Convergence of distributed symbolic regression using metaheuristics

Ben Cardoen
ben.cardoen@student.uantwerpen.be

May 21, 2017

1 Abstract

Symbolic regression (SR) fits a symbolic expression to a set of expected values. Amongst its advantages over other techniques is the ability for a practitioner to interpret the resulting expression, determine important features by their usage in the expression, and insights into the behavior of the resulting model such as continuity, derivatives, extrema. SR combines a discrete combinatoric problem, combining base functions, with the continuous optimization problem of selecting and mutating real valued constants. One of the main algorithms used in SR is Genetic Programming (GP). The convergence characteristics of SR using GP are still an open issue. The continuous aspect of the problem has traditionally been an issue in GP based symbolic regression. This paper will study convergence of a GP-SR implementation on selected use cases known for bad convergence. We introduce modifications to the classical mutation and crossover operators and observe their effects on convergence. The constant optimization problem is studied using a two phase approach. We apply a variation on constant folding in the GP algorithm and evaluate its effects. The hybridization of GP with 3 metaheuristics (Differential Evolution, Artificial Bee Colony, Particle Swarm Optimization) are evaluated. In this work we will use a distributed GP-SR implementation to evaluate the effect of topologies on the convergence of the algorithm. We introduce and evaluate a topology with the aim of finding a new balance between diffusion and concentration. We introduce a variation of k-fold cross validation to estimate how accurate a generated solution is in predicting unknown datapoints. This validation technique is implemented in parallel in the algorithm combining both the advantages of the cross validation with the increase in covered search space for each instance. We combine our incremental support with a design of experiment technique applied on a simulator.

2 Acknowledgement

I would like to thank Prof. Dr. Jan Broeckhove for his continued guidance and support. I appreciated the advice of Sean Stijven and Lander Willem. Discussing my thesis with Elise Kuylen, Tim Tuijn and Stijn Manhaeve helped me solve issues in my implementation and stay sane. On a personal level I would like to thank my family for supporting me during my degree and thesis. Finally, as no writer is without muse, Debbie Derantere continues to inspire my writing.

Contents

3 Introduction

3.1 Overview

In this work we will study the convergence behavior of Symbolic Regression. The convergence of symbolic regression is still an open problem [18, 16, 17, 21]. We will investigate the causes of this and suggest approaches. In particular we tackle the constant optimization problem. We evaluate our approach on a previously introduced [18] set of benchmark problems. We parallelize our approach and evaluate the effect on convergence. Finally we test our implementation on a real world use case and study its convergence. In the remainder of this section will give a problem statement, describe the challenges faced and why they are relevant.

3.2 Symbolic Regression

Symbolic regression generates an expression that, when evaluated on a set of input points X, approximates an expected output set. This is a minimization problem where we can use as fitness a distance function. The expression is comprised of base functions, constants and features.

3.2.1 Problem hardness

For any given input set and output set there are an infinite number of expressions that give the exact expected output. If we have a reasonably small set of base functions, a strict limit on the length of the expression, and a finite range for the constants, the number of expressions that matches the output is still unfeasibly large. The set of approximating expressions that come within a threshold distance is obviously even larger. In general we do not have a starting point for the optimization process, forcing us to use a random sample from the search space as initial values. Let B be the set of base functions with $|B| = b \wedge b < \infty$. The feature set F is user provided and will in practice rarely exceed 1e3: $|F| = f <= 1e3$. If we use $\mathbb{R}$ as the domain for constants we have an infinite search space. Infinite precision floating point numbers have a high performance cost. It is more reasonable to use the IEEE754 double precision floating point standard in order to avoid an infinite search space and to guarantee reasonable performance : $|C| = c \sim 2^{64}$.

Problem representation If we represent an expression as a tree we can describe the size of the search space. Suppose we allow expressions that are represented by trees with depth d. An internal node is a base function, a leaf is a feature or a constant. If we limit our base function set to binary and unary functions, a full tree of depth d will have on average N nodes:

$$N = \frac{2^{d+1} - 1 + d}{2} = O(2^d)$$

The number of leaves is:

$$L = N - \frac{2^d + 1}{2} = O(2^{d-1})$$

The number of internal nodes for a tree of depth d is given by:

$$I = N - L = O(2^{d-1})$$

Size of the search space For a full random tree of depth d we need to pick I functions with replacement from B. If we select for a leaf with probability $\frac{1}{2}$ a feature or a constant we need to select L/2 constants from C and L/2 from F, both with replacement. The total number of trees we can generate with B,C,F and d given is :

$$S = b^I c^{\frac{L}{2}} f^{\frac{L}{2}}$$

Clearly c dominates this expression. While the search space is not infinite, it might as well be given our calculation. For any given input and output set we do not know the 'correct' value of d. We do know the maximum number of features the expression can use, and from this could derive a heuristic to determine the expected value d. If we have k features and expect all of them to be significant, we would need at least 2k leaves, with half being features and half constants. This would give us a minimum depth of $\lceil \log_2 2k \rceil$. In practice the depth should be greater than this value. We cannot rule out that features will need to be reused by an optimal expression. If a tree is not full the number of leaves will be significantly smaller resulting in a higher minimum depth needed to use all features. We have to assume that all features are significant. This reflects once again the incomplete information we have about the search space. Evolutionary algorithms that evolve these expressions are likely to introduce bloat, subexpressions that do not contribute to the fitness value. We have to take this into account if we let f guide our value of d. Suppose we introduce the guide $d = 8f$ then we can approximate S as :

$$S = b^{2^{8f}} c^{2^{8f-2}} f^{2^{8f-2}}$$

With $c >> b \wedge c >> f$

$$S = O(c^{2^f})$$

The size of the search space and lack of information means we can only approximate this problem with a metaheuristic.

Solution We do not know what our solution is. Metaheuristics are typically applied to problem statements where the solution is best described as "I know it when I see it". Only when discovered can we evaluate the worth of a solution. In symbolic regression we could describe our desired solution by requiring that it has a distance 0. This is in and of itself not enough to use in real world applications. We know that the solution will not be unique, so given 2 solutions with an equal distance which do we prefer? The issue of bloat reappears here. Do we prefer simpler expressions above more complex? If h and g both have equal distance but h is continuous and has clearly defined extrema whereas g is discontinuous, is h then a 'better' solution? What happens if we use h and g on unknown data? Do we want only minimum distance or also would like maximum predictive capability? These questions introduce multiobjective fitness functions which use the distance function as only a part of the heuristic driving the algorithm. The diversity between solutions is another aspect that is sometimes a goal in these problem

statements. We would like solutions that contain as much unique information as is possible.

3.2.2 Compared to other techniques

When a metaheuristic is applied to a problem and returns a solution a practitioner would like to be able to validate the solution. In general we do not want black box behavior, where the algorithm returns a solution and we do not know how this solution was obtained. A symbolic expression offers a more transparent model to a practitioner than for instance a neural network or a trained classifier. The use of mathematical functions allows insight into the behavior of the model which other approach cannot. From the expression we can deduce how features influence each other and which is more significant. Other insights such as continuity and derivation are more difficult to interpret. We cannot assume that the model we wish to approximate with symbolic regression is continuous simply because our best approximation is continuous.

3.2.3 Applications

There are a wide variety of applications for symbolic regression. Simulation is one of the main applications. Simulation of non trivial processes is requires a vast amount of computational resources. To complicate matters, simulators have often a large parameter space. We can use symbolic regression to gain insight into the correlation between parameters. The resulting expression is a model that approximates the simulator. If we have an accurate model with good predictive capabilities we can save simulations and use the expression instead. The symbolic expression allows us to gain insights into the underlying model. We can use mathematical analysis to distinguish significant parameters and correlated parameters. We will apply SR to a simulation use case to demonstrate the advantages and disadvantages of this application.

3.3 Convergence

3.3.1 Measures

Quality We define the quality of a solution as its fitness value. If we use only the distance function as the fitness value, we can ask the SR tool to return solutions that are at least as fit as a given threshold. This threshold value is hard to set as each problem statement will have different convergence characteristics. In order to use this approach the fitness function should have a finite range with a defined scale.

Convergence rate A practitioner would like to know how long it would take for an SR algorithm to find solutions of a given quality. It is equally important to know how convergence behaves over time. Suppose we have a fitness function with range [0,1] with 0 optimal. The SR tool evolves g generations and the tool and finds a solution with fitness 1e-12. A practitioner would like to know how the increase in generations would reflect on the decrease in fitness. We would like to be able to answer the question : "How many generations are needed to improve fitness by an order of magnitude?".

Current implementations in SR are unable to answer these questions. It is possible to track convergence over time and based on this extrapolate future behavior. CSRM offers the user such insights by visualizing the convergence rate amongst other statistics.

Accuracy If we know that there exists a single global optimal solution, we can define accuracy as a distance measure. We score each solution the algorithm evolves based on the distance to the known optimal solution. Accuracy measures the algorithm's capability to approximate a known solution.

3.4 Metaheuristics

The symbolic regression problem has an intractable search space with incomplete information. Such a problem statement can be approximated with metaheuristics. In this work we focus on population based algorithms where a set of solutions is updated and the optimization process is accelerated using sharing of information between the solutions.

3.4.1 Exploration versus exploitation

All metaheuristics need to find a balance between exploration and exploitation. Exploration is needed in order to have a reasonable probability of finding a neighbourhood in the search space where the global optimum can be found. Exploitation is then needed to find that global optimum, given such a neighbourhood. Exploration has to be limited, by virtue of the problem statement full coverage of the search space is infeasible. Given an intractable or infinite search space exploration should be a sampling process capable of finding regions in the search space that contain optima. Exploration prevents premature convergence, allowing the optimization process to escape from local optima. Finally it promotes diversity. While this is a necessity if we have a multimodal problem statement, diversity is always beneficial for an optimization process. The semantic similarity or difference between solutions with equivalent fitness values offers valuable insights into the underlying problem. Exploitation will increase the quality of solutions at a risk of overfitting and premature convergence.

3.4.2 Analogy with nature

A large subset of metaheuristics is nature inspired, but care should be taken when using this analogy. Each year new algorithms are introduced based on some observed optimization process in nature, typically a population of social creatures foraging for food. A second analogy is made with evolution itself. Solutions are constructed as genetic material, assigned a fitness function and evolved using several evolution strategies influenced by nature. These approaches have been successfully applied to hard problems, but we have to make an informed application of them.

Fitness A first concern is the fitness function. The topology of the fitness function determines to a large extent the convergence of any optimization algorithm. The optimization algorithm's capability to successfully deal with a fitness topology that con-

tains local optima which are hard to escape from is its robustness. A robust optimization algorithm can be applied to a wide set of problems with a low risk of issues such a premature convergence. The problems solved in nature by evolution and cooperation should have a fitness toplogy that at least can be translated to the problem we are trying to solve. A second observation to make regarding fitness is that in nature fitness in and of itself is not the goal of the process, survival is. A genome, animal or insect only has to have sufficient fitness in order to survive. In contrast, in computer science, we are interested in the best, most fit, solution. Both approaches are similar in goal but differ enough to lead to interesting side effects when applying nature inspired algorithms to optimization problems. Genetic algorithms have a tendency to evolve introns, exactly like their counterparts in nature. Introns are large sections of a genome that do not contribute to the fitness. Their role is not completely accidental, however, introns can function as genetic memory and can even help in crossing zero gradient areas in the fitness function topology. In optimization algorithms introns cause serious issues. While their beneficial effects can be present here as well, in general introns, or bloat, will only degrade the performance of the algorithm.

Swarm Intelligence A second concern is the lack of novelty when introducing new algorithms based on observations from nature. Most swarm intelligence algorithms are derived from observed behavior in nature where swarms of insects or animals forage for food or defend themselves. In computer science such algorithms always feature the same aspects : population management, improving local solutions, discovering new solutions and communicating between the particles in the swarm. Theoretical results for most of these algorithms are lacking or incomplete, yet vital. If we have a theoretical result that guarantees convergence with a resource limit, we have a robust algorithm. This is especially important given that we are trying to solve problems with an intractable search space. The growth in algorithms, and the fact that most of these algorithms have a large set of parameters that impact their convergence makes it difficult for theoretical analysis to keep pace.

3.4.3 Optimal algorithm

The No Free Lunch theorem for metaheuristics [27] states that no single metaheuristic is optimal for all given problem instances. This result can be used as a quality measure for publications in the field of metaheuristics. Some application domains lack benchmark problems that the community can reuse in order to measure new or modified algorithms. We can measure the quality of a good set of benchmark problems if no algorithm can be found to be optimal for all problems. This follows from the NFL theorem. A new algorithm that would demonstrate such convergence characteristics would indicate that the problem set is not general enough.

Hyperheuristics Hyperheuristics may offer a solution to this issue. A hyperheuristic generates an optimal metaheuristic for a given problem instance, and thus can, in theory, offer optimal solutions for all problem instances [23]. An intermediary approach between optimizing a problem with a metaheuristic, and generating a metaheuristic to

optimize a problem, is to optimize an existing metaheuristic. For this we do not need a hyperheuristic, a simple discrete or continuous optimizer suffices. Its input will be the parameter set of the existing metaheuristic we wish to apply to our problem. A variant of this approach is a self optimizing optimizer where the metaheuristic has parameters that are self-adapting based on feedback during the optimization process. As an example, Particle Swarm Optimization (PSO) [13] has self adapting variants that have been shown to improve convergence [6]. This is an attractive approach as it can be applied during the optimization process and does not require a hyperheuristic. If we are solving a large set of similar optimization problems then it can be more beneficial to generate a metaheuristic using a hyperheuristic. If each problem we face is sufficiently diverse a self adapting metaheuristic is preferred.

3.4.4 Combinatorial versus continuous

The translation or mapping of the problem statement to a representation that can be approximated by a metaheuristic is non trivial. We differentiate between combinatorial and continuous optimization problems. SR is a combination of a combinatorial and continuous optimization problem. Some metaheuristics are more fit to solve a combinatorial problem, for example Ant Colony Optimization (ACO) applied to the travelling salesman problem (TSP) [5]. Others were constructed to focus on continuous problems such as PSO. This does not mean that the distinction is binding, both ACO and PSO have continuous and discrete applications. If our problem combines both a discrete and continuous optimization problems, then using a memetic or hybrid metaheuristic is a useful approach. We combine complementary algorithms in order for them to complement them. A typical example is combining an algorithm known for high exploitation as a local search agent with an algorithm good at exploration for global search [20]. In this work we combine an algorithm for the discrete subproblem with several algorithms that optimize the continuous subproblem.

