The form of our tree is known, since every variable should appear only once. This means that there are n children at the root, $n-1$ children of children, and $n-d$ children at depth d. Thus a level-1 search takes $n + (n-1) + (n-2) + \ldots + 1 = n(n+1)/2$ evaluations. It can be shown that a level k search takes S_n^{k+n}, where S is the Stirling Number of the First Kind. This number grows rapidly: if $k = 1$ and $n = 15$, the number of evaluations is 120, and for level $k = 2$, it takes 8500 evaluations. For an expression with 100 variables, a level-1 search takes 5050 evaluations, and a level-2 search takes 13 092 125 evaluations.

The evaluation function is expensive for our polynomials: HEP(F_{13}) takes about 0.4 second per evaluation and HEP(F_{24}) takes 3.0 seconds. We have experimented with paralellising the evaluation function, but due to the fine-grained nature of the evaluation function, this was unsuccessful. For HEP(F_{24}) a million iterations will be slow, hence for practical reasons we have only experimented with a level-1 search.

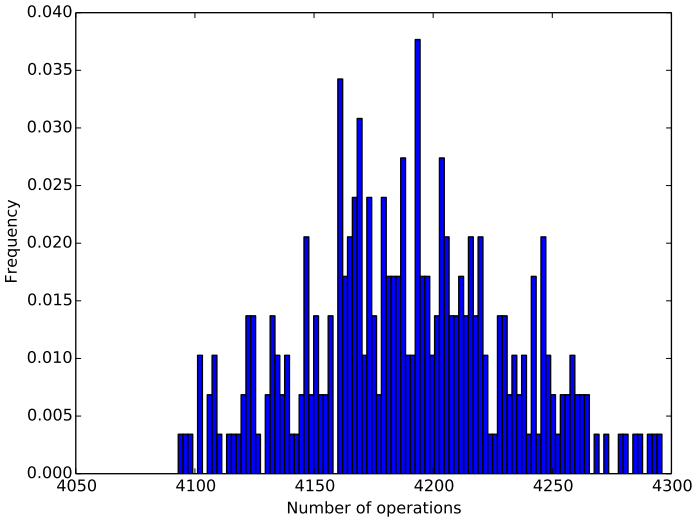


Figure 5: NMCS level 2 for HEP(σ), taking 8500 evaluations. This is comparable in CPU time to MCTS with 8500 runs. The number of operations is 4189 ± 43, averaged over 292 samples.

The domains in which NMCS performed well, such as Morpion Solitaire and SameGame, have a cheap evaluation function relative to the tree construction [54]. If the evaluation function is expensive, even the level-1 search takes a long time.

Based on the remarks above, we may conclude that for polynomials with a large number of variables, NMCS becomes infeasible.

2.3.3 *SA-UCT*

Since NMCS is unsuitable for simplifying large expressions, we return our focus to MCTS, but this time on the UCT best child criterion. We now consider the role of the exploration-exploitation constant C_p. We notice that in the first iterations of the simulation there is as much exploration as there is in the final iterations, since C_p remains constant throughout the search. For example, the final 100 iterations of a 1000 iterations MCTS run are used to explore new branches even though we know in advance that there is likely not enough time to reach the final nodes. Thus we would like to modify the C_p to change during the simulation to emphasise exploration early in the search and emphasise exploitation towards the end.

We introduce a new, dynamic exploration-exploitation parameter T that decreases linearly with the iteration number:

$$T(i) = C_p \frac{N - i}{N} \,, \tag{9}$$

where i is the current iteration number, N the preset maximum number of iterations, and C_p the initial exploration-exploitation constant at $i = 0$.

We modify the UCT formula to become:

$$\underset{\text{children } c \text{ of } s}{\operatorname{argmax}} \; \bar{x}(c) + 2T(i)\sqrt{\frac{2\ln n(s)}{n(c)}}\,, \tag{10}$$

where c is a child of node s, $\bar{x}(c)$ is the average score of child c, $n(c)$ the number of visits at node c, and $T(i)$ the dynamic exploration-exploitation parameter of formula (9).

The role of T is similar to the role of the temperature in Simulated Annealing [69]: in the beginning of the simulation there is much emphasis on exploration, the analogue of allowing transitions to energetically unfavourable states. During the simulation the focus gradually shifts to exploitation, analogous to annealing. Hence, we call our new UCT formula "Simulated Annealing UCT (SA-UCT)".

In the past related changes have been proposed. For example, Discounted UCB [70] and Accelerated UCT [71] both modify the average score of a node to discount old wins over new ones. The difference between our method and past work is that the previous modifications alter the importance of exploring based on the history and do not guarantee that the focus shifts from exploration to exploitation. In contrast, this work focuses on the exploration-exploitation constant C_p and on the role of exploration during the simulation.

We implemented four improvements over UCT. (1) The final iterations are used effectively. (2) There is more exploration in the middle and at the bottom of the tree. This is due to more nodes being expanded at lower levels, because the T is lowered. As a consequence, we see that (3) more branches reach the end states. As a result, (4) there is exploration near the bottom, where there was none for the random playouts.

In order to analyse the effect of SA-UCT on the fine-tuning of C_p (the initial temperature), we perform a sensitivity analysis on C_p and N [14]. In figure 6 the results for the res(7,5) polynomial with 14 variables are displayed. Horizontally, we have C_p, and vertically we have the number of operations (where less is better). A total of 4000 MCTS runs (dots) are performed for a C_p between 0.001 and 10. On the left we show the results for UCT and on the right for SA-UCT. We identify a region with local minima for low C_p, a diffuse region for high C_p and an intermediate region in C_p where good results are obtained. This region becomes wider if the number of iterations N increases, for both SA-UCT and UCT.

However, we notice that the intermediate region is wider for SA-UCT, compared to UCT. For $N = 1000$, the region is $[0.1, 1.0]$ for SA-UCT, whereas it is $[0.07, 0.15]$ for UCT. Thus, SA-UCT makes the region of interest about 11 times larger for res(7,5). This stretching is not just an overall rescaling of C_p: the uninteresting region of low C_p did not grow significantly. For $N = 3000$, the difference in width of the region of interest is even larger.

In figure 7, we show a similar sensitivity analysis for HEP(σ) with 15 variables. We identify the same three regions and see that the region of interest is $[0.5, 0.7]$ for

UCT and $[0.8, 5.0]$ for SA-UCT at $N = 1000$. This means that the region is about 20 times larger relative to the uninteresting region of low C_p, which grew from 0.5 to 0.8. We have performed experiments on six other expressions, and we obtain similar results [14].