3.4.5 Continuous optimizers

For the continuous part of the symbolic regression problem we focus on three metaheuristics. All are population based, and can be classified as swarm intelligence. Differential Evolution [26] works by combining vectors in a multidimensional search space. PSO is, like DE, a well established continuous metaheuristic. PSO focusses on the concept of particles with velocities and position, rather than abstract vectors. Unlike newer algorithms, both have a thorough theoretical analysis that can be used as a guide by a practitioner. Artificial Bee Colony [11] is one of the newer algorithms in continuous metaheuristics. The bee analogy is somewhat confusing, it helps to regard ABC simply as a population of particles split of 3 groups where each focusses on exploration or exploitation. All three are robust, performant algorithms with a distinct approach at solving continuous optimization problems. This selection serves as a representative sample for the set of swarm intelligence algorithms. All three have reasonably small parameter sets and published work that studies optimal values for their parameters.

3.4.6 Genetic Programming

Genetic programming [19] evolves not only instances or solutions, but is capable of generating programs. This makes it a hyperheuristic, it can generate a metaheuristic that can optimize a problem, or it can optimize it itself. We will cover GP in detail in section 4 when we discuss our implementation.

3.5 Constant optimization problem

From section 3.2.1 we know that selecting the 'right' constant value to use in the tree is very hard. The probability of selecting at random a constant even close to an optimal value is extremely small. We can mitigate this by constricting the range of constants. Constant values from a small range can be used by base functions to generate far greater values. We are using subexpressions to help generate the 'right' constant value. This is something an evolutionary algorithm will tend to do by itself. It will try to approximate a constant by combining a few constants with base functions. The problem with this approach is that it is time and space inefficient. The entire subtree can be replaced by a single constant, yet it requires a significant amount of generations to generate the subtree. One approach is to hand over the constant optimization problem to a continuous optimizer and leave the selection of the base functions to the combinatorial optimizer. In section 6 we will cover this problem in detail.

3.6 Parallel

The hardness of the SR problem makes parallelization highly desirable. Regardless of the metaheuristic we apply there are several stages where parallelization can be applied. Evaluating fitness functions will be the main cost of any metaheuristic, obtaining a speedup in this stage would have a great impact on the runtime of the algorithm. This type of parallellization would not alter the behavior of the algorithm, the result would be invariant with or without parallelization. What makes parallel metaheuristics even more interesting is that parallelization can offer improved convergence compared to sequential execution. We can view a set of instances of the algorithm again as a cooperating swarm. The parallel metaheuristic starts to function as a self optimizing metaheuristic. By communicating information about the search space each metaheuristic instance is covering the entire process can accelerate convergence. While extremely powerful, parallel metaheuristics themselves require a careful implementation and configuration in order to achieve this goal. From an implementation standpoint we have to consider overhead in messaging and synchronization in order to avoid a serialization effect. The optimization process formed by the parallel metaheuristic has its own parameters that will define its balance between exploration and exploitation. A parallel metaheuristic can also be used as a hyperheuristic, running several instances of a metaheuristic with different parameter values in parallel and evolving the best performing. In section 5 we will cover this topic in detail.

4 Design

In this section we will detail the design of our tool, the algorithm and its parameters.

4.1 Algorithm

4.1.1 Input and output

The algorithm accepts a matrix X = n x k of input data, and a vector Y = 1 x k of expected data. It will evolve expressions that result, when evaluated on X, in an 1 x k vector Y' that approximates Y. N is the number of features, or parameters, the expression can use. K is the number of datapoints. The algorithm makes no assumptions on the domain or range of the expressions or data set. While domain specific knowledge can be of great value, in real world situations such data is not always known. We aim to make a tool that operates under this uncertainty.

4.1.2 Control flow

4.1.3 Entities

We will describe briefly the important entities in our design. In Figure 2 the architecture of our tool is shown. Functionality such as IO, statistics and plotting is left out here for clarity. The interested reader is referred to the source code documentation.

Algorithm The algorithm hierarchy extends each instance with new behavior, only overriding those functions needed. Code reuse is maximized here by the usage of hook functions, which in the superclasses result in a noop operation. An example of this is the evolve function, by and large the most complex function of the algorithm. By using hooks such as *requireMutation()* this function can remain in the superclass, and the subclass need only implement the hook.

Tree The Tree and Node classes represent the main data structure of CSRM's population. The Tree is comprised of Node objects, each of which holds an optional constant weight. Leaves are either features (VariableNode) or constants. In 4.8 we go in depth into the design choices made for this representation. The Tree class holds a few utility functions, for example to create trees from an expression using a parser in the tool module. Functionality needed by the mutation and crossover operators is present here as well. Of note are both construction functions, which allow for efficient generation of random trees. In section 4.3.1 we will cover the problem these functions solve.

Optimizer CSRM provides an interface to continuous optimizers. In section 6 we will go deeper into the specific algorithms used. The population data structure is shared between the subclasses. The instances in the population can be subclassed if specific functionality is needed by an optimizer. This approach allows us to reuse a large part of the code between algorithms. For convenience a stub optimizer is added. The optimization algorithms share configuration parameters, and encapsulate those that are distinct to their specific nature.

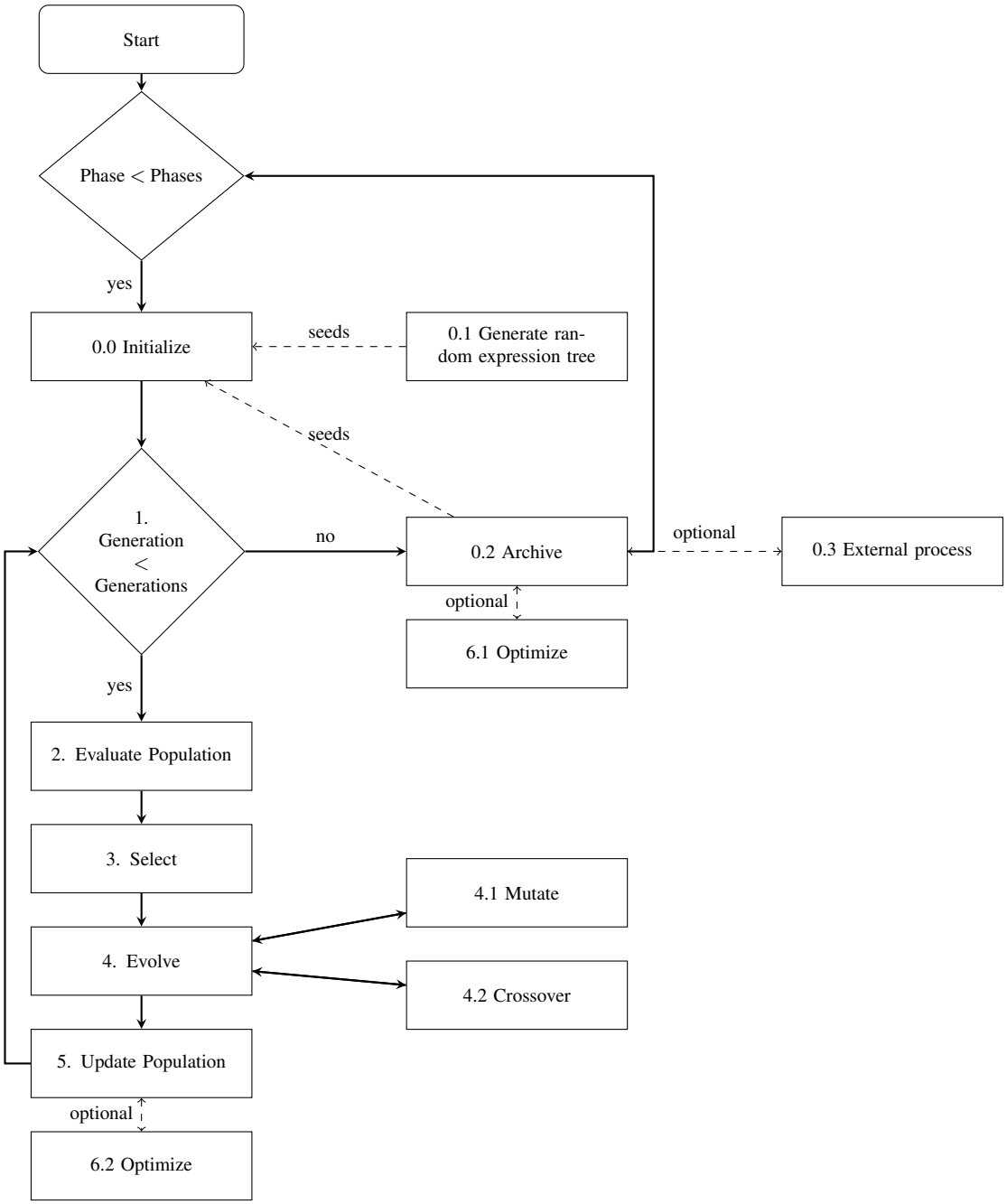


Figure 1: CSRM control flow.

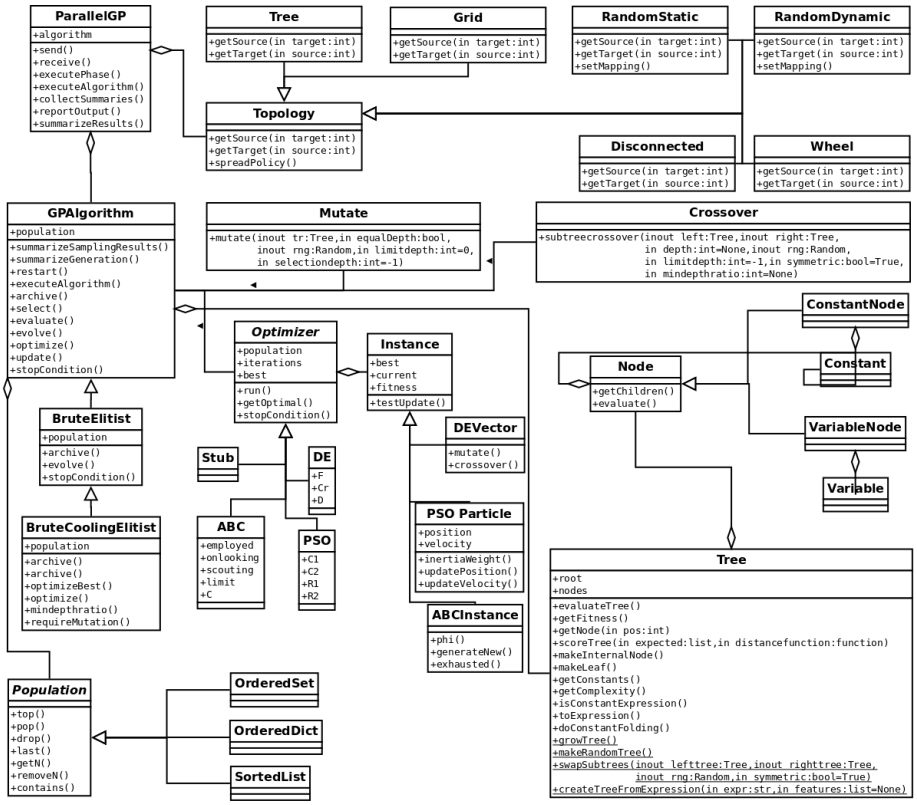


Figure 2: UML of CSRM.

4.1.4 Implementation

Our tool is implemented in Python. The language offers portability, access to rich libraries and fast development cycles. The disadvantages are speed and memory usage compared with compiled languages (e.g. C++) or newer scripting languages (e.g Julia). Furthermore, Python's usage of a global interpreter lock makes shared memory parallelism not feasible. Distributed programming is possible using MPI.

4.2 Fitness

In this work we interpret optimization as a fitness minimization process. In the following we will discuss the fitness function and its behavior.

4.2.1 Distance function

The goal of the algorithm is to find f' such that

$$Y = f(X)$$

$$Y' = f'(X)$$

$$dist(Y, Y') = e$$

results in e minimal. F is the process we wish to model or approximate with f'.
Not all distance functions are equally well suited for this purpose. A simple root mean squared error (RMSE) function has the issue of scale, the range of this function is [0, +∞), which makes comparison problematic, especially if we want to combine it with other objective functions. A simple linear weighted sum requires that all terms use the same scale. Normalization of RMSE is an option, however there is no single recommended approach to obtain this NRMSE.
In this work we use a distance function based on the Pearson Correlation Coefficient. Specifically, we define

$$dist_p(Y, Y') = 1 - |r|$$

with

$$r = \frac{\sum_{i=0}^{n} (y_i - E[Y]) * (y'_i - E[Y'])}{\sqrt{\sum_{j=0}^{n} (y_j - E[Y])^2 * \sum_{k=0}^{n} (y'_k - E[Y'])^2}}$$

R has a range of [-1, 1] where 1, -1 indicate linear and negative linear correlation and 0 no correlation. This function has a range [0,1] which facilitates comparison across domains and allows combining it with other objective functions. The function reflects the aim of the algorithm. We not only want to assign a good (i.e. minimal) fitness value to a model that has a minimal distance, we also want to consider linearity between Y an Y'.

4.2.2 Diversity

Diversity, the concept of maintaining semantically different specimens, is an important aspect in metaheuristics. Concepts such as niching and partitioning are often used to promote this behavior, amongst other reasons to prevent premature convergence or even to enable multimodality. Our tool uses a simple measure that mimics the more advanced techniques stated above. It should be clear that for any combination of input and output data, there are a huge set of expressions with an identical fitness value. Such expressions can lead to premature convergence where a subset of the fittest expressions all have the same fitness value without introducing new information. Those expressions approximate the same local optimum. Our tool will aim to prevent retaining expressions that have identical fitness values. If the fitness function has a zero gradient surface allowing replacement of solutions based on equal fitness value can help the optimizer to traverse such an area.

4.2.3 Predictive behavior

The algorithm evaluates expressions based on training data X_t. X_t is an n x rk matrix based on the original input matrix X (n x k). R is the sampling ratio, the ratio between training and test data. After completion of the algorithm the population is ordered based on minimized fitness values calculated on the training data. In real world applications practitioners are also interested in the predictive qualities of the generates expression. In other words, how well do the expressions score on unknown data. In the final evaluation we score each expression on the full data to obtain this measure. While this gives us some information on how good an expression is on the full data set, we are also interested in how the convergence to this value progresses. If we add 10 more generations, or increase the population by a factor 1.5, what do we gain or lose in predictive quality of the expressions? To define this we use a correlation measure between the fitness values using the training data and those of the full data. This measure quantifies the predictive value of the final results. Finally, we calculate a correlation trend between the training fitness values at the end of each phase, and the final fitness values calculated on the full data. This trend describes the convergence process of the algorithm over time, specifically directed at the predictive value of the solutions found. It is important to note that the entire final population is considered in these calculations, not only the best. While ideal, there is no guarantee that the expression that has the lowest fitness value on the training data will score best on the full data set. We would like these measures to give us a descriptive capability for the entire process.

4.2.4 Convergence limit

As a stopcondition our tool uses a preset number of iterations. The user specifies the number of generations (g) and phases (r), and the algorithm will at most execute g x r iterations. Convergence stalling is implemented by keeping track of the number of successful operations (mutation or crossover). If this value crosses a threshold value convergence has stalled and the algorithm is terminated.

4.3 Initialization

Initialization is done using the 'full' method [19]. The algorithm has a parameter initial depth, each new expression in the population is created using that depth. Unless the maximum depth is equal to the initial depth, the algorithm will quickly vary in depth, evolving an optimal depth.

4.3.1 Invalid expressions

Generating a random expression is done by generating random subtrees and merging them. An important observation here is that randomly generated expressions can be invalid for their domain. The ubiquitous example here is division by zero. Several approaches to solve this problem exist. One can define 'safe' variants of the functions, in case of division by zero, returning a predefined value that will still result in a valid tree. The downside to this approach is that the division function's semantics is altered, a practitioner, given a resulting expression, would have to know that some functions are no longer corresponding entirely to their mathematical equivalents, and what 'safe' values the implementation uses. The other option is assigning maximum fitness to an invalid expression. While simple, this approach needs a careful implementation. From a practical standpoint wrapping functions in exception handling code will quickly deteriorate performance. This last problem is solved by a quick domain check for each calculation, avoiding exceptions.

Invalidity probability We define the probability that a randomly generated tree of depth d, with n possible features, k possible base functions, and j data points as q. With more complex problems d will have to be increased. GP is also susceptible to bloat [9], increasing d even further. This issue will affect generation of subtrees in mutation. With more datapoints the probability of at least one leading to an invalid evaluation will increase. An increase in d will lead to an exponential increase in the number of nodes in the tree. A node can be either a basefunction or a leaf (parameter or constant). For each additional node the probability q increases. We can conclude that q, while irrelevant for small problems and depths, becomes a major constraint for larger problem statements.

Bottom up versus top down There are two methods to generate a tree based expression : bottom up and top down. The top down approach is conceptually simpler, we select a random node and add randomly selected child nodes until the depth constraint in satisfied. The problem with this approach is that the expression can only be evaluated at completion, early detection of an invalid subtree is impossible. In contrast in a bottom up construction we generate small subtrees, evaluate them and if and only if valid merge them into a larger tree. This allows for early detection of invalid expressions. A downside of this approach is the repeated evaluation of the subtrees, which can be mitigated by caching the result of the subtree.

Disallowing invalid expressions in initialization We can generate random expressions and let the evolution stage of the algorithm filter them out based on their fitness

value, or we can enforce the constraint that no invalid expressions are introduced. The last option is computationally more expensive at first sight, since the algorithm is capable by definition of eliminating unfit expressions from the population. This can lead to unwanted behavior in the algorithm itself. For high q values we can have a significant part of the population that is at any one time invalid. This can lead to premature convergence, similar to a scenario where the population is artificially small or dominated by a set of highly fit individuals. Another observation to make is that the algorithm will waste operations (mutation, crossover) on expressions that are improbable to contribute to a good solution. While more expensive computationally, we therefore prohibit generation of invalid expressions in the initialization.

4.4 Evaluation and cost

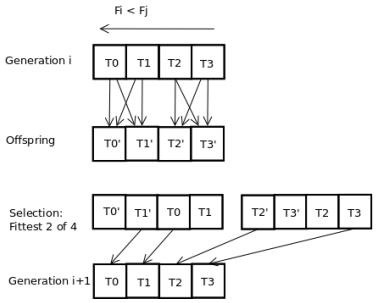
Evaluating a tree requires a traversal, once for each datapoint. In order to compare optimization algorithms across implementations practitioners can use as a measure the number of evaluations required to reach a certain fitness level. If this measure is used, one should take into account that not all evaluations are equal. With the trees varying in depth and density the evaluation cost varies significantly. Furthermore, evaluating "1 + 2" is computationally far less expensive than "log(3, 4)". A simple count of evaluation functions executed does not really reflect the true computation cost, especially when we consider the variable depth of the trees. Increasing the depth leads to an exponential increase in the number of nodes, and thus in the evaluation cost. Our tool uses a complexity measure that takes into account the density of the tree and which functions are used. A tree comprised of complex functions will score higher in cost than a corresponding tree using simple multiplications and additions. Although this is an option, we do not use this measure in the objective function. We would like to observe the effect of the cost, but not directly influence the algorithm. There is no direct link between more complex functions and an optimal solution. In certain domains the argument can be made that more complex functions are more likely to lead to overfitting, or more likely to lead to invalid trees due to smaller domain.

4.5 Evolution

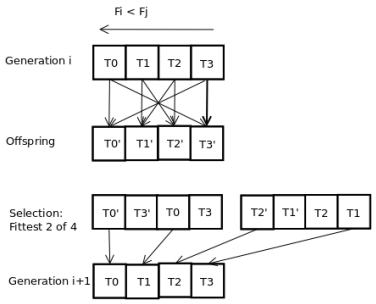
In the evolution stage we apply two operators on (a selection of) the population. The operators are configured to constrain the depth of the modified trees, enforcing the maximum depth parameter. We trace the application of the operators during the execution using an effectiveness measure. Each time an operator application is able to lower the fitness of a tree, this measure increases. Using this measure we can gain insight when and how certain optimizations and modifications work inside the algorithm instead of simply observing the algorithm as a black box.

4.5.1 Mutation

Mutation works by replacing a randomly chosen subexpression with a newly generated. In our implementation this means selecting a node in the tree and replacing it with a new subtree. Our mutation operator can be configured to replace the subtree

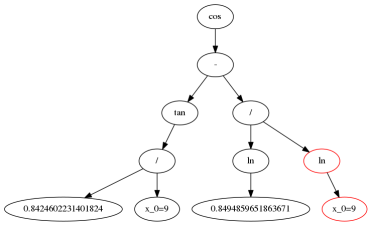


(a) Pairwise crossover.

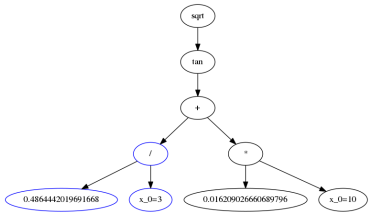


(b) Random crossover.

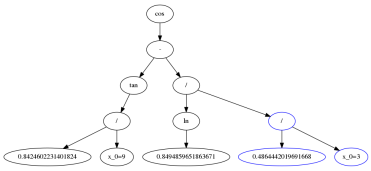
Figure 3: Selection procedures applied by crossover.



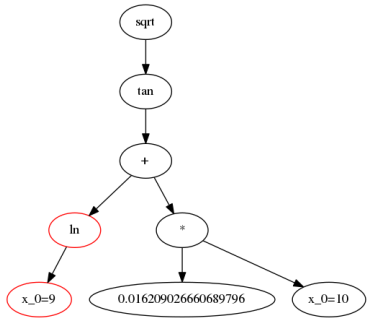
(a) A before crossover.



(b) B before crossover.



(c) A after crossover.



(d) B after crossover.

Figure 4: Symmetric crossover with two random trees.

with one of equal depth or a randomly chosen depth. The insertion point can be made depth-sensitive. A shallow node, a node with a low depth, is the root of a subtree with significant depth. Replacing such a subtree is therefore more likely to have a significant effect on the fitness value. Replacing a deep node will have on average a smaller effect. We will investigate this assumption in our experiments. In our implementation the mutation operator is applied using a cooling schedule, or on the entire population. The cooling schedule uses the current fitness and the current generation of a tree to decide if the tree is likely to benefit from mutation. This is similar to the approach in simulated annealing. Mutation introduces new information into the optimization process (a new subtree). This process can be constructive (lowering fitness) or destructive (increasing fitness). The idea behind the cooling schedule is that for fitter trees the mutation operation will not be able to improve fitness (decrease it), while for less fit trees it has a higher probability. This probability is estimated using the current generation and fitness value (relative to the population), and using this information a random choice is made whether or not to apply mutation. In the experiments section we will investigate if this assumption holds, and if it leads to gains in fitness and or effectiveness. The mutation operates has to generate new subtrees, and therefore the same issues seen in the initialization process which we discussed in section 4.3.1 apply here as well. The mutation operator will generate subtrees until a valid one is found. If the depth sensitive operation proves to generate equal or better fitness values, this could significantly reduce the computation cost of the operator.

4.5.2 Crossover

Crossover operates on 2 trees in the population. It selects subtrees from both, and swaps them between each other. The resulting set of 4 trees is scored and the 2 fittest replace the original trees. As with mutation crossover can be configured to operate on a set depth, or pick a random depth. It can also work symmetric, picking an equal depth insertion point in both trees. Crossover in our implementation can be applied in sequence to the entire population, with pairwise mating in order of fitness. Alternatively a random selection can be used. A variant of both, alternating between the two schedules based on a random choice is a third option. This is similar to the roulette wheel selection method, although without replacement. Crossover, unlike mutation, does not introduce new information in the sense that no new subtrees are generated. Crossover can be configured to work with a decreasing depth, operating only on deeper nodes at the later stages of the algorithm. The assumption for this mode of operation is similar to that made for mutation. In Figure 3 we see both selection procedures visualized. An important difference with tournament selection is that there is no replacement used in our selection. Crossover will be applied to an expression exactly once. Crossover is based on the idea that 2 expressions can improve by combining parts of themselves. The operator is selecting these subexpressions randomly. Suppose we have 2 trees, A and B, with randomly selected subtrees a and b respectively. We do not know how much of the fitness of A is the result of its subtree a, and vice versa for B and b. In addition, we do not know or even are able to estimate if a will improve B's fitness value if we replace b with it. In Figure 4 we see this visualized for two random trees. If the expression represented by a tree is a set of separable functions we could use crossover

and mutation only on those subexpressions. Unfortunately the search space is composed of both separable and non separable functions, and we do not know in advance if we gain from focussing only on separable solutions. Both crossover and mutation lack this information when dealing with subtrees and are forced to operate as random selectors.

4.6 Selection

After evolution a decision has to be made on which expressions will be retained in the population. In our tool the population is fixed, so a replacement is in order. We use an elitist approach. If an expression has a lower (better) fitness value after mutation, it is replaced. In crossover we combine two expressions r, and t, resulting in two offspring s, u. From these four expressions the two with minimal fitness survive to the next generation.

4.7 Archiving

The algorithm holds an archive that functions as memory for best solutions obtained from the best expressions at the end of a phase., from seeding, or from other processes. At the end of a phase we store the j best expressions out of the population. J is a parameter ranging from 1 to the population size n. With j == n we risk premature convergence, with j == 1 the risks exists that we lose good expressions from phases which will have to be rediscovered. While there are numerous archiving strategies described in literature, we use a simple elitist approach. This means that there is no guarantee that the best j samples of phase i are retained, if they have fitness values lower than those present in the archive and the archive has no more empty slots, they will be ignored. This leads us to the size of the archive. While no exact optimal value for this exists, in order to function as memory between phases it should be similar in size to the amount of phases. The j parameter will influence this choice as well.

4.8 Representation and data structures

4.8.1 Expression

We use a tree representation for an arithmetic expression, where each internal node represent a base function (unary or binary), and each leaf either a feature or a constant.

Tree representation We use a hybrid representation of a tree structure. The tree is a set of nodes, where each node holds a list of children nodes. This representation favors recursive traversal algorithms. In addition to this representation, the nodes of a tree are stored in a dictionary keyed on the position of a node. This allows for O(1) access needed by the mutation and crossover operators. The overhead of this extra representation is minimal, since the keys are integers and only a reference is stored. A list representation would be faster in (indexed) access, but would waste memory for sparse trees. With a mix of unary and binary operators the average nodecount of a tree with depth d is $\frac{2^{d+1}-1+d}{2}$ = O(2^d) resulting in savings on the order of 2^d, with

d the depth of the tree. The algorithm has the choice which mode of access to use depending on the usage pattern. As an example, selecting a random node in the tree is O(1), selecting a node with a given depth is equally O(1). Splicing subtrees is equally an O(1) operation.

Base functions The set of base functions is determined by the problem domain. For symbolic regression we use the following set:
+, -, %, /, max, min, abs, tanh, sin, cos, tan, log, exp, power
A problem with this set is the varying domain of each, why may or may not correspond with the domain of the features. A possible extension is to use orthogonal base functions such as Chebyshev polynomials. In our solution the functions listed correspond with their mathematical equivalent, e.g. division by zero is undefined. Other constraints are based on floating point representation (e.g overflow).

Constants Constants are fixed values in leaves in the tree. The representation also allows for multiplicative weights for each node, which can be used to optimize the final solution. In contrast to the base functions with a limited set to choose from, constants have the entire floating point range at their disposal. The probability of selecting a 'right' value is extremely small, and domain information is lacking for these constants, it depends on the entire subtree holding the constant. We select a constant from a small range and allow the algorithm to recombine these constants later to larger values. This limiting of the search space can lead to the algorithm converging faster to a suboptimal solution. The constants are reference objects in the tree, and so can be accessed and modified in place by an optimizer.

Features Features are represented as an object holding a reference to a set of input values, and always occur as leaves. The choice for a random leaf is evenly distributed between constants and features.

4.8.2 Population

The population of trees is kept in a sorted set, where the key is the fitness value and the value is a reference to the tree representing the expression. Sorted datastructures are notably lacking from Python, we use the sortedcontainers [10] module which provides sorted datastructures that are designed with performance in mind. This representation allows for fast access in selecting a (sub)set of fittest trees. It also allows for an O(1) check if we already have an equivalent solution, a tree with a fitness score which we already have in the population. This allows our diversity approach mentioned in 4.2.2. In Figure 2 we see that the actual data structure used as population is interchangeable. If duplicate fitness values are allowed, and membership testing is no longer needed, a sorted list could be used instead.

4.9 Parameters

In this section we briefly list the main parameters of the algorithm and their effects on convergence, runtime and complexity.

4.9.1 Depth

The depth of trees van vary between an initial and maximum value. If we know in advance an optimal solution uses 13 features, the tree should have at least 13 leaves in order to use each feature at least once. This requires a depth of at least 4. In practice the depth will need to be greater due to the use of unary functions, and bloat. Bloat is a known issue in GP [9] where the algorithm, unless constrained by limits or an objective function that penalizes depth, will tend to evolve deep trees without gaining much in fitness. In the worst case entire subtrees can be evolved that do not contribute to the fitness value, mimicking introns in genetics. These can still have a valid purpose, serving as genetic memory. Their disadvantage is also clear: a computational overhead without clear effect on the fitness value. A similar problem arises with the generation of constants. Suppose we would like to generate the constant 3.14, the probability of generating this by a single random call is extremely small. The algorithm will try to build an expression fitting 3.14, for example ”1+(3*1) + 28/200”. The problem with this is that this process is highly inefficient. It uses iterations that, if a more efficient approach exists, could be used to optimize the fitness value of the tree further. The constant expression wastes nodes that could be used to improve the tree. One approach is folding such expressions into a single constant, which mitigates some of the effects mentioned but does not prevent such subtrees from forming. Without a solution for this issue, the user would have to take this into account and increase the depth parameter. From 4.3.1 we know that the initialization process and the mutation operator will become more costly exponentially with the increase in depth. A similar argument can be made for the time and space complexity of operating on deeper trees, both increase exponentially.