On the basis that SA-UCT compared to UCT (1) decreases the sensitivity to C_p by an order of magnitude, and (2) produces Horner schemes of the same quality, we may conclude that SA-UCT reduces the fine-tuning problem without overhead.

2.4 STOCHASTIC LOCAL SEARCH

In the preceding sections we have concluded that MCTS with SA-UCT is able to find Horner schemes that yield a smaller number of operations than the naive occurrence order schemes. For some polynomials, MCTS yields reductions of more than a factor 24. However, in section 2.3.1 we have seen that there are some intrinsic shortcomings to using a tree representation, especially if the depth of the search tree becomes (too) large. We noticed that many branches do not reach the bottom when there are more than 20 variables (we remind the reader that the problem depth is equivalent to the number of variables) as is the case with many of our expressions. MCTS determines the scores of a branch by performing a random play-out. If the branch is not constructed all the way to the bottom, the final nodes are therefore random (no optimisation). For Horner schemes, the *entire* scheme is important, so sub-optimal selection of variables at the end of the scheme can have a significant impact.

The issue of poor optimisation at the bottom of the tree motivated us to look for a method that is symmetric in its optimisation: both the beginning and the end should be optimised equally well. In this section we (re)consider which class of algorithms is best suited for the Horner scheme problem. The Horner scheme problem belongs to the class of permutation problems. Many algorithms for optimising permutation problems have been suggested in the literature, such as Stochastic Hill Climbing (SHC) [72], Simulated Annealing (SA) [69], Tabu Search [62], Ant Colony Optimisation [73], and Evolutionary Algorithms [74]. Since measurements take weeks per algorithm, we limit ourselves to two. In [11] we provide qualitative motivations for focusing on SHC and SA. In summary, the absence of heuristics and the high cost of sampling to tune parameters make the other options less interesting.

The structure of section 2.4 is as follows. In section 2.4.1 we consider the differences between SHC and SA, in section 2.4.2 we find an appropriate neighbourhood structure, and in 2.4.3 we study the state space properties for given neighbourhood structures.

2.4.1 *SHC versus SA*

A Stochastic Hill Climbing procedure, also known as iterative improvement local search, has two parameters: (1) the number of iterations N, and (2) the neighbourhood structure, which defines the transition function [75]. We consider the

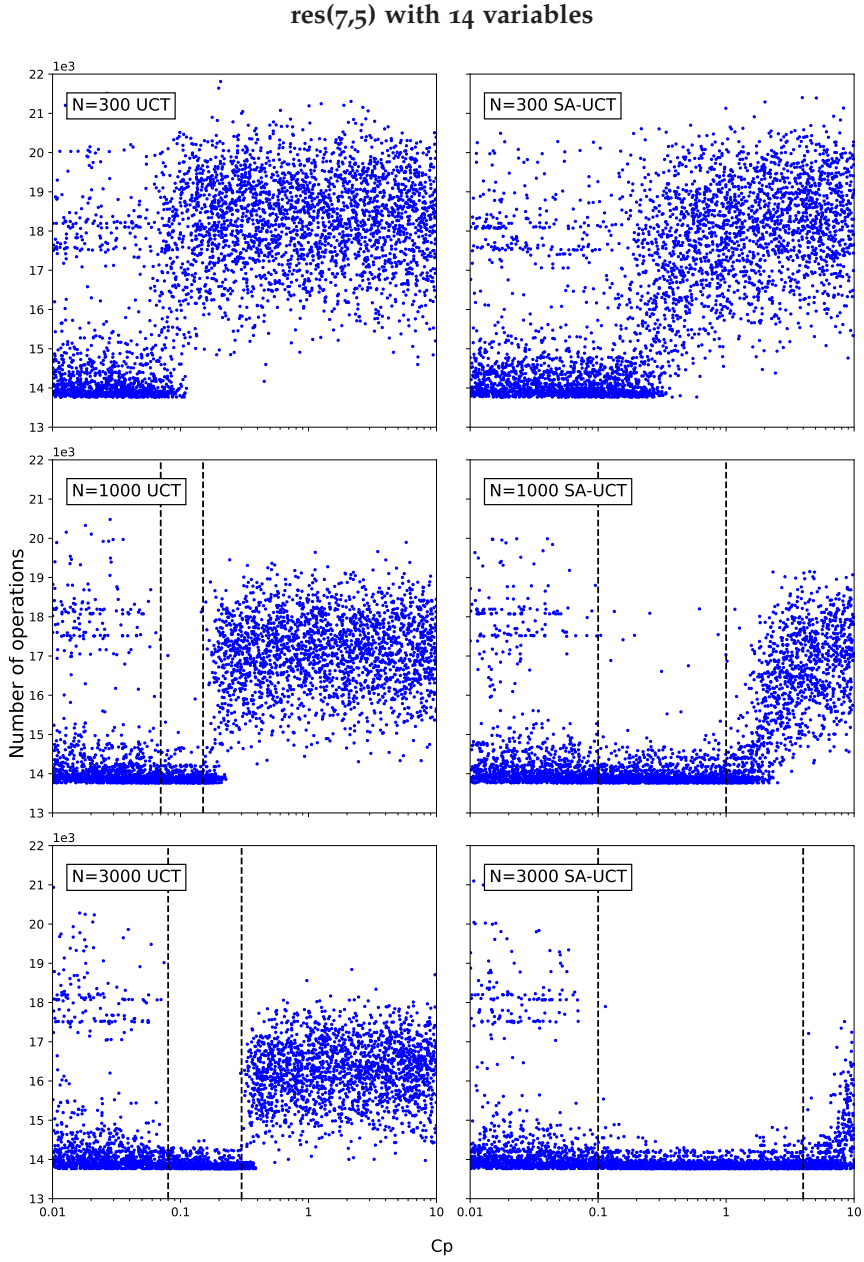


Figure 6: res(7,5) polynomial with 14 variables: on the x-axis we show C_p and on the y-axis the number of operations. A lower number of operations is better. On the left, we show UCT with constant C_p and on the right we show SA-UCT where C_p is the starting value of T. Vertically, we increase the tree updates N from 300, to 1000, to 3000. As indicated by the dashed lines, an area with an operation count close to the global minimum appears, as soon as there are sufficient tree updates N. This area is wider for SA-UCT than for UCT.