4.9.2 Population size

The population size is directly related to convergence speed. The catch is the quality of the solution, a very small population will lead to premature convergence, the population is lacking in diversity (or information viewed from a different perspective). A large population on the other leads to a large increase in runtime, unless the operators only work on a subset of the population. The optimal value is problem specific, but values in the range of 20-50 give a good balance for our implementation.

4.9.3 Phases and generations

The execution of the algorithms comprises of g generations and r phases, resulting in a maximum of g x r iterations. For both parameters a too small value will hinder convergence, while a high value can lead to overfitting. The optimal value is domain specific and dependent on the iterations required to approximate the expected data. This is by virtue of the problem statement unknown. There is a subtle difference between these

parameters. Each phase a reseed of the algorithm is done using the archive. This archive holds the best results from previous phases, external seeds and in a distributed setting the best best solutions from other processes. The remainder of the population is initialized with random tree. These trees introduce new information into the optimization process, and while expensive and with a low probability of improving fitness, nonetheless will help avoid premature convergence. If the archive size is less than the population size, new trees will always be introduced. The rate at which the archive fills is dependent on the number of phases and a parameter which determines how many trees are archived. In a distributed setting this process accelerates using the input of other processes. If the archive is full after i phases, and the archive size is equal to the population, further phases will no longer have the benefit of newly generated trees. This does not prevent convergence, but could reduce the convergence rate in some scenarios. Increasing g and r both can lead to overfitting, but in order to decide on r we also have to take into account the archiving strategy. In order to find a good starting value for g one can look at the population size. Diffusion, where the information from each expression is shared with the others, will require at least the same amount of generations as there are expressions, depending on the operators and the individual fitness of the expressions. Concentration, where we only look at maximizing the few existing fittest expressions, requires far less generations, but risks premature convergence.

4.9.4 Samples

The user can provide the algorithm with input data, or can specify a range from which to sample the input data. In addition, the ratio between training and testing data can be adjusted. Care should be taken in tuning this value. A low ratio will increase the probability that evolved solutions are invalid on the test data, while a high ratio will lead to overfitting.

4.9.5 Domain

Domain specific knowledge can significantly reduce the time needed to converge to a solution. In our implementation we assume no domain knowledge. While this increases the complexity of obtaining a good solution, it also makes for a fairer evaluation of the algorithm and optimizations used.

4.10 Incremental support

Our tool supports incremental operation. The user can provide seeds, expressions that are known or assumed to be good initial approximations. The algorithm writes out the best solutions obtained in an identical format. By combining this the user can create a Design Of Experiment (DOE) setup using the algorithm. As a use case, suppose the user wants to apply symbolic regression on the output of an expensive simulation. The simulator has k features or parameters, each with a different domain and j datapoints. The user wants insights into the correlation of the parameters. A naive approach would be to generate output data for a full factorial experiment, and feed this into the CSRM tool. For both the simulator and the SR tool the cost of this approach would be

prohibitive. It is likely that some features are even irrelevant, leading to unnecessary computation. Instead we can opt for a combined DOE approach. We start with a subset of features k' $<$ k and datapoints j' $<$ k. The simulation results Q of this subset are then given to the CSRM algorithm. It generates a solution, optimized for this subset of the problem. The user then adds more parameters, k' $<$ k" $<$ k and/or datapoints j' $<$ j" $<$ j, runs the simulator again. The resulting output is given the CSRM tool, with Q as seed. This seed is used as memory of the previous run on the smaller input set. Unless there is no correlation between the incrementing datasets the CSRM tool can use the knowledge gained from the previous run to obtain faster convergence for this new dataset. In addition, by inspecting Q the user can already analyze a (partial) result regarding the initial parameter set. Suppose k = 5, and only 3 parameters are used in Q. Then the user can exclude the missing two parameters from the remainder of the experiment, as these are unlikely to contribute to the output. By chaining the simulator and CSRM tool together in such a way, an efficient DOE approach can potentially save both simulation and regression time, or result in increased quality of solutions. The advantages are clear :

- The SR tool can start in parallel with the simulator, instead of having to wait until the simulator has completed all configurations.
- Reducing irrelevant parameters detected by the SR tool can save in configuration size for the simulator.
- The SR tool can reuse previous results as seeds, and tackles and slowly increasing search space with those initial values instead of starting without knowledge in a far larger search space.

Possible disadvantages are :

- The SR tool is not guaranteed to return the optimal solution, it is possible the process is misguided by suboptimal solutions. This risk exist as well in the full approach.
- Interpretation of the results can be needed in order to make the decision to prune features.

We will investigate this approach in our use case in section 8. The following diagram illustrates a DOE hypercube design using our tool and a simulator.

4.11 Statistics and visualization

Stochastic algorithms are notoriously hard to debug. By virtue of the problem statement we do not know whether the returned solution is what is expected. The size of the search space makes detecting all edge cases infeasible. In addition the algorithm functions as a black box, where output is presented to the user without a clear trace indicating how or why this output was obtained. Both for developer and practitioner insight into the algorithms inner workings is vital. Our tool provides a wealth of

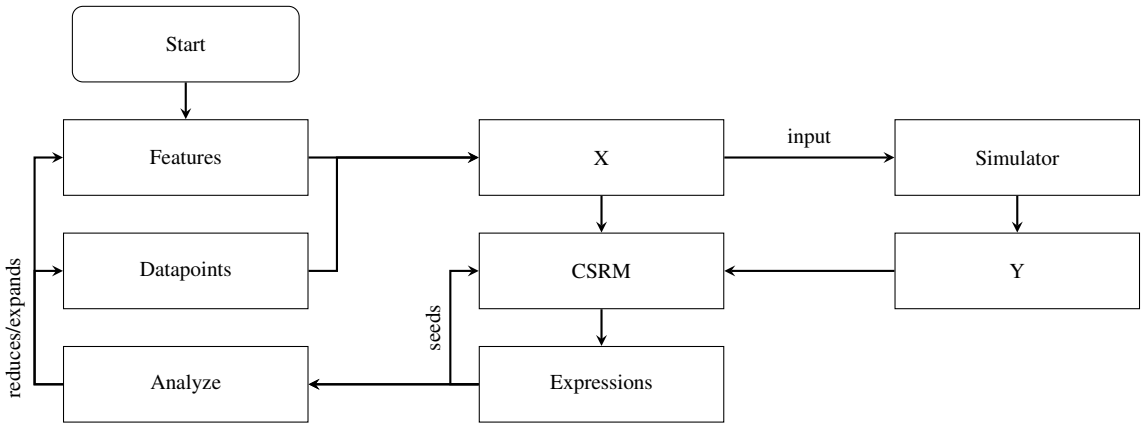


Figure 5: Incremental DOE using CSRM and a simulator.

runtime statistics ranging from fitness values per expression for each point in the execution, convergence behavior over time, depth and complexity of the expressions, cost of evaluations, effectiveness of operators, correlation between training and test fitness values and more. These statistics can be saved to disk for later analysis, or displayed in plots in a browser. Using this information the user can tune the parameters of the algorithm (e.g. overfitting due to an excessively large number of phases), the developer can look at how effective new modifications are (e.g. new mutation operator) and so on. It is even possible to trace the entire run of the algorithm step by step by saving the expressions in tree form, displayed in an SVG image rendered by Graphviz [8]. In Figure 6 a selection of the collected statistics on a testfunction is shown. The third of our set of testproblems was used with a depth $\in$ [4,10], 30 generations, populationsize 30, and 4 phases.

4.12 Conclusion

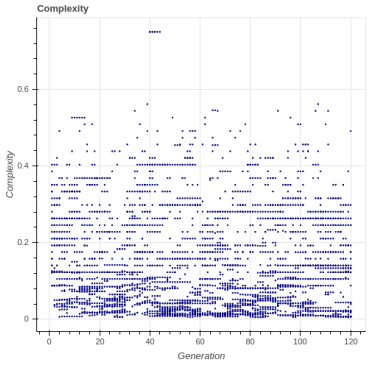
In this section we have covered in detail the design of the CSRM tool, highlighting the choices made.

5 Distributed SR

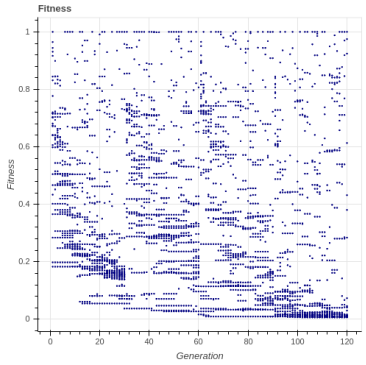
In this section we will cover the parallelization of symbolic regression and in this context detail the design choices we made in our approach.

5.1 Approaches

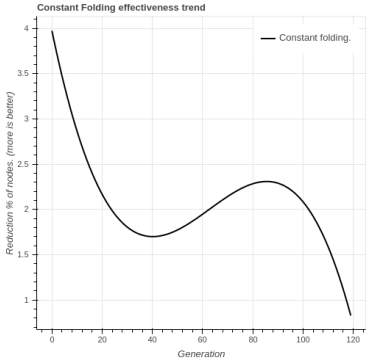
In this discussion we distinguish between fine grained parallelism and coarse grained parallelism.



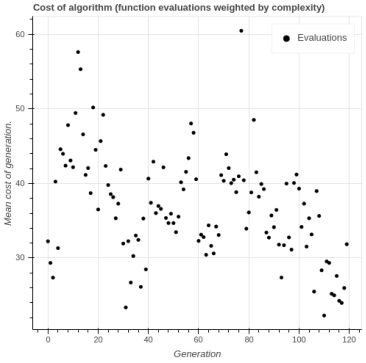
(a) Scaled complexity over generations.



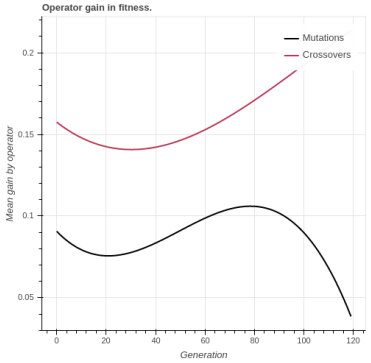
(b) Fitness values over generations.



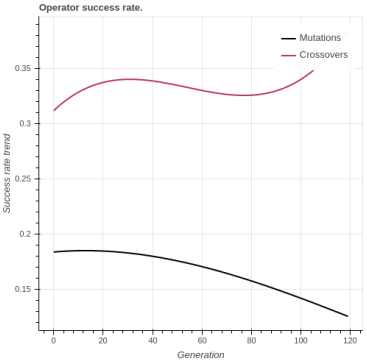
(c) Constant folding savings.



(d) Mean evaluation cost.



(e) Operator gain.



(f) Operator success rate.

Figure 6: Selection of visualizations generated by CSRM.

Fine grained parallelism In fine grained parallelism we parallelize a single step in the algorithm, where we execute a number of tasks in parallel that are independent of each other and where each task has a relative short completion time. The evaluation of the fitness function is a prime example of such a task. Evaluating fitness functions takes up the greatest part of the runtime in the algorithm. The fitness evaluation itself can be quite complex, but in comparison with the entire runtime of the algorithm the computational cost of a single calculation is small. In order to efficiently parallelize such a taskset we have to employ a mechanism that has minimum overhead. Shared memory parallelism using threads is ideal for this use case. Optimizations such as dynamically allocated threadpools will reduce the overhead even further, and thus increase the speedup. Overhead is split over the implementation overhead of starting, assigning and administering a thread, and the copying of the problem data and its solution. Shared memory implies a near zero cost in copying, only a reference is copied. A lock on the shared data is not needed, since the instance the thread operates on is not used by any other part of the program while the fitness function is executed. This savings both code complexity and synchronization overhead.

Coarse grained parallelism In coarse grained parallelism we run a series of tasks in parallel that have a long runtime or complex workload. In our setting an example of coarse grained parallelism is running the entire algorithm in parallel, with several instances tackling a distinct subset of the problem as separate processes. Because task now has a longer runtime of the task, the overhead of parallelizing the problem can be significantly greater without impacting the speedup obtained. Typically we only start and stop such a task once, in contrast with fine grained tasks which are constantly started and then stopped again. We can use processes or threads for this form of parallelism. Threads have the benefit of being lighter in comparison to processes in terms of overhead. If we allow communication between the tasks threads can elide copying the data, at the cost of introducing locks. Processes typically have to copy data in order to communicate. We can compare both approaches with an interrupt based approach versus message passing. Whichever we choose, communication between tasks requires synchronization of some form. This introduces not only memory and time overhead, but also significantly complicates the implementation. Invariants that hold in sequential implementations are no longer guaranteed in parallel, and a careful implementation is required.

Our approach

Evaluating fine grained parallelism Python has poor support for shared memory parallelism. While threading is available, it will not offer a speedup except for IO bound tasks due to the presence of global lock in the interpreter (GIL). It is possible to use compiled extensions in C to work around this issue, at a high cost in code complexity. A prototype implementation that parallelized the fitness function calculation proved that both threading and processes are unable to offer any speedup, often even running slower than sequential. A large part of this cost is due the overhead of copying Python objects, which are reference counted and thus require a graph traversal in order

to create full copies. In order to evaluate a population of n expression trees of depth between 5 and 10 in parallel, we have to first copy the n expressions to each of the n processes, then evaluate the tree, and then copy the result back. The copying operation is orders of magnitude more expensive than the evaluation function, with both scaling exponentially with the depth of the tree. A prototype using threads elided the copy, but the GIL ensured that performance was lower than the sequential approach. This ruled out fine grained parallelism in CSRM.

Evaluating coarse grained parallelism We create k processes, and give each an instance of the algorithm to execute in parallel. Communication is done via message passing. We use the MPI framework, which offers Python bindings, to allow processes to communicate and synchronize. Copying is still costly, but now a speedup is possible. Threading did not work in this approach due to the GIL. CSRM uses a set of communicating processes in order to solve the distributed SR problem.

5.2 Distributed SR

We will now discuss the benefits of the distributed (coarse grained) application to our problem.