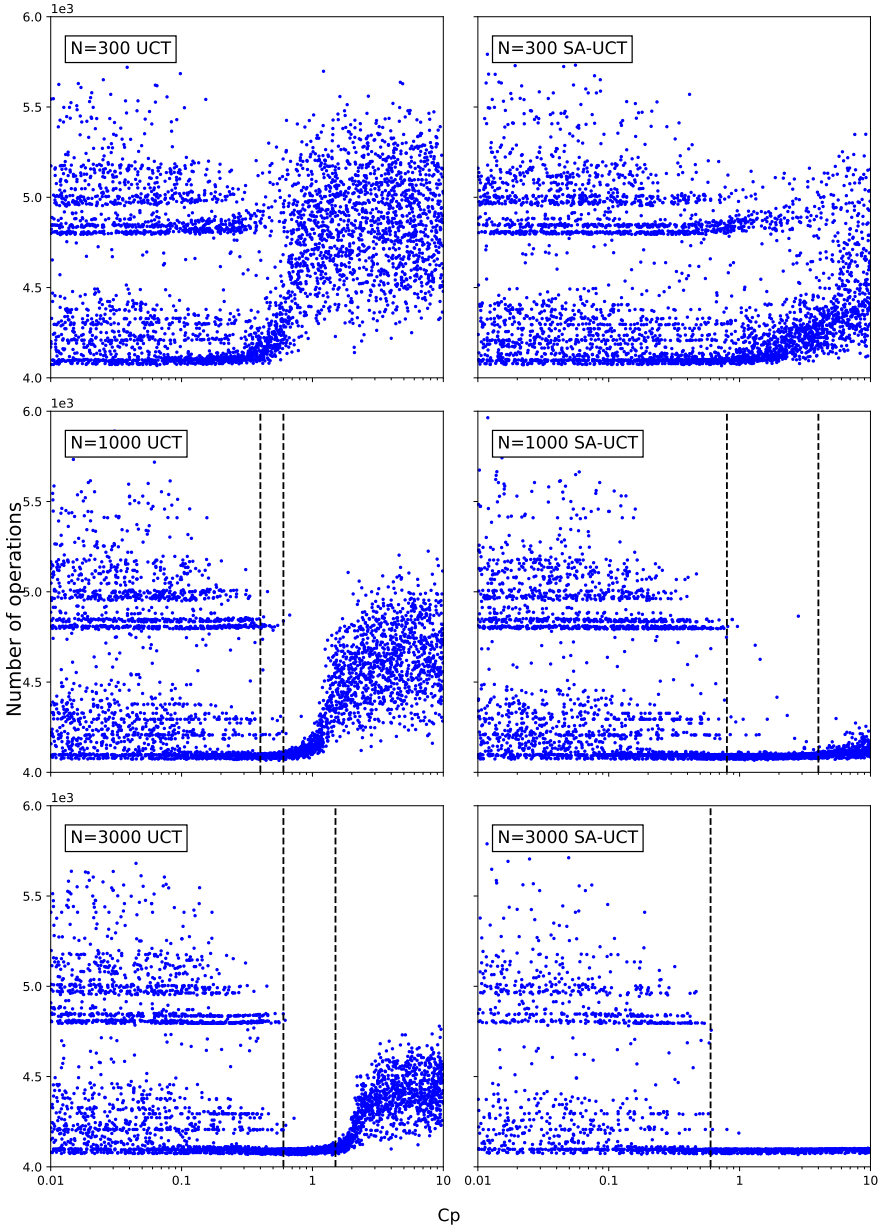


Figure 7: HEP(σ) with 15 variables: on the x-axis we show C_p and on the y-axis the number of operations. A lower number of operations is better. On the left, we show UCT with constant C_p and on the right we show SA-UCT where C_p is the starting value of T. Vertically, we increase the tree updates N from 300, to 1000, to 3000. As indicated by the dashed lines, an area with an operation count close to the global minimum appears, as soon as there are sufficient tree updates N. This area is wider for SA-UCT than for UCT.

neighbourhood structure the most important parameter, since it is tunable. A Stochastic Hill Climbing procedure only moves to a neighbour if the evaluation score (number of operations) is improved. As a consequence, SHC could get stuck in local minima. Therefore, we consider to use Simulated Annealing instead of SHC, since SA has the ability to escape from local minima.

Simulated Annealing (SA) is a popular generalisation of SHC. It has four additional parameters with respect to SHC, namely (3) the initial temperature T_i, (4) the final temperature T_f, (5) the acceptance scheme, and (6) the cooling scheme [69]. The temperature governs the probability of accepting transitions with an energy higher than the energy of the current state. The cooling scheme governs how fast and in what way the temperature is decreased during the simulation (linearly, exponentially, etc.). Exponential cooling is frequently used. The acceptance scheme is most often the Boltzmann probability $\exp(\Delta E/T)$, that defines the probability of selecting a transition to an inferior state, given the difference in evaluation score ΔE. The number of iterations (1) of SA is not independent of the other parameters, as it can be computed from the initial temperature, final temperature, and cooling scheme. Most often, the search is started from a random position in the state space. In our application, we start from a random permutation of the variables. A basic SA algorithm is displayed in Algorithm 1.

$s \leftarrow$ random state, best $\leftarrow s$, $T \leftarrow T_i$;
while $T > T_f$ **do**
  $s' \leftarrow$ random neighbour(s);
  if $e^{(E_s - E_{s'})/T} > rand(0,1)$ **then**
    $s \leftarrow s'$; `/* Accept new state */`
    if $E_s < E_{best}$ **then**
      best $\leftarrow s$;
    end
  end
  $T \leftarrow \alpha T$; `/* Annealing */`
end
return best;

Algorithm 1: A basic SA implementation, with a neighbourhood function (2), initial temperature T_i (3), final temperature T_f (4), a Boltzmann acceptance scheme (5), and exponential cooling with cooling parameter α (6). The number of iterations (1) is $\log_\alpha(T_f/T_i)$.

For SA we consider the following. If the initial temperature is high, then transitions to inferior states are permitted, allowing an escape from local minima. In order to determine the effect of the initial temperature T_i on the results, we will perform a sensitivity analysis. We use Boltzmann probability as the acceptance scheme, exponential cooling, a final temperature of 0.01, $N = 1000$ iterations, and a swap neighbourhood structure (for a visualisation, see figure 9).

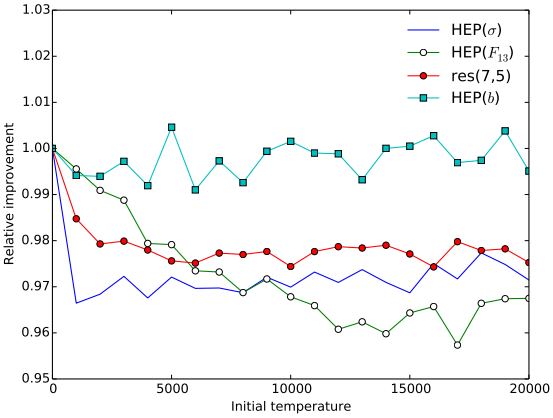


Figure 8: The relative improvement (lower is better) of the number of operations for a given initial temperature T_i, compared to $T_i = 0$. Each data point is the average of more than 100 SA runs with 1000 iterations, and a swap neighbourhood structure. We show the expressions HEP(σ), HEP(F_{13}), res(7,5), and HEP(b). The number of operations is only slightly influenced by the initial temperature, since the best improvement over $T_i = 0$ is smaller than 5%.