Motivation A parallel metaheuristic has the obvious advantage of speedup in comparison with a sequential implementation. By dividing the problem over k processes we can, in ideal circumstances, obtain a speedup linear in k. With speedup we then refer to the time, or number of evaluations, needed to reach a certain threshold of fitness. The advantages of parallelization do not stop with this speedup. We can view the parallel processes as a swarm in itself, where each instance communicates with the others using a predefined pattern. Instead of k standalone processes we now have a set of k cooperating processes. Each process can now use the information of others to improve its own search. Using this approach a superlinear speedup is possible. There are, however, downsides. Communication implies overhead, not only in memory but also in synchronization. Without communication the only obstacle to obtain a linear speedup is dependent on the implementation. If we compare with the sequential process we need to clearly define measures to do so. If we run k processes, each with a population p, for g generations in r phases the entire process has executed k * p * g * r iterations. We cannot find an exact equivalent in the sequential algorithm. We could run the sequential algorithm k*r phases in order to simulate the same workload but the sequential and parallel algorithms will be different search processes. Increasing the phases or generations can easily lead to overfitting. Giving a sequential algorithm a population of k*p is not equivalent to the parallel algorithm. If we increase the population we need more generations and phases. If the number of generations is less than the population size not all expressions will have been able to use combine using crossover. Finally each of the k processes starts in a different part of the search space, on a different sample of the data. There is no direct translation between k parallel processes and a sequential process. In this work we will focus on the improvement in the quality

of solution. We will measure the difference in quality of solution between the parallel implementations and with a single sequential process.

Constraints With communication we introduce synchronization constraints which can bottleneck a subset of the processes, or in the worst case serialize the entire group. Our aim is to to exchange the most valuable information with the least amount of overhead. This balance is problem specific, the cost of copying depends on what exactly is being copied when and to whom it is sent. Even if we restrict us to a 1-1 link between two processes, and only exchange the fittest expressions between the two processes, we do not know in advance how large the expression we copy will be, as the depth and sparseness of the tree representing the expression will vary. While each process is given an equal sized subproblem, there are no guarantees that the actual workload of the different processes will be equal. We know that the evaluation of the fitness function has variable computational load. The stochastic nature of the metaheuristics used in the SR implementation compound this issue. By virtue of the problem statement we do not know the optimal solution to our problem, and with different starting points convergence between the different processes is likely to differ. We will address each of these issues.

5.3 Topology

A topology in this context is the set of communication links between the processes. The topology determines the convergence behavior of the entire group. In a disconnected topology, there is no diffusion of information. If a process discovers a highly fit expression, that knowledge has to be rediscovered by the other processes. The only edge case where this is an advantage is if we see the group of processes as a multiple restart versions of the sequential algorithm. If the risk for premature convergence due to local optima is high, we can try to avoid those optima by executing the algorithm in k instances, without communication. The implementation should offer the process an efficient way to lookup both processes from which it will receive information (sources) and processes it has to send to (targets). Source lookup is needed in order to solve the synchronization problem. Any topology that contains a cycle between processes can introduce a potential serialization effect at runtime.

Diffusion and concentration Our aim is for the processes to share information in order to accelerate the search process. With a topology we introduce two concepts : concentration and diffusion. Concentration refers to processes that share little or no information and focus on their own subset of the search space. Like exploitation concentration can lead to overfitting and premature convergence. It is warranted when the process is nearing the global optimum. Diffusion, in this work, is the process of sharing optimal solutions with other processes. Diffusion accelerates the optimization process of the entire group. It is not without risk, however. If diffusion is instant, a single suboptimal solution can dominate the other processes, leading to premature convergence. The topology will determine the synchronization characteristics of the processes, as well as the balance between diffusion and concentration.

Synchronization Synchronization between the processes will play an important role. While it does not directly influences convergence, it will constrain the runtime performance of the entire group. If we denote S_i and T_i as the set of processes that are sources and targets respectively for process i, we would like have S_i minimal. The message processing code will have to wait for the slowest processes in S_i before continuing. Without asynchronous communication process i would have to wait even longer, with the slowest process blocking the receipt of messages from the other processes. Synchronization implies that process i cannot send until its receiving stage has completed. If $S_i \cap T_i \neq \emptyset$ we have a cyclic dependency which can introduce deadlock in the implementation. By extension, if there is a cycle in the topology between any i, j processes this deadlock is a real risk. Dealing with this in the communication code is non trivial. We will show in section 5.4 how CSRM is able to deal efficiently with cycles in the topologies that it implements. Even with deadlock resolved, cycles will introduce a tendency for the the processes to serialize on the slowest process in the node. The time spent waiting in the communication stage is lost to the convergence process. We will show how this is mitigated in our implementation.

5.3.1 Grid

This topology arranges a set of k processes in a square 2D grid. Each process is connected with neighbouring processes along the dimensional axes. Some variations include diagonal links, or create 3 or higher dimensional meshes. The general idea behind the grid remains invariant in those configurations. A grid connects all processes, allowing for diffusion of the best solution to each individual process. The key observation here is that diffusion is gradual. While a process has an immediate neighbourhood, reaching all processes takes a variable amount of communication links. Diffusion of a dominating solution will take time, and if the solution is suboptimal this time allows the other processes to evolve their own optimal solutions thereby preventing premature convergence. This risk is only mitigated by gradual diffusion, not eliminated. Elimination is only possible in the extreme case by an absence of communication or in a more advanced configuration the usage of cliques. Nodes on the borders of the grid communicate with their counterparts at the mirrored side of the grid. This ensures that the communication process is symmetric. CSRM supports both square grids, where k is square of natural number, or incomplete squares. If $\sqrt{k} \neq n$ for some $n \in \mathbb{N}$ we create a grid that would fit j processes with j given by

$$\forall j \in \mathbb{N} \min(j) > k \wedge \sqrt{j} = n$$

This grid is filled row by row with k processes. This configuration should be avoided, as the communication pattern will not always be symmetrical. It is implemented to allow a fair comparison with other topologies where the processcount is not a square. In Figure 10 we see the communication pattern in a grid with 9 processes.

Cost The communication cost for k processes in a single iteration, with m messages sent per exchange, is in our configuration 4 mk. The synchronization constraints are high, all processes are interdependent (directly or at most $\sqrt{k}$ links removed). The

lookup of targets is static, at configuration each parallel process is given a simple integer indexed mapping, the symmetry of the mapping simplifies the lookup code.

5.3.2 Wheel

A wheel or circle topology connects all processes with a single link shared between each source and target. Diffusion is slower than compared to the grid. For k processes it takes k-1 iterations for a message to reach all processes. Some variations introduce a 'hub', with spokes reaching out to the circle itself, completing the wheel analogy. CSRM implements this topology as a simple circle without hub or spokes. A spoked wheel topology, if the hub has bidirectional communication with all processes, has a maximum distance of 2 between all processes offering fast diffusion. Without a hub but with bidirectional links the maximum distance is $\frac{k}{2}$. A unidirectional variant is shown in Figure 10a.

Cost For k processes with m messages sent per exchange the communication cost is km. This is a static configuration with symmetric lookup. A circle is a cycle, this results in high synchronization constraints. The variant with hub and spokes introduces even higher synchronization constraints. At a doubling of message cost the doubly linked circle has an significantly higher synchronization cost than the singly linked variant.

5.3.3 Random

In this topology a process selects a random target. We can configure it to do so statically, such that the target remains invariant at runtime, or select a new target after each phase. The number of targets is variable as well. CSRM implements all variants, allowing for both dynamic and static random topologies with a variable amount of targets per sending process. The idea behind a random topology is that it avoids communication patterns that are present in the structured topologies. If such a pattern leads to premature convergence, poor synchronization or fails to gain from the exchange of information between the processes there exists a possibility that a random approach can work. The downside is that we do not know what the maximum distance is between two processes, or even if they are connected. Cliques or cycles can form in the topology. Avoiding or detecting these requires more complex code than a simple random assignment of targets. By increasing the number of targets we decrease the probability for cliques while increasing the probability for cycles. Increasing targets will minimize the distance between two processes but increases synchronization and memory overhead. CSRM does not enforce constraints such as clique or cycle forming in its random topologies. In Figure 10d we see how a clique of cycles created by a random static configuration. A configuration with 2 links per process is shown in Figure 10c.

Cost The cost for a singly linked random static topology of k processes with m messages per exchange is clearly km. Synchronization constraints are unknown, and would have to be resolved by the processes. Cyclic dependencies between the processes are likely. The lookup code for a static configuration is simple, symmetric and fast. For a dynamic configuration the lookup is simple, but only symmetric at single points in

time. This observation is important because the virtual time of a process will diverge from that of the other processes. By allowing a dynamic configuration we introduce a new type of synchronization constraint. In CSRM processes have a copy of the global topology, but this is now no longer immutable. In order to resolve sources the process has to know the virtual time of the other processes. The topology remains deterministic, the seed used is identical between each process. Synchronization is highly complex in a random dynamic topology as cycles and cliques vary over time.

5.3.4 Tree

We introduce a binary tree topology. The links are unidirectional, with a single root and a given depth. The processcount should ideally be $k = 2^{d+1} - 1$ for some $d > 0 \in \mathbb{N}$ to create full binary tree. This is not a hard constraint, for values of not satisfying the constraint we construct a tree of depth $d = \lfloor \log_2 k \rfloor$ and fill the last level left to right until all processes are assigned a position. Communication is unidirectional, from the root to the leaves. This topology is free from cliques or cycles. The leaves act as sinks, while the root is a source. Lookup is fast, static and symmetric. Each process except the leaves has at most two targets, and one source (except the root). The tree topology offers a structured balance between diffusion and concentration. The distance between two processes range from 1 to $\log_2 k$. In the case of leaves the distance is infinite. Note that depending on the communication strategy, each subtree behaves as a sink. The subtrees will not share information instead concentrate on distinct parts of the search space. A variant on this topology is a singly linked list, where the maximum distance is k. The problem with this topology is that its diffusion scales linearly with k. The full binary tree configuration is shown in Figure 10b. An alternative to this configuration is an inverted tree. If we invert all edges we now have for a tree of depth d 2^d leaves as independent processes feeding their best expressions to 2^{d-1} nodes. Instead of each process receiving at most from 1 other process their best solutions, each non leafprocesses will now receive from two other processes. Diffusion in this topology is best described in terms of shared knowledge. As we go up the tree from the leaves each node progressively learns more of this shared knowledge. In this variant a node receives knowledge from both subtrees, instead of distributing it over its subtrees.

Cost For k processes and m messages per exchange (k-1)m messages are exchanged per iteration. Synchronization overhead is minimal, there are no cycles and a process is influenced at most by $\log_2 k$ other processes and influences at most k-1. With the exception of a disconnected topology the tree topology allows for the fastest speedup compared with grid, random and circle. The tree topology offers a structured alternative to the grid topology, with a staggered diffusion pattern. The tree topology saves a factor 4 in messaging cost compared to the grid topology. While this factor is constant, its effects are significant. For the same messaging cost a process in a grid topology can send trees of depth d, whereas in the tree topology it can send trees of depth d+2. This difference in depth allows for more expressive trees that can result in higher fitness value for the same communication cost. The disadvantage is that total diffusion is no longer possible.

5.3.5 Disconnected

The disconnected topology has zero diffusion and an infinite distance between processes. The only applications of this topology are in cases where the risk for premature convergence due to diffusion is high and can for some reason not be mitigated. Secondly it can serve as a comparison with the other topologies in order to measure the diffusion effect, memory and synchronization cost.

Cost Cost is near zero, no messages are sent nor is any synchronization needed. In practice the collecting of all results will still have to be done by either an elected process or a process statically assigned as collector. This holds true for all topologies.

5.4 Asynchronous communication

We have to tackle deadlock and synchronization delay in our parallel implementation. We will first describe the interaction between the processes and using this context demonstrate our solution.

Control flow The control flow of our parallel SR process is shown in Figure 7. A parallel process in our implementation executes a single phase of the algorithm, then collects at most m of its fittest expressions from the archive. The set of target processes is resolved using the topology, then the m messages are send to the targets. After the sending stage the process looks up its sources using the topology, and waits for messages from those sources. The received messages are decoded to expressions which are used by the algorithm for its next phase. The algorithm stores the expressions in its archive and then uses that archive to reseed the next phase. The archive is used for both external input and the best solutions from the previous phases. Since the archive has a fixed size, it will introduce an evolutionary pressure. A new expression will replace an existing if its fitness value is strictly lower. Allowing expressions with equal fitness values to replace each other can be useful in some cases where it allows the optimization process to traverse zero gradient areas in the topology of the fitness function. It can also lead premature convergence where a large subset of the fittest population holds identical fitness values with low to no diversity. CSRM enforces a strict order in its population in order to prevent this last scenario.

Waiting for messages If a process does not wait for messages from other processes the convergence behavior of the entire group non deterministic. In most platforms there are no hard guarantees about inter process scheduling. We can end up in extreme cases with a single process only sending and never consuming messages. This would break the intent of a structured topology. While this is an extreme example, even in average cases an extra level of non determinism is introduced without there being an explicit need for it. Not waiting for messages requires that messages are buffered by the parallel framework, and this can lead to internal buffers varying in size. We will show another approach where the deterministic execution of the parallel metaheuristic is retained. This does not imply that the process itself is no longer stochastic, only that we are

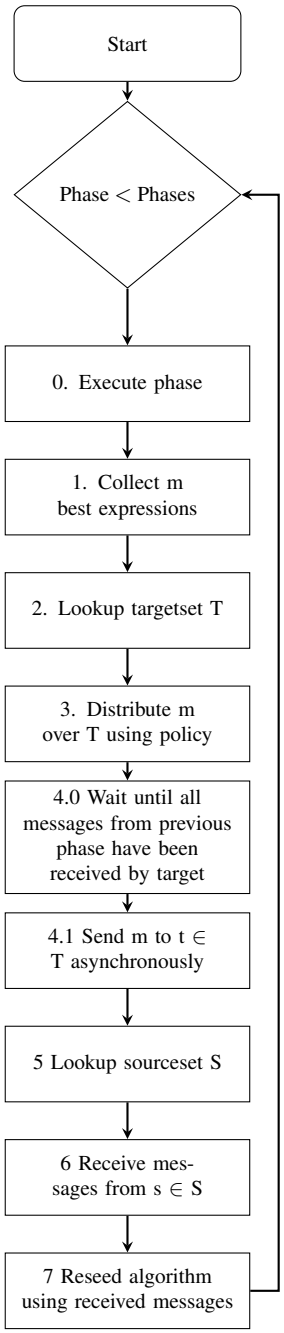


Figure 7: Parallel control flow.

guaranteed that our designed communication pattern is strictly adhered to. Our algorithm can be provided with seeds making it deterministic. Each different seed is likely to lead to a different solution, or at least a different starting point. Without determinism it becomes very difficult to accurately compare configurations of the algorithm.