In figure 8 we show the relative improvement (lower is better) of the number of iterations for a given initial temperature T_i compared to $T_i = 0$ for the expressions HEP(σ), HEP(F_{13}), res(7,5), and HEP(b). Naturally, for $T_i = 0$, the relative improvement to itself is 1. For all expressions except HEP(b), we see a region where the improvement is largest: for HEP(σ) it is approximately $[1000, 7000]$, for HEP(F_{13}) it is $[12\,000, 17\,000]$ and for res(7,5) it is $[5000, 20\,000]$. This improvement is less than 5%. For higher T, too many transitions to inferior states are accepted to obtain good results. HEP(b) seems to be independent of the initial temperature. The fluctuations of 1% are statistical fluctuations.

The difference between the best results for all the expressions in figure 8 and the result at $T = 0$ is less than 5%. Since a $T_i = 0$ SA search is effectively an SHC search, this means that an almost parameterless Stochastic Hill Climbing (SHC) is able to obtain results that are only slightly inferior. This is surprising, since the way SHC traverses the state space is different from the way by SA. We here reiterate once more, SHC can get stuck in local minima, whereas SA has the possibility to escape. Furthermore, if a saddle point is reached, SA is able to climb over the hill, whereas SHC has to walk around the hill in order to escape. In subsection 2.4.3 we will show that local minima are uncommon, and that most of them are actually saddle points (i.e., local "minima" with a way to escape). Consequently, SA performs slightly better not because it can escape from local minima, but because, for some

polynomials, walking over a saddle point (SA) is slightly faster to find better states than trying to circumvent the saddle point (SHC).

The reason why we prefer SHC over SA is that (1) the fundamental algorithmic improvement of SA – the ability to escape from local minima – is not used in practice, as we will see in subsection 2.4.3, and (2) tuning the SA parameters is expensive. Several methods have been suggested to tune the initial temperature, such as [76] and [77], but they often take several hundred iterations to obtain reliable values (which is quite expensive in our case). The small benefit of SA can be obtained by three other ways. First, by performing SHC runs in parallel (see section 2.4.2), second, by increasing the number of iterations, and third by selecting an initial temperature based on previous information such as figure 8. Using these three optimisations, a small improvement is obtained without increasing the run time.

2.4.2 *Neighbourhood structure*

The main parameter of SHC is the neighbourhood structure. Choosing an appropriate neighbourhood structure is crucial, since it determines the shape of the search space and thus influences the search performance. In [78] it is observed that the neighbourhood structure can have a significant impact on the quality of the solutions for the Travelling Salesman Problem, the Quadratic Assignment Problem, and the Flow-shop Scheduling Problem.

There are many neighbourhood structures for permutation problems such as Horner schemes. For example, a transition could swap two variables in the Horner scheme or move a variable in the scheme. However, there are also neighbourhood structures that involve changing larger structures. Figure 9 gives an overview of four basic transitions from which others can be constructed. From top to bottom, it shows (a) a single swap of two variables in the scheme, (b) a shift of a variable, (c) a shift of a sublist, and (d) a mirroring of a sublist. At each iteration of SHC, a transition to a randomly chosen neighbour is proposed. For the single swap transition, this involves the selection of two random variables in the scheme.

To examine which neighbourhood structure performs best for Horner schemes, we investigate seven (combinations of) neighbourhood structures, viz. (1) a single swap, (2) two consecutive swaps, (3) three consecutive swaps, (4) a shift of a single variable, (5) mirroring of a sublist, (6) a sublist shift (which we call 'many shift'), and (7) mirroring and/or shifting with an equal probability (which we call 'mirror shift'). Swapping multiple times in succession allows for faster traversal of the state space, but also runs the risk to miss states. Moreover, we have tested hybrid transitions. For instance, we performed two consecutive swaps in the first half of the simulation and resorted to single swaps for the latter half. However, we found that these combinations did not perform better. In order to present clear plots, we have omitted the plots resulting from these combinations.

Below we discuss (A) the measuring quality, (B) detailed results for res(7,6) and HEP(σ), (C) detailed results for HEP(F_{13}) and HEP(b), and (D) combined results.

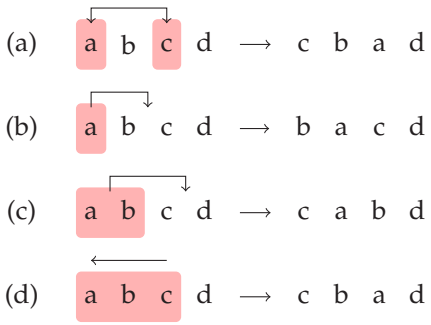


Figure 9: The elementary neighbourhood structures we use. From top to bottom: (a) a single swap, (b) a single shift, (c) shift of a sublist, and (d) mirroring of a sublist.

(A) Measuring quality

We now start investigating two methods of measuring the quality of a neighbourhood structure by: (A1) the average number of operations obtained by using a neighbourhood structure, and (A2) the lowest number of operations after performing *several* runs.

A1 Figure 10 shows the distribution of the number of operations of the expression HEP(F_{13}) after 10 000 SHC runs with the neighbourhood structure that exchanges two random variables. The average of this distribution is somewhere in the middle, but the actual values that one will measure will be either near 51 000 or near 62 000. Thus, the average is not an appropriate measure.

A2 So, we decided to measure the lowest score of several runs (A2), because in practice SHC is run in parallel, and so the results are more in line with those from practical applications. Thus, we are interested in the neighbourhood structure that has the lowest expected value of the minimum of k measurements. Here, we can use the expected value $\mathrm{E}\left[\min\left(X_0, \ldots, X_{k-1}\right)\right]$:

$$V_0 + \sum_{t=0}^{L-2} \left(V_{t+1} - V_t\right) \left(1 - \mathrm{cdf}(D, t)\right)^k \tag{11}$$

where k is the number of measurements, X_n is the score of the nth measurement, t is an index in the discrete distribution, V_t is the number of operations at t, D_t is the probability of outcome V_t, L is the number of possible outcomes, and cdf the cumulative distribution function. We shall denote the expected value of the minimum of k runs by $\mathrm{E}_{\min,k}$.

Because the number of measurements k is in the exponent in eq. (11), $\mathrm{E}_{\min,k}$ decreases exponentially with k and finally converges to V_0. As a consequence,

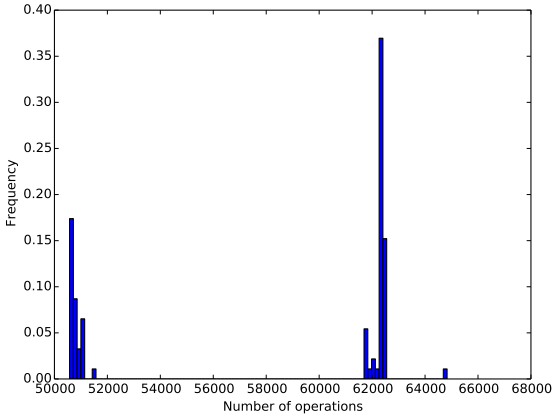


Figure 10: HEP(F_{13}) expression with 10 000 runs and 1 swap as a neighbourhood structure. Typical for our domain is that there are often two or more spikes. If the simulation is run multiple times, the probability of finding a value close to the minimum is high.

neighbourhood structures with a high standard deviation are more likely to achieve better results, since at high k the probability of finding a low value at least once is high. We found that four parallel runs ($k = 4$) yielded good results.

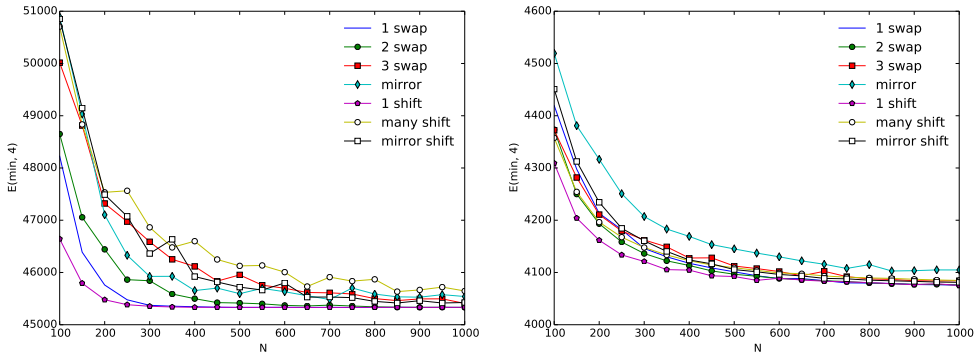
The results for res(7,4), res(7,5), and HEP(F_{24}) are similar, and are omitted for brevity (see [11]). Polynomial res(9,8) is too time consuming for such a detailed analysis (it would take around 35 days to collect all data).

(B) Results for res(7,6) and HEP(σ)

In figure 11a the performance of the neighbourhood structures for the expression res(7,6) is shown. We see that shifting a single variable ('1 shift') has the best performance at a low number of iterations N, followed by 2 consecutive swaps ('2 swap'). At around $N = 900$ all neighbourhood structures have converged. Thus, from $N = 900$ onward it does not matter which structure is chosen.

In figure 11b we show the performance of the neighbourhood structures for HEP(σ). We see that '1 shift' has the best performance at low N. At $N = 600$, all the neighbourhood structures have converged. The characteristics of this plot are similar to those of res(7,6).

We suspect that for a small state space, i.e., a small number of variables, there is not much difference between the neighbourhood structures, since the convergence occurs quite early (below $N = 1000$). Therefore, we look at two expressions with more variables: HEP(F_{13}) with 24 variables, and HEP(b) with 107 variables.



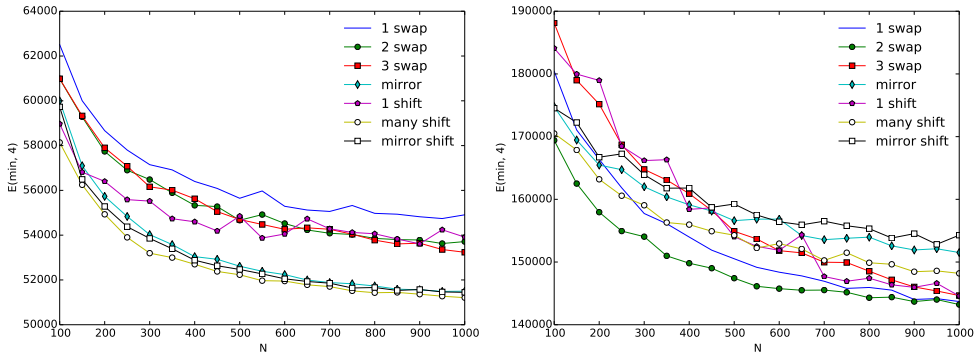
(a) res(7,6) (b) HEP(σ)

Figure 11: The expected number of operations of the minimum of four SHC runs with N iterations for res(7,6) (left), and HEP(σ) (right). For res(7,6) the 1 shift performs best for low N, followed by the 2 swap. All the neighbourhood structures converge at $N = 900$. For HEP(σ), we see that 1 shift has the best performance at low N. At $N = 600$, all the neighbourhood structures have converged. The characteristics of the two plots are similar.

(C) Results for HEP(F_{13}) and HEP(b)

In figure 12a we show the results for HEP(F_{13}). We see that all the neighbourhood structures that involve small changes ('1 swap', '2 swap', '3 swap', and '1 shift') are outperformed by the neighbourhood structures that have larger structural changes ('mirror', 'many shift', and 'mirror shift'). The difference is approximately 8%. Both groups seem to have converged independently to different values. However, for larger N, we expect all neighbourhood structures to converge to the same value. The point of convergence has shifted to higher N compared to res(7,6), and HEP(σ), since the state space has increased in size from 15! to 24!. From this plot, we may conclude that the state space of HEP(F_{13}) is more suited to be traversed with larger changes.

In figure 12b the performance of the neighbourhood structures for HEP(b) is shown. We see that two consecutive swaps perform best at low N and that '1 swap', '2 swap', '3 swap', and '1 shift' converge at $N = 1000$. The neighbourhood structures that involve larger structural changes ('mirror', 'many shift', 'mirror shift') perform worse. These results are different from those of HEP(F_{13}): for HEP(b), with an even larger state space than HEP(F_{13}), smaller moves are better suited. This means that the mere number of variables is not a good indicator for the selection of a neighbourhood scheme.