Deadlock From our previous discussion we know that deadlock is a risk if there are cycles in the topology. Since each process has full knowledge of the topology (except in a random dynamic topology) it is possible to implement an algorithm that resolves deadlock by ordering the sending an receiving of the messages. Suppose A waits for B, and B for A in the most simple example. A simple solution of A sending to B, B receiving from A, then B sending to A and A receiving from B would break the deadlock. This simple interleaving solution is not unique, but the priority is not important here. As long as the deadlock is broken the processes can continue. There are 2 serious issues with this approach. First, this approach serializes concurrent processes. This is detrimental to any speedup we were hoping to achieve. Second, the implementation becomes more complex. The order of calls is no longer statically defined, we need an algorithm that acts as a central coordinator between any group of processes in a cycle, computes a solution, and executes that solution in order. We cannot directly invoke operations on other processes, so the only solution is to execute the coordinating algorithm in parallel on all processes if they detect that they are a member of a cycle. Each process will then know when it can send and receive based on the computed order. While this solves deadlock, we still end up with a serialized execution and the communication code becomes far more complex than a simple sequential sending and receiving call.

Asynchronous solution CSRM instead opts for an asynchronous sending of messages in order to resolve deadlock and mitigate the serialization. Cycle detection is no longer needed. A process executes a phase, then looks up its targets for messages. Instead of sending the messages with a blocking call to the framework, the sending process now allocates a buffer and sends each set of messages to its target with an asynchronous call. It stores a future object that can later be checked to verify if the receiving process has executed its receive call. The order of sending is no longer relevant, the sending call returns immediately for each target. Next the process calls receive for each of its sources. This is a blocking call, but this does not introduce a risk for deadlock. In the worst case our process waits until the source has stopped its own sending call, but due to its asynchronous nature this process is quite fast. The order of sources is no longer a risk for deadlock. After receiving all messages the process continues its execution as before. Each process still has clear the allocated buffers and callback objects from its send operation. The buffers cannot be deallocated as long as the receiving process has not acknowledged receipt of the contents. In order to check this we invoke the future objects, but care must be taken here. Calling the futures for each receiving process results in a blocking call. If we time this operation incorrectly we simply have deadlock all over again. The latest a process can wait is that point in time where the send buffers are needed again, which is in the next iteration. As soon as a process enter the sending phase, the first operation it executes is waiting on the future objects. After

each returning call the corresponding buffer is cleared for reuse. It should be clear that this approach does not simply delays deadlock but prevents it from occurring. The asynchronous approach allows the processes to interleave in any order that resolves the deadlock. Instead of an explicit algorithm, we simply let each cycle of processes resolve the deadlock in their own optimal sequence. Finally note that the receive call returns immediately once the framework has registered the corresponding send call, it is not directly waiting on the sending process.

Synchronization delay We know from our previous discussion that the execution time of a single phase will vary per process, and even between phases for a single process. The receiving of messages is a synchronization point between processes, but not a strict one as the sending process will only wait for receipt in the next phase. This means that a process will only be waiting on any other process after completion of a phase. If we introduce drift as the virtual time difference between processes, in order for processes to delay each other the drift would have to be greater than the execution time of a single phase. A strict serialization effect is therefore not possible. What we introduce with this implementation is latency buffer equal to the runtime of the next phase. The time to execute a phase, while variable, will still be on average far greater than the time needed to communicate results. One of our aims in distributing SR is obtaining a speedup by, amongst other things, finding a balance between phase runtime and communication time.

Delay tolerance without cycles In Figure 8 we visualize how our approach allows a faster process to avoid waiting on a slower process. Solid vertical lines are blocking calls, the line represents the time spent waiting. The receive and check calls are blocking but return immediately. The send call is asynchronous and returns immediately. Asynchronous calls start the lifetime of a future object visualized with a vertical dotted line. The object's lifetime ends with a corresponding call from another process. The length of a phase is the delay tolerance that prevents strict serialization. We see three processes in a topology without cycles, where the phase time t_p follows this pattern : $t_{pa} < t_{pb} < t_{pc}$. Process A can at most tolerate a delay of a single phase. If there is a strict ordering in average phase time between processes, then the processes will end up being serialized if the delay has exceeded a single phase. The average phase runtime will vary over time for a single process due to the ever evolving populations of differing depth and evaluation functions of varying complexity. The very fact that the runtime varies ensures that, with the exception of edge cases, the phase runtime average between processes will tend to the same values. Our approach allows the avoidance of serialization in the general case, given that the variation between the runtime will not be too extreme. The runtime of a single phase is dimensioned by the population and the number of generations. If we increase either one the runtime will increase, but the effect on the variation is far less predictable. In practice the runtime will depend on the average depth in the population, and the average complexity of the population. With the depth limited the complexity is limited as well. The only remaining variable influencing runtime is then the scheduling algorithm of the operating system. With the exception of real time operating systems there is no upper limit here, but in practice

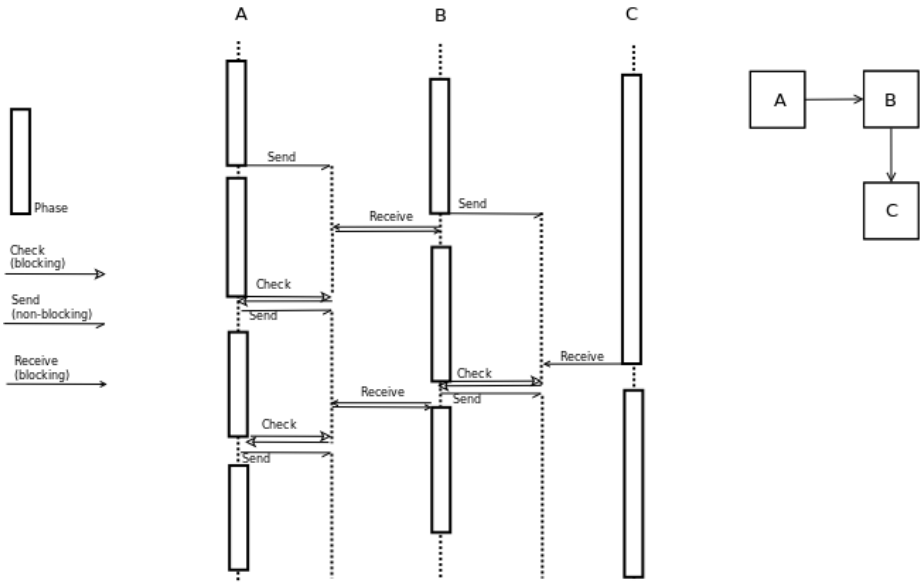


Figure 8: Synchronization delay tolerance in CSRM.

the average runtime of a process on a system that is not overloaded will tend to a constant. This means that we can estimate a distribution of the phase runtime for a single sequential process, and use the mean and standard deviation of that distribution to find a value that prevents with a high probability serialization.

Delay tolerance in the presence of cycles When two processes have a cyclic dependency, that is that they wait on each other's communication, the delay tolerance cannot avoid serialization. In Figure 9 we see how processes A and B are serialized due to their interdependency. This cannot be avoided, process A needs the messages from B and vice versa in order to proceed. In the figure we see clearly the receive call waiting until a corresponding send call has been issues. While the send call returns immediately, the receive call has to block until send has been invoked. The same applies for the check call, until receive returns the check call blocks. In the figure the time spent waiting is visualized with a solid vertical line. This figure demonstrates our deadlock resolution method. If the send call would block for either process, then both A and B would wait indefinitely. By making this operation return immediately the deadlock is resolved. This generalizes to larger cycles, for example in the wheel or grid topologies. Without this implementation only tree topology or random topology with cycle detection could work.

Future improvements If we allocate a buffer per communication phase, we can wait even longer before invoking future objects. This means we can leave a configurable amount of phases before we wait on the receipt of messages. The downside of this approach is that the memory constraints for each process now signficantly increase

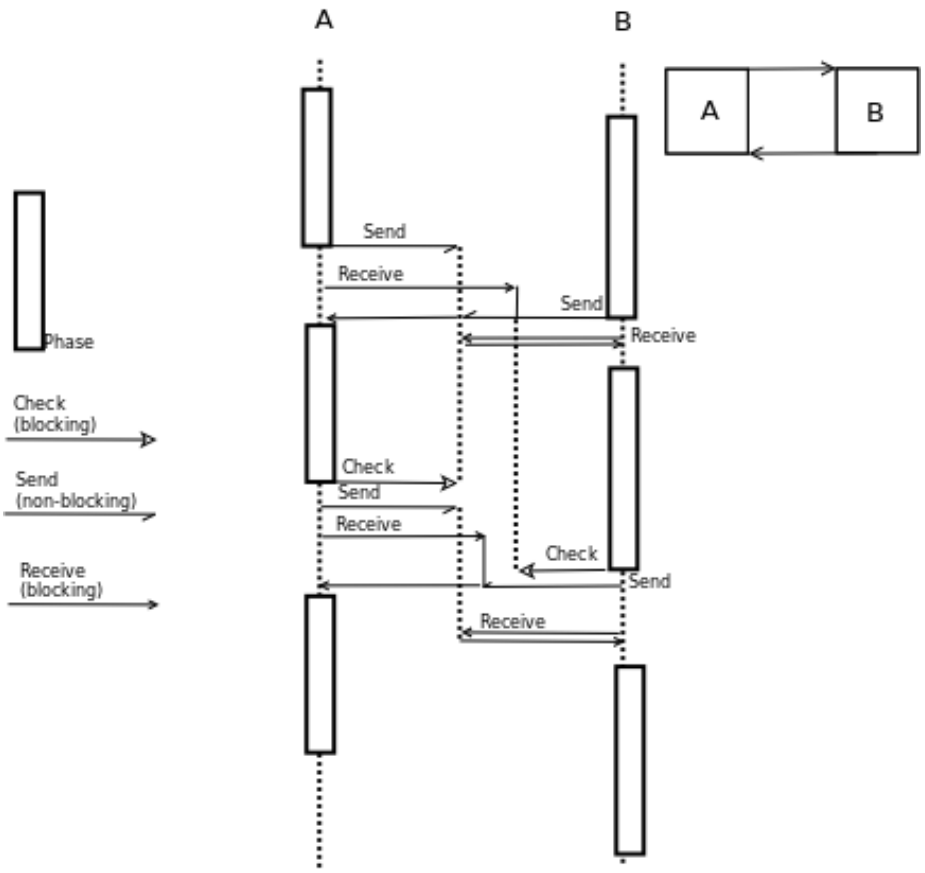


Figure 9: Synchronization delay tolerance in CSRM in the presence of cycles.

from m to pm where p is the number of phases we opt to wait. The receive call is executed in sequence for the list of sources in the target process. In order to minimize delay even further we could execute this call asynchronously as well. Extra code would need to be introduced to handle the non deterministic behavior this introduces. The order in which the messages are received can influence the algorithm. When the archive is seeded with these external expressions we drop any expressions with identical fitness values. There is thus a risk that by varying the order we introduce non determinism in our results. This can be resolved in the receiving code by preallocating the buffer for all sources and after all asynchronous receive calls storing the messages in the same order of the sources list. Asynchronous receiving is not implemented in CSRM.

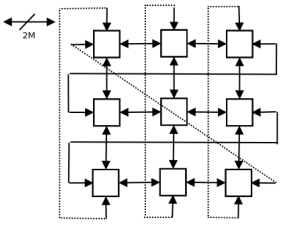
5.5 Communication strategies

If the process has j targets, and the algorithm is configured to distribute the m best solutions, we can use several approaches to send those to their targets. We can spread m over j, using a random, interleaving, slicing or any other type of sampling technique. An alternative is copying m j times so that each target receives m messages. CSRM has a Spreadpolicy interface that hides the implementation of this behavior for the processes. CSRM defaults to a slicing policy. If we have m messages to distribute over j targets, we will sequentially assign each of the j process $\lfloor m/j \rfloor$ messages. This policy is efficient as it avoids copying but has a direct effect on the diffusion between the processes. First of all the value of m should be chosen with the topology in mind. With this policy a value of m = 2 for a grid pattern is problematic. A grid requires 4 outgoing messages. If we only send 2, the intended diffusion pattern is no longer valid. To avoid issues like this the spread policy will default to a copying strategy when m is insufficient for all links. In other words, m = 2 with 4 required messages will result in the m messages being copied to each outgoing link. For a tree topology m should be 2 as well, else we create a symmetric series of cliques, which is unlikely be intended. If we use a copy policy the value of m can be as low as 1. For random topologies m depends on the parameter determining the number of targets. In our circle topology m = 1 is sufficient for both policies.

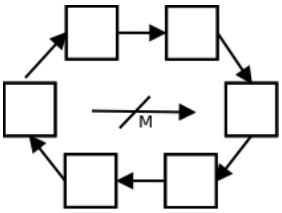
5.6 Exploiting parallelism for validation

5.6.1 Predictive capability

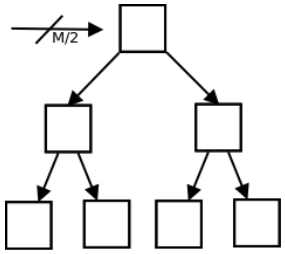
In SR we try to find an expression that, based on some distance function, fits input data as close as possible to expected data. If we do not use test data to score the resulting expression, the result of overfitting is very real. Not only is it possible that the expression fits the new data points from the test set poorly, the new datapoints may fall outside the domain of the generated expression. By scoring the expression on unknown data we measure its predictive capability P_{sr}. In sequential mode, CSRM evolves an expression on training data, then scores the expression on the full data. We record the correlation between the fitness on the training data and the fitness on the full data in order to measure, over generations and phases, the convergence process. We would like to have an answer to the question : How does P_{sr} of the SR process evolve over



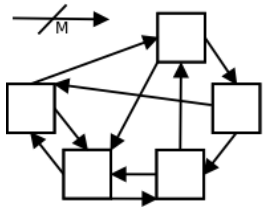
(a) Grid topology with k = 9.



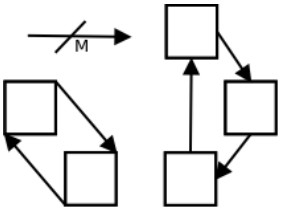
(b) Circle topology with k = 6.



(c) Tree topology with k = 7



(d) Random topology with 2 outgoing links per process, k = 5.



(e) Random topology resulting in cliques of cycles, k = 5.

Figure 10: Selection of visualizations generated by CSRM.

time?
This question is vital to a practitioner. If we have an indication that prediction is no longer increasing, or even decreasing we should halt the process. On the other hand if we see that a linear increase is still present we can opt for extending the runtime of the algorithm. Conventional validation, as described here, can be replaced with cross validation in order to obtain a more accurate measure for P_{sr}.