(a) HEP(F_{13}) (b) HEP(b)

Figure 12: The expected number of operations of the minimum of four SHC runs with N iterations for HEP(F_{13}) (left), and HEP(b) (right). For HEP(F_{13}), mirror, many shift, and mirror shift converge to a lower value than the neighbourhood structures 1 swap, 2 swap, 3 swap, and 1 shift. For HEP(b), the 2 swap performs best for low N. 1 swap, 2 swap, 3 swap, and 1 shift converge at $N = 1000$. The other neighbourhood structures perform worse. Thus, the characteristics of HEP(F_{13}) and HEP(b) are different.

(D) Combined results

For the four benchmark expressions described in (B) and (C), and for the other three benchmark expressions, we observe that the relative improvement of the choice of the best neighbourhood structure compared to the worst neighbourhood structure is never more than 10%. Furthermore, we observe that there are two groups of neighbourhood structures when the state space is sufficiently large. Group 1 makes small changes to the state ('1 swap', '2 swap', '3 swap', '1 shift'). Group 2 makes large structural changes ('mirror', 'many shift' 'mirror shift'). The two groups converge before $N = 1000$ for expressions with small state spaces, such as HEP(σ), but are further apart for expressions with more variables, such as HEP(F_{13}) and HEP(b). The difference in quality in the group itself is often negligible (less than 3%). Thus, as a strategy to apply the appropriate neighbourhood structure, we suggest to distribute the number of parallel runs evenly among the two groups: in the case of four parallel runs, two of the runs can be performed using one neighbourhood structure from the group 1 and two from group 2.

2.4.3 *Two state space properties*

The fact that SHC works so well is surprising. Two well-known obstacles are (1) a Stochastic Hill Climbing can get stuck in local minima which yields inferior results [72], and (2) the Horner scheme problem does not have local heuristics, so

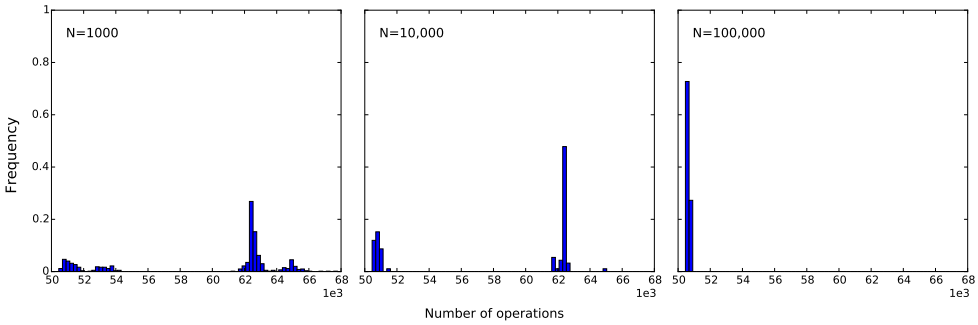


Figure 13: The distribution of the number of operations for the HEP(F_{13}) expression with 1 swaps for 1000 iterations (left), 10 000 iterations (middle) and 100 000 iterations (right) at $T = 0$. There appears to be a local minimum around 62 000. However, as the number of iterations is increased, the local minimum becomes smaller relative to the global minimum region (middle) and completely disappears (right). We may conclude that the apparent local minimum is not a local minimum, but a saddle point, since SHC is able to escape from the 'minimum'.

there is no guidance for any best-first search. Remarkably, SHC only needs 1000 iterations for a 107 variable expression (HEP(b)) to obtain good results, whereas a TSP benchmark problem with a comparable state space size, viz. kroA100 [79] with 100 variables, takes more than a million iterations to converge using a manually tuned SA search.

A thousand iterations is also a small number compared to the size of the state space. The average distance between two arbitrary states is 98 swaps. A thousand iteration SHC search accepts approximately 300 suggested swaps, so at least 33% of all the accepted moves should move towards the global minimum. This scenario would be unlikely if the state space is unsuited for SHC, so perhaps the state space has convenient properties for our purposes. We discuss the following two properties in the subsections below: (A) local minima, and (B) the region of the global minimum.

(A) Local minima and saddle points

To obtain an idea on the number of local minima, we measure how often the simulation gets stuck: if there are many local minima, we expect the simulation to get stuck often. In figure 13, we show the distribution of HEP(F_{13}) for 1000, 10 000, and 100 000 SHC runs respectively. For 1000 and 10 000 runs we see two peaks: one at the global minimum near 51 000 and one at an apparent local minimum near 62 000. As the number of iterations is increased, the weight shifts from the apparent local minimum to the global minimum: at 1000 iterations, there is a probability of 27.5% of arriving in the region of the global minimum, whereas this is 36.3% at

10 000 iterations. Apparently the local minimum is 'leaking': given sufficient time, the search is able to escape. The figure on the right with 100 000 iterations confirms the escaping possibility: the apparent local minimum has completely disappeared. Thus, the local minimum is in reality a saddle point, since for a true local minimum there is no path with a lower score leading away from the minimum. Since SHC requires many iterations to escape from the saddle point, only a few transitions reduce the number of operations.

We observe that apparent local minima disappear for our other benchmark expressions as well. SHC runs with 100 000 iterations approach the global minimum for all of our benchmark expressions. For example, for HEP(F_{13}) mentioned above, the result is 50636 ± 57 and for HEP(σ) the result is 4078 ± 9. The small standard deviations indicate that no runs get stuck in local minima (at least not in local minima significantly higher than the standard deviation).

From these results we may conclude that true local minima, from which a Stochastic Hill Climbing cannot escape, are rare for Horner schemes.

(B) Flatness of the state space

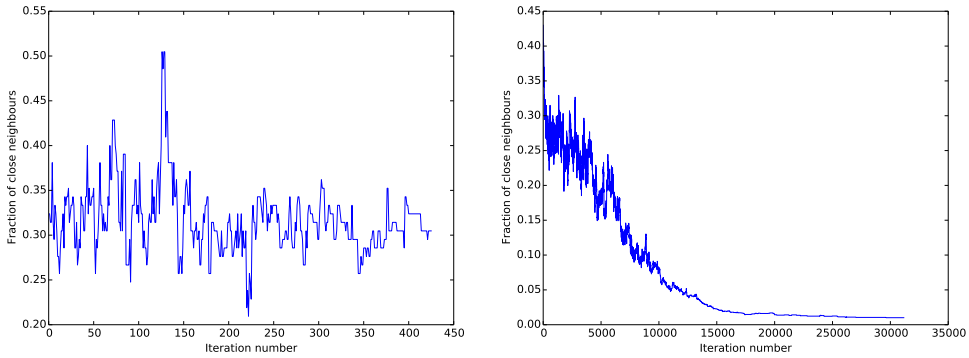
To build an intuition for what the state space looks like, we consider its flatness. We measure how many of the neighbours have a value (number of operations) that does not differ by more than 1%: $\frac{|x_n - x|}{|x|} < 1\%$, where x is the reference state and x_n is a neighbour of x. For brevity, we shall refer to this as 'close'.

In figure 14a we show the results for HEP(σ) for the current states during a typical single SHC run. We see that throughout the simulation the percentage of close neighbours is approximately 30%. We compare these results to an SA run of the TSP problem kroA100 (displayed in figure 14b). We see that for a random starting state the number of close neighbours is 30% as well, but as the simulation approaches the global minimum (at the right of the graph), the number of close neighbours decreases to 0.9%. As a result, the global minimum for TSP must be very narrow.

These results are a first hint that the state space of Horner is flat and terrace-like, whereas the TSP problem is more trough-like, with steep global/local minima. To investigate the flatness more deeply, we have looked at the distribution of the relative difference $\frac{|x_n - x|}{|x|}$. For the global minimum of the HEP(b) expression, this is depicted in figure 15. We see that about 75% of the neighbours are within 1% and 95% within 5%, which is even higher than for HEP(σ). We observed similar features for the other points in the state space, including hard to escape saddle points.

The property that the state space is flat is not only present in physics expressions, but is found in our other four benchmark expressions as well. Additionally, we have generated test expressions that we know to have interesting mathematical structures, such as powers of expressions. For example, for the expression $(4a + 9b + 12c^2 + 2d + 4e^3 - 2f + 8g^2 - 10h + i - j + 2k^2 - 3j^4 + l - 15m^2)^6$, 43% of the neighbours, 18% of the second neighbours and 7.3% of the third neighbours are close.

The question arises why the number of close neighbours is so high for the HEP(b) expression. For most expressions it is around 30%, but for HEP(b) it is 75%. A closer



(a) HEP(σ) with SHC

(b) kroA100 with SA

Figure 14: The typical number of close neighbours (value difference within 1%) for the current state for HEP(σ) and SHC on the left, and for SA on the the TSP benchmark problem kroA100 with 100 cities on the right. HEP(σ) does not show a decrease in the number of close neighbours, but remains steady around 30%. This is an indication that the state space is flat. For kroA100, the early states have many close neighbours, but as the simulation is converging to a minimum, the number of close neighbours decreases to about 0.9%. This is an indication that the state space has steep (local) minima.

inspection revealed that the HEP(b) expression has the special property that 90 of the 107 variables never appear in the same term: a term that contains variable x does not contain variable y and vice versa. As a result, the partial Horner schemes x, y and y, x yield the same expression. The HEP(b) expression is not the only expression with a high number of close neighbours: it represents a class of problems that often appears in electron-positron scattering processes.

The fact that some variables do not appear together in the same term is caused by a symmetry of the expression, since rearranging these variables in the scheme does nothing if they are direct neighbours in the scheme. The more symmetrical the expression is, the more likely it is that neighbours have the exact same value or a close value (within 5%). In the case of a uniformly random expression where the number of terms is much greater than the number of variables, we expect that practically all swaps are ineffective. The reason is that there is a high probability that each variable appears in an equal number of terms and has equal mixing.

Many, if not all, large expressions exhibit the 'flatness' property of their state space, since in most cases the number of terms is much larger than the number of variables. For the expressions that we have tested, the ratio of the number of terms and the number of variables is always more than a factor 1000. As a consequence, most

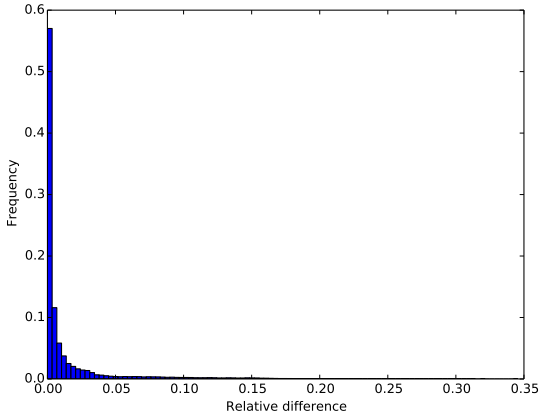


Figure 15: Relative difference of the values of swap neighbours of global minima for the HEP(b) expression, sampled over 37 states. The mean is 0.015 ± 0.034. The region is very flat: we observe that 85% of the neighbours have a value that is within 1% of the current value.

variables will appear in many terms, which in turn increases uniformity, resulting in neighbouring states with small differences in value (less than 5%).

2.5 PERFORMANCE OF SHC VS. MCTS

Below, we compare the results of Stochastic Hill Climbing to the previous best results from MCTS, for our eight benchmark expressions res(7,4), res(7,5), res(7,6), res(9,8), HEP(σ), HEP(F_{13}), HEP(F_{24}) and HEP(b). The results of all the MCTS runs except for res(9,8), and HEP(b) are taken from [53].[3] The results are displayed in table 2.

The results for MCTS with 1000 and 10 000 iterations are obtained after considerable tuning of C_p and after selecting whether the scheme should be constructed forward or in reverse (i.e., the scheme is applied backwards [14]).

For smaller problems, we observe that the averages of SHC are on a par with MCTS. However, we see that the standard deviations of SHC are higher than MCTS. This is because for MCTS the first nodes are fixed rather fast, which limits the variety. Consequently, we expect SHC to outperform MCTS if several runs are performed in parallel. Indeed, this is what we see in the last column of table 2. The standard deviations of MCTS are often an order of magnitude smaller than those of SHC, so the benefits of running MCTS in parallel are smaller. We may conclude that SHC has a better minimal behaviour if run in parallel.