Cross validation Several approaches in cross validation exist. We distinguish between exhaustive and non-exhaustive methods. The first uses all possible combinations of training and test data to obtain the maximum amount of information of P_{sr}. Its disadvantage is a high computational cost, although this can still be linear in the length of the dataset if we omit only a single datapoint per selection from the dataset. We will focus on k-fold cross validation (KCV), a non exhaustive validation technique. In KCV we split the data over k samples, then use k-1 of those as the training data and one as the validation data. The process is repeated k times, to obtain full coverage. Even though this approach is non-exhaustive it would still require k iterations of the entire algorithm.

5.6.2 Parallelization

Approximating k fold cross validation With the distributed implementation of SR we can avoid the factor k increase required for KCV. CSRM can, as we have shown, run n instances of the algorithm in parallel. Those instances can cooperate by exchanging their current best solutions, possibly accelerating the convergence of the entire process. Instead of translating our sequential approach to the distributed approach and giving each process the same training data, we can approximate KCV without the linear increase in runtime. We do not implement full KCV, but mimic the process. For a dataset of length d, k fold will split it into k equal sized sections of size s = $\frac{d}{k}$. Each training set has $s * (k - 2)$ overlapping datapoints. The value of k determines the effect of the outcome to a large extent. We would like to keep the overlap above a certain treshold, insensitive to k. If the overlap is too small the probability of highly fit expressions proving to be invalid on the validation data becomes too large.

Approach With k processes running in parallel, and d datapoints, we give each process a random sample of size r*d with $r \in [0, 1]$. In practice we use r = $\frac{4}{5}$. This is the same approach we use in our sequential implementation. The difference now is that each process is given a different trainingset of size r*d, with a stochastically determined overlap threshold. Each process uses its own distinct trainingset to evolve expressions from and at the end of execution scores its population on the full data set. Each process uses a different seed for its random number generators, so we have k processes working on a different area of the search space, but there is enough shared information to make the results of each individual process relevant to each other. Our approach is visualized in Figure 12 with $r = \frac{3}{4}$ and k = 4. The probability that any two trainingsets share a single datapoint is r^2. This probability is invariant to k, the number of processes. So for any two communicating processes, regardless of the topology or

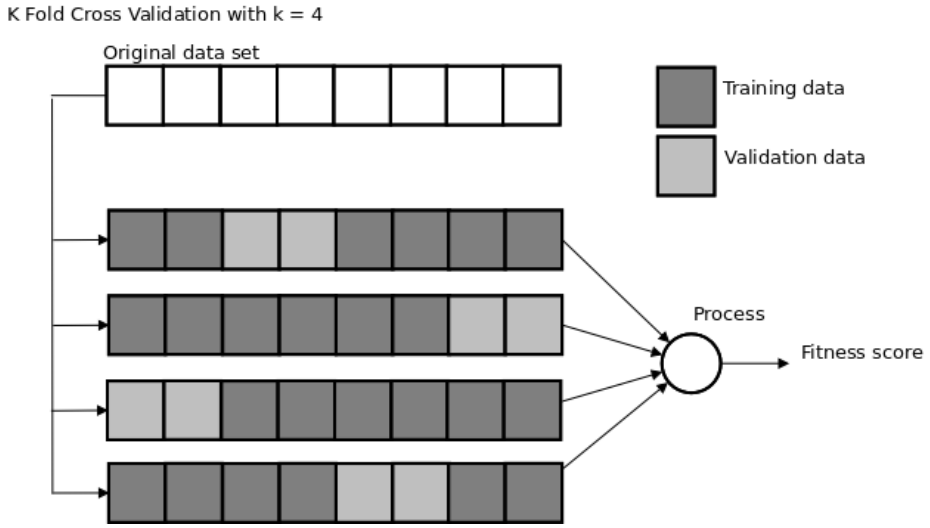


Figure 11: Visualization of k fold cross validation with k = 4.

validation process, the ratio of shared data points is constant. When two processes exchange their best expression, the new expression is scored using the receiving process' training data set. The invalid expression problem is not only present at the end of the algorithm, it is also a vital factor during the communication between the processes. If process A evolves an expression that has a high probability of being invalid on process B's trainingset, the possible gain we have in sharing that expression evaporates. By establishing a constant threshold we can mitigate this risk and still gain from the validation process. The choice of concurrent processes is constrained in practice, typically it is determined by the available hardware or the topology. Our validation approach is insensitive to this value.

5.7 Conclusion

We have covered our distributed design and highlighted the issues that drove our choices. CSRM offers the practitioner several topologies each with its strengths and weaknesses. With a wealth of topologies and configurations we also increase the dimensions of the parameter optimization problem. In the experiments we will evaluate some, but not all parameter choices. Our implementation can easily be extended with more topologies or policies without risking deadlock or serialization. Our validation approach in parallel CSRM allows an approximation of KCV without increasing the runtime cost by a factor k. The validation is insensitive to the number of processes and finds a balance between generating predictive expressions while minimizing the occurrence of invalid expressions.

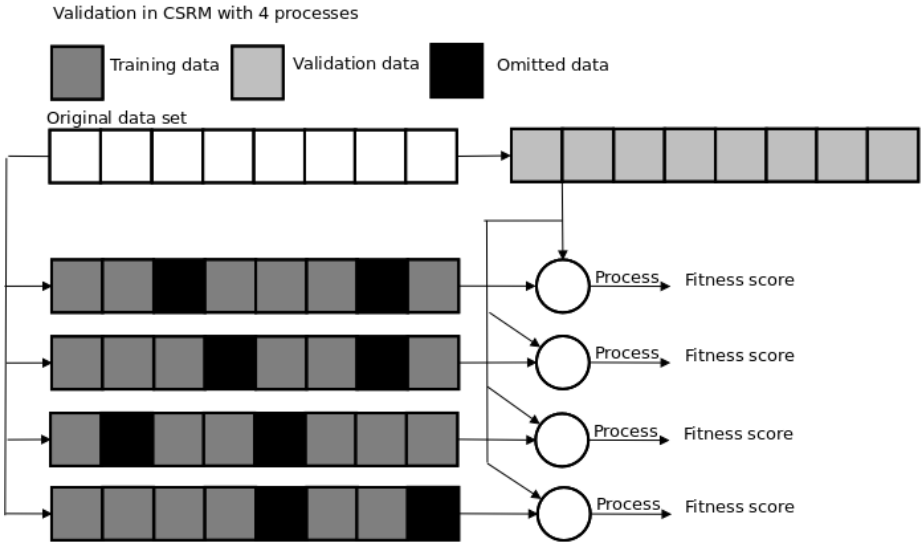


Figure 12: Approximation of k fold cross validation with parallel processes, k = 4, r = $\frac{3}{4}$.

6 Constant optimization

6.1 Constant Optimization

Selecting base functions is a combinatoric, discrete problem. Selecting the right constants to use is a continuous problem, that GP tries to solve with a combinatoric approach. There are several aspects to this issue, we will go over each individually and show our approach.

6.1.1 Restricting the search space

From section 3.2.1 we know that the size of the constant set dominates the size of the search space. We intentionally restrict this set to [0,1]. This reduces the size of c from 2^64 to 2^23. This restriction in range does not prevent larger constants from being evolved. The algorithm will evolve subtrees combining base functions with constants in order to approximate the desired values. Selection [0,1] as the reduced range is a logical choice given that floating point numbers have the highest density in this range. Despite the significant reduction in size of c, it still dominates the search space size.

6.1.2 Initialization revisited

During initialization it is possible that a generated tree represents a constant expression. Such an expression is only a valid approximation if no feature has an influence on Y, which is an unlikely edge case. A constant expression is of no use to the algorithm, but without measures to prevent or remove these they will still be formed. Ideally such an

expression will have a worse fitness value than non constant expressions, and will be eventually filtered out. Since detecting a constant expression is feasible in worst case O(n) with n the number of nodes in the tree, a more efficient approach is preventing constant expressions from being generated.

Constant expression detection An expression is a constant expression if all its children are constant expressions. As a base case, a leaf node is a constant expression if it is not a feature. This problem statement allows us to define a recursive algorithm to detect constant expressions. It should be noted that its complexity is O(n) only in the worst case, when the tree is a constant expression. Upon detecting a non constant subtree, the algorithm returns early without a full traversal.

Preventing constant expressions Using the checking procedure in the initialization, a tree marked as a constant expression is not allowed in the initialization procedure. It is still possible to create constant expressions by applying mutation and crossover. If the left subtree of a node is a constant expression, and the right is not, and this right is replaced by either mutation or crossover with a constant expression then the tree becomes a constant expression. The mutation operator will not generate constant subtrees, so this leaves only crossover. Our tool does not prevent constant expressions from forming in this way, the evaluation following crossover will filter out the constant expressions using evolutionary pressure.

6.1.3 Folding

Constant subtree problem A tree can contain subtrees that represent constant expressions. This is an immediate effect of the GP algorithm trying to evolve the correct constant. This can lead to large subtrees that can be represented by a single node. Nodes used in such a constant subtree are not available for base functions. They waste memory and evaluation cost, without their presence the tree could become a fitter instance. There is a counterargument to be made here: parts of a constant subtree can help evolve constants faster than a pure random selection. It is possible that folding such subtrees leads to slightly lower fitness values.

Constant subtree folding We can use a depth sensitive objective function to try to mitigate this effect, but a more direct approach is replacing the subtrees. Using the previous constant expression detection technique we can collect all constant subtrees from a tree. We evaluate each subtree, and replace it with the constant value it represents. Constant folding requires an O(n) detection check, since the entire tree needs to be traversed. The folding operation itself is at worst O(n), if the entire tree is constant.

Analysis This leads to savings in nodes and possibly a reduction in depth. These savings can have an effect on the convergence as well. Mutation and crossover will no longer operate on constant subtrees, and the iterations and place in the tree that becomes available can be used to improve the fitness of the tree. These two advantages are intuitive and expected. There is also a counterintuitive disadvantage to constant

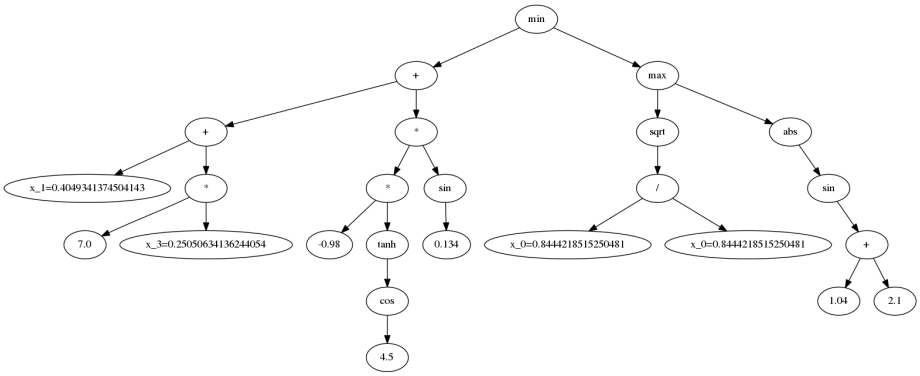


Figure 13: Tree before subtree folding.

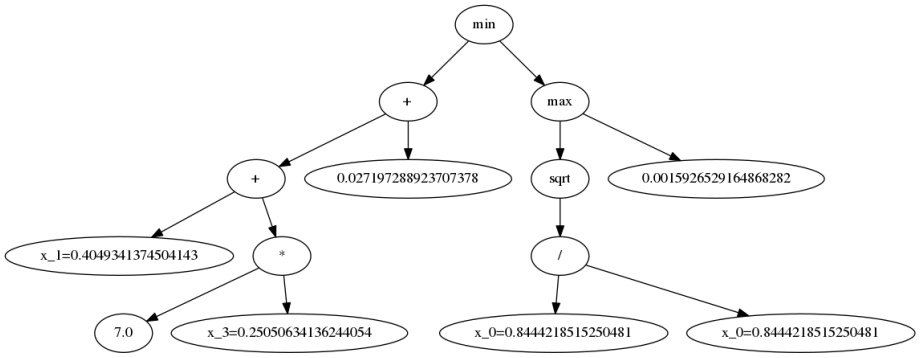


Figure 14: Tree after subtree folding.

folding. A constant subtree is the algorithm's attempt at evolving a single constant. In constant folding we assume that the subtree only holds the information needed to represent this single constant, and therefore is more efficiently represented by that single constant. This is not true. A constant subtree with depth d represents between d and $2^{d+1} - 1$ constants. Each leaf is by definition a constant. Each internal node is the root of a constant subtree. Each of those represents a new constant, which is eventually combined to the root of the largest constant subtree. Removing the entire subtree by folding it into a single constant can actually lead to worse fitness. To see this we need to look at constant subtrees as a set of discovered constants. The GP algorithm has 2 ways to generate a constant : crossover and mutation. Mutation will generate a subtree and replace it in an existing tree. In order to generate a constant a constant subtree will need to be generated. This process is unaffected by folding. What is affected is when the mutation target node is a node in a constant subtree. Mutation is able to generate far more complex expressions than a single selection of a constant leaf node is able to do, it is evolving the existing subtree. If we apply constant folding mutation can only replace the existing constant with a new one. The probability of this new value having better fitness is lower compared to mutation the subtree that represents the constant. In crossover the effect is even stronger. The constant subtrees of a tree form a library of constants from which crossover can select and combine to form a new one. Given the exponential number of constants represented by such a subtree it is clear that expressive power of crossover without constant folding is greater than with constant folding applied.

Edge cases In Figure 14 we see the effect of applying the constant subtree folding. There are subtle edge cases that can't be detected using the above method. Consider the tree representing the expression

$$f(x) = \max(x, 4 + \sin(0.3))$$

The subexpression 4 + sin(0.3) is detected and folded into the constant 4.26. We should not stop there, since f can still be a constant expression if $\forall i : x_i < 4.26$. To verify this we need to evaluate f on all i datapoints. In contrast, the constant subtree detection code needs only 1 evaluation. In Figure 14 we see a similar case in the right subtree where the first value of x_0 the right subtree is indeed constant. Even if we should evaluate f for all i, this will not guarantee us that f is indeed a constant expression. All this check then proves is that for all i in the training data, f is constant. The testing data holds new values, for which f may or may not be constant. We conclude that this edge case cannot be prevented from occurring. Another case occurs in expressions of the form

$$f(x, y) = \tan(y) * \sqrt{x/x}$$

In this reduced example the node $\sqrt{x/x}$ is simply $\sqrt{1}$. Outside of this example, detecting such cases is non trivial. There is a clear benefit to do so, despite their low occurrence: if the domain of x includes 0 this tree is never generated because it leads to division by zero in a single datapoint. Discarding this expression is however not needed, since $\sqrt{1}$ is a valid subexpression. One way to solve this is to use mathemati-

cal software to simplify the expressions, and then convert them back to tree representations. Deciding to apply this step should be based on the cost versus benefit and the frequency of such occurrences.