3 We only consider optimisations by Horner schemes and CSEE. Additional optimisations that are mentioned in [53], such as 'greedy' optimisations, can just as well be applied to the results of SHC.

	vars	original	MCTS 1k	MCTS 10k	SHC 1k	SHC 10k	$E_{min,4}$ 1k
res(7,4)	13	29 163	$(3.86 \pm 0.1) \cdot 10^3$	$(3.84 \pm 0.01) \cdot 10^3$	$(3.92 \pm 0.28) \cdot 10^3$	3834 ± 26	3819 ± 9
res(7,5)	14	142 711	$(1.39 \pm 0.01) \cdot 10^4$	13768 ± 28	13841 ± 441	13767 ± 21	13770 ± 5
res(7,6)	15	587 880	$(4.58 \pm 0.05) \cdot 10^4$	$(4.54 \pm 0.01) \cdot 10^4$	46642 ± 3852	$(4.61 \pm 0.25) \cdot 10^4$	$(4.55 \pm 0.16) \cdot 10^4$
res(9,8)	19	83 778 591	$(5.27 \pm 0.25) \cdot 10^6$	$(4.33 \pm 0.31) \cdot 10^6$	$(4.13 \pm 0.34) \cdot 10^6$	$(4.03 \pm 0.17) \cdot 10^6$	$(3.97 \pm 0.18) \cdot 10^6$
HEP(σ)	15	47 424	4114 ± 14	4087 ± 5	4226 ± 257	4082 ± 58	4075 ± 25
HEP(F_{13})	24	1 068 153	$(6.6 \pm 0.2) \cdot 10^4$	$(6.47 \pm 0.08) \cdot 10^4$	$(5.99 \pm 0.51) \cdot 10^4$	$(5.80 \pm 0.55) \cdot 10^4$	$(5.37 \pm 0.40) \cdot 10^4$
HEP(F_{24})	31	7 722 027	$(3.80 \pm 0.06) \cdot 10^5$	$(3.19 \pm 0.04) \cdot 10^5$	$(3.16 \pm 0.23) \cdot 10^5$	$(3.06 \pm 0.23) \cdot 10^5$	$(2.98 \pm 0.09) \cdot 10^5$
HEP(b)	107	1 817 520	$(1.81 \pm 0.04) \cdot 10^5$	$(1.65 \pm 0.08) \cdot 10^5$	$(1.50 \pm 0.08) \cdot 10^5$	$(1.40 \pm 0.06) \cdot 10^5$	$(1.44 \pm 0.04) \cdot 10^5$

Table 2: SHC compared to MCTS. The MCTS results for all expressions except res(9,8) and HEP(b) are from [53]. All the values are statistical averages over at least 100 runs. SHC results have a larger standard deviation, and thus the expected value of the minimum is often lower than these values (see last column).

For our largest expressions, HEP(F_{13}), HEP(F_{24}) and HEP(b), we observe that SHC with 1000 iterations yields better results than MCTS with 10 000 iterations. For HEP(F_{24}), the average of SHC with 1000 iterations is about 20% better than the average for MCTS with 1000 iterations. In fact, the results are slightly better than MCTS with 10 000 iterations. If we take the $E_{min,4}$ into account, the expected value for HEP(F_{24}) is an additional 7% less.

The fact that SHC outperforms MCTS when the number of variables is larger than 23, may be due to the fact that there are not sufficient iterations for the branches to reach the bottom, making the choice of the last variables essentially random (see section 2.4 and [14]). This may also be the reason why for MCTS it is important whether the scheme is constructed forward or in reverse: if most of the performance can be gained by carefully selecting the last variables, building the scheme in reverse will yield better performance.

SHC is 10 times faster (in clock time) than MCTS, since most of the time is spent in the evaluation function. It is able to make reductions up to a factor of 26 for our largest expression.

2.6 CHAPTER CONCLUSION

Monte Carlo Tree Search (MCTS) with UCT has proven to be a good candidate to simplify large expression [53]. A downside of this method is that the constant C_p that governs exploration versus exploitation has to be tuned. The quality of the final scheme largely depends on this constant. We have modified the UCT algorithm so that C_p decreases linearly with the iteration number [14]. As a result, the final iterations are spent on *optimising* the end of the tree, instead of *exploring*. We show that using this modified UCT, the sensitivity to C_p is decreased by at least a factor 10 [13, 14]. Thus, the tuning is simplified.

Tree search methods, even with SA-UCT, have the problem that the beginning of the tree is optimised more than the end. For Horner schemes this does not lead to optimal solutions. Therefore we considered other algorithms that optimise uniformly.

Since sampling is slow for our use case, tuning many parameters is infeasible. For this reason, we preferred straightforward algorithms over sophisticated ones. In the case of Horner schemes, we have found that one of the most basic algorithms, Stochastic Hill Climbing, yields the best results.

2.6.1 *Findings and conclusions*

Stochastic Hill Climbing provides a search method with two parameters: (1) the number of iterations (computation time) and (2) the neighbourhood structure, which is a tunable parameter. We found that running half of the simulations with a neighbourhood structure that makes minor changes to the state (i.e., a single shift of a variable), and running the other half with a neighbourhood structure that involves larger changes (i.e., the mirroring of a random sublist) is a good strategy for all of our benchmark expressions (see subsection 2.4.2). Consequently, only the computation time remains as an actual parameter. From our experimental results we arrive at three subconclusions: (1) SHC obtains similar results to MCTS for expressions with around 15 variables, (2) SHC outperforms MCTS for expressions with 24 or more variables, and (3) SHC requires ten times fewer samples than MCTS to obtain similar results. Therefore we may conclude that SHC is more than 10 times faster [11].

The result that a basic algorithm such as SHC performs well is surprising, since Horner schemes have at least two properties that make the search hard: (1) there are no known local heuristics, and (2) evaluations could take several seconds. In the previous sections we have shown that the performance of SHC is so good, because the state space of Horner schemes is flat and has few local minima.

The number of operations is linearly related to the time it takes to perform numerical evaluations. The difference between the number of operations for the unoptimised and the optimised expression is more than a factor 24 compared. As a consequence, we are able to perform numerical integration (via repeated numerical evaluations) at least 24 times faster.

For High Energy Physics, the contribution is immediate: numerical integration of processes that are currently experimentally verified at CERN can be done significantly faster.

2.6.2 *Future research*

We see two promising options for future research. First, our algorithms assume that the expressions are commutative, but our implementation could be expanded to be applied to generic expressions with non-commuting variables. Especially in physics, where tensors are common objects, this is useful. Horner's rule can only be applied uniquely to commutative variables, but the pulling outside brackets keeps the order of the non-commuting objects intact. Thus, for Horner's rule the only required change is the selection of commutative variables for the scheme. The common subexpression elimination should honour the ordering of the non-commutative

objects. For example, in figure 2, the two highlighted parts are not a common subexpression if the variables are non-commutative ($a + e$ cannot be moved to the left of c). To enable non-commutative objects, CSEE should only compare connected subsets.

Second, additional work can be put in finding other methods of reductions. For example, expressing certain variables as linear combinations of other variables may reduce the number of operations even further. Many of these patterns cannot be recognised by common subexpression elimination alone. Determining which variables should be expressed as linear combinations of other variables to yield optimal results is an open problem. Perhaps Stochastic Local Search techniques are applicable to this subject as well.

Our algorithms are implemented in the next release of the open source symbolic manipulation system FORM [80] and are used by multiple research groups.