Conclusion Constant detection and folding mitigate some of the side effects of the constant optimization problem. We have discussed the advantages and the risks of applying constant folding. In general it is still advised to apply constant folding in order to constrain the excessive growth of trees in the population. In section **??** we will analyse the effect of constant folding. Constant folding itself does not solve the constant optimization problem. For this we need a continuous optimizer, an algorithm designed specifically to optimize real valued problems.

6.2 Optimizers

Hybridizing GP Using a real valued optimizer in combination with GP is a known solution [22, 7]. Which algorithm to combine is a difficult question. To our knowledge there is no comparison made between optimizaton algorithms to find out which is a better fit to combine with GP.

Problem statement Given a tree with k constant leaves, with all constant subtrees folded, we would like to find the most optimal values for those constants resulting in a better fitness. It is vital to perform the constant folding step before optimization takes place. Suppose a given tree has on average k constants, which after folding become j with j $<=$ k. Without folding the optimizer has to solve a k dimensional optimization problem, whereas after folding the problem is only j dimensional. The underlying approximation only has j dimensions, so this step is a necessity.

Initialization In most optimization problems the initial solution is not known, only the problem domain and an objective function. The problem we face here does have an initial solution, namely that generated by GP. Instead of choosing random points in the search space, we therefore opt by perturbing this initial solution. An immediate problem here is that the optimizer may simply converge on the initial solution. This risk can be high, given that GP already has evolved it as a suboptimal solution. The problem faced by the population initialization step 4.3.1 reappears here. We could pick random values in the search space, but these are likely to generate invalid trees. A balance between exploration and exploitation is once again called for.

Domain Each tree instance has a completely different domain for each constant. We cannot guide the optimizer with domain specific knowledge. This also reflects in the choice of parameters for the optimizer, which will have to be suboptimal for specific problem instances, as we want them to work on the largest set of problems.

Comparison and configuration In order to keep the comparison between the algorithms fair they are each configured as similar as is possible. Since they are all population based, and given a maximum number of iterations, all three share these same

parameter values. Each application of the algorithm has a significant cost in comparison with the main GP algorithm. In a single iteration with population p, the GP algorithm is likely to perform 2n evaluations (mutation and crossover). If we give the optimizer a population of m and q iterations, it will execute at most in the order of m x q evaluations. Based on the optimal value for PSO [13] we use a default population of 50. The iterations are set at 50. This means the cost of the optimizer will quickly dominate that of the main GP algorithm, depending on when and on what we apply it. We can apply the optimizer on each expression each iteration, only on a selection of the best expressions at each iteration, or only at the end of phase on all, a selection, or only the best expressions. In our experiments we will show the effect of these choices on convergence.

6.2.1 ABC

Artificial Bee Colony [11] is a relatively new nature inspired optimization algorithm. It is not limited to continuous optimization, and has even been used for symbolic regression itself [12]. One of its key advantages over other algorithms is a lower parameter count. Optimal values for an optimization algorithm have a large impact on its convergence behavior, so much so that other optimizers can be required to find the parameters of an optimizer. With a small parameter set finding optimal values becomes easier. Finding optimal values for these parameters is quite often related to the problem domain, and as we have seen each instance here will have a new domain. ABC is good at exploration (thanks to the scouting phase) though sometimes lacking in exploitation. To resolve this ABC can be combined with more exploitative algorithms [20].

Algorithm ABC is a population based algorithm, using 3 distinct phases per iteration. We will refrain from using the nature analogy in describing the algorithm as it can be confusing. The algorithm maintains a set of potential solutions and a population of particles. Each particle is either employed, onlooker, or scout. An employed particle perturbs a known solution, and if an improvement in fitness is obtained replaces the old solution. If this fails a preset number of iterations, the solution is discarded and a new one scouted. Scouting in this context is generating a new potential solution. After the employed phase the onlooking particles decide, based on a fitness weighted probability, which solutions should be further optimized. In contrast to the employed particles they swap out solutions each iterations, whereas an employed particle is linked to a single solution. Finally exhausted solutions are replaced by scouted solutions.

Initialization The algorithm initializes its solution set by perturbing it. Perturbation is done by multiplying each constant with a random number $\in [-1, 1]$. Using the modified constants the tree recalculates its fitness value and the global best. Each instance records its best solution. The original source, the expression tree that we wish to optimize, is retained as a solution, and not perturbed. This can lead to premature convergence if the algorithm is configured to focus too much on exploitation versus exploration.

Modification In the employed stage, a solution x at time i is modified by combining it with another randomly chosen solution y. The choice for y is made with replacement, and obviously y $\neq$ x. With k random $\in [0, |x|)$

$$\begin{cases} x_{ij} = x_{ij} & j \neq k \\ x_{ij} = y_{ij} & j = k \end{cases}$$

The employed particle tries to improve the current source. If the modification leads to an improved fitness value, the solution is updated. Note that only a strict improvement warrants an update, an equal fitness value will not lead to an update. The onlooker phase is executed next. Each onlooker is assigned a solution using roulette wheel selection. For the set of solutions S we calculate fitness weights using:

$$w_i = \frac{1}{1 + f_i} \forall i \in S$$

Optimization in CSRM is a fitness minimization process, which leads to the fraction. A list is built using a cumulative sum of these weights. From this a uniform random number picks a single value based on these weights. A smaller fitness value will have a relatively larger section of the list compared with a larger fitness value resulting in a fitness bias in selection. Once assigned to an onlooker, the same modification process used by the employed particle is applied. Since selection is done with replacement it is possible that a single highly fit value is modified more than twice in an iteration. After the onlooker phase the algorithm checks in the scouting phase if any sources are exhausted. Exhaustion indicates that a solution can't be improved upon for at least *limit* modifications. Depending on the fitness value this limit can be reached faster for more fit values. There are several edge cases to consider here. Clearly this approach is beneficial if the exhausted solution has a poor fitness value. Improvement was impossible, so replacement is likely to improve the overall quality of the solution set by introducing new information. There is however the risk that algorithm discards highly fit solutions that fail to be improved. Discarding such a solution is warranted if that solution is a local optimum, but by the very nature of the problem statement we cannot know this in advance. We could prohibit discarding the best solution, even if the improvement limit has been exceeded. This does not solve all edge cases. If ABC is used in a multimodal search, it is still possible that valid solutions are discarded. Compounding the problem is that an equality update is not used. In CSRM we implement a strict check, so it is possible that equivalent solutions are discarded. With s scouts and e exhausted solutions $min(s, e)$ solutions will be replaced with new solutions. These are generated using a Normal distribution. In most optimization problem good starting positions are not known and thus a random point is selected. In our problem we already have a reasonably good solution, so we use this initial value as the mean of the normal distribution and generate values within 2 standard deviations around that mean. A configurable scaling factor is introduced, in our test problems we use 20. This value is trade-off between exploration and an increasing probability for generating invalid solutions. The values chosen have a far greater range than those generated by the initialization process. We do not know the domain of our problem, nor do we have the computational resources to cover the entire floating point range. The initial value is

already evolved as a fit solution to our problem. The probability that the true optimum is far beyond the range of our initial value is estimated as low, though we can never be sure of this. The scouted solution replaces the exhausted solution. In our problem statement this can lead to issues. There is no guarantee that the scouted solution is actually valid. As we have seen in the initialization problem 4.3.1 this probability can be quite high. It is possible that an increasing part of the solutions are invalid. These will still be chosen to contribute in the modification step. It is unlikely though not impossible that they contribute to the convergence. In the worst case with enough iterations and a very small domain there it is possible that the entire population is replaced with invalid solutions. In our implementation the values of the threshold regulating exhaustion are chosen such that this is unlikely to occur. One solution here is to apply the same generation loop used in the GP algorithm, which keeps generating solutions until a valid one is found. Given the already high cost in evaluations the optimizer introduces we have chosen not to use this here. This applies for all optimizers implemented in CSRM, not just ABC. Note that the same problem is present in the initialization step, although far less severe given the small perturbation applied there.

Selection A solution is updated if the fitness value is improved. The new fitness weights for the roulette wheel selection are calculated and the global best is updated. Unlike PSO a solution is only updated if an improvement is measured. In contrast to DE, equal fitness values do not lead to an update. An equality update allows an optimization algorithm to cross zero gradient areas in the fitness landscape. The influence other solutions have on each other in ABC is not as great as in DE. From the modification stage we also observe that at most 2 dimensions per iteration per solution are modified. In PSO the entire position, in all dimensions, is updated. In DE this depends on the Cr parameter which we discuss in 6.2.3. This distinction can have a large impact on convergence. The balance sought here is influenced by the interdependence of the constants. The modifications made by the algorithm can be seen as a process trying to extract information about this dependency. Suppose we have k constants, of which 2 have a large effect on the result. Then it only makes sense to modify those 2 dimensions, modification in the others does not gain anything except noise. The problem becomes more difficult if those two are correlated. Modifying all dimensions will not guide our optimization process as clearly as modifying only those 2. Modifying only a single one of the 2, as in ABC, is too strict as we lose the information about the correlation. We do not know in advance if our problem instances are separable or not. Related to this discussion, PSO can be sensitive to a bias along the axes of the dimensions [25] with improvements suggested in recent work [2].

Cost With a solution set of n, m employed, j scouts, i onlookers and k iterations we now look at the evaluation cost. Initialization requires n evaluations. Each iteration we execute m evaluations in the employed phase, i evaluations in the onlooker phase and at most j evaluations in the scouting phase. With m, j and i all $<=$ to n we have per iteration at most 3n evaluations. With our configuration, listed below, this value will be at most 2n. This results in an evaluation complexity of n + k 2 n, or O(kn).

Configuration

- limit = 0.75 * onlookers / 2 : If a solution can't be improved after this many iterations, it is marked exhausted and will be scouted for a new value. This limit is scaled by the number of dimensions per instance.
- population = 50 : This is the solution set, or set of sources.
- onlookers = 25 : The number of onlookers, instances that will be assigned solutions to exploit based on fitness values. Setting this value to half that of the employed finds a balance between exploitation and evaluation cost.
- employed = 50 : Instances that try to improve an assigned solution. If we use a value lower than the solution set we have to define an assignment procedure, which would mimick the onlooker phase. We therefore set the employed count equal to the size of the solutions set.
- scouts = 25 : This is a maximum value, up to this number are used to scout after a solution is exhausted. A higher scouting value leads to more exploration, a lower value favors exploitation. More exploration would result in the initialization problem dominating the runtime cost of the optimizer.

This configuration is guided by the findings in [11].

6.2.2 PSO

Particle Swarm Optimization [13] is one of the oldest population based metaheuristics. It consists of n particles that share information with each other about the global best solution.

Algorithm

Initialization Each particle is assigned a n dimensional position in the search space. A particle's position is updated using its velocity. This last is influenced by information from the global best and the local best. The concept of inertia is used to prevent velocity explosion [3]. Each particle is given a random location at start. In our application we already have an (sub)optimal solution, the constant values in the tree instances have been evolved by the GP algorithm. Rather than pick random values, we perturb the existing solution. This is a difficult trade-off. If the perturbation is not large enough the optimizer will simply converge on the known solution. If we perturb too greatly the risk for invalid solutions increases, rendering a large selection of the population invalid. We can initialize the population with n perturbed solutions or with n-1 with the initial value remaining intact. The n-1 solution is useful to test the algorithm, ideally the swarm will converge on the known best value. When applying the optimizer the n-perturbation approach is used, minimizing the risk for premature convergence. CSRM multiplies each constant with a random value in [0,1]. Each particle is assigned a small but non-zero velocity. The reason for this is again avoiding premature convergence. Without this velocity all particles are immediately attracted to

the first global best. While attraction to this value is desired, it should not dominate the population. The small value of the initial velocity once again reflects an empirical discovered balance between exploration and exploitation. Each particle is assigned an inertiaweight. This value is one approach to combat the velocity explosion problem, which we will cover in 6.2.2. Finally the global best is recorded.

Modification The algorithm updates all particles in sequence, then records the new global best. Let d be the dimension of the problem, or in our case the number of constants in the tree to optimize. The velocity v at iteration i of a particle is updated using:

$$v_{i+1j} = w_i * v_{ij} + C_1 * (p_{ij} - g_{ij}) * R_1 + C_2 * (p_{ij} - G_{ij}) * R_2 \forall j \in [0, d)$$

with

- v_i Current velocity
- p_i Current position (set of constant values)
- g_i Local best
- G_i Global best
- C_1 Constant weight influencing the effect the local best has on the velocity.
- C_2 Constant weight influencing the effect the global best has on the velocity.
- w_i Inertia weight simulating physical inertia.
- R_1 Random value perturbing the effect the local best has on the velocity.
- R_2 Random value perturbing the effect the global best has on the velocity.

Without the inertia weight PSO has issues with velocity explosion, the velocity has a tendency to increase to large values. This increases the distance between particles, but more importantly is far more likely to generate positions that are no longer inside the domain of one or more of the dimensions. Inertia weighting will dampen this effect. The position is updated using :

$$x_{i+1,j} = x_{ij} + v_{ij} \forall j \in [0, d)$$

The R_1 and R_2 parameters make the modification stochastic, they introduce perturbations in the calculation. These changes have the benefit that they can break non optimal behavior resulting from the deterministic calculation. If there is a suboptimal best value (local or global) that leads to a too strong attraction and thus forcing premature convergence, we can with a certain probability escape from such a value by perturbing the velocity calculation. The C constants determine how strong the effect is of respectively the local best and the global best. This reflects the balance between exploration and exploitation respectively, where a particle is influenced more by its own information or that of the swarm. After all particles are updated, the new global best is recorded for the next iteration.

Selection An interesting difference with other algorithms is that the position is always updated, whether it improves the fitness or not. The local best records the best known position, but the particle is allowed to visit positions with lower fitness values. This allows it to escape local optima. The global best is obviously only updated with an improved value. The comparison is strict, meaning that only a better value can become the new global best. This may not seem significant, but allowing equality updates can actually benefit an optimization process. To see this it helps to view the fitness domain as a landscape with troughs, valleys and heights. Allowing for equality updates allows the global best to move despite no apparent improvement in fitness. In this analogy we represent a (sub) optimal value with low points in the landscape. This allows it to cross areas where a zero gradient is observed, for example between two low points where the lower one is a local optima.

Cost With a population size of n, the algorithm requires n fitness evaluations per iteration. The computational cost of updating the velocity and position is small given the evaluation cost of an expression tree over several datapoints. However, it is linear in d, the number of dimensions. If the tree increases in size, the number of constants can increase in the worst case exponentially. On average due to the construction algorithm we expect that half the leaves in the tree are constants. Given our discussion of the constant folding algorithm we know that a full binary tree is unlikely, so the number of constant nodes is equally unlikely to increase exponentially, but will nonetheless scale poorly. In our implementation we will halt the algorithm if it cannot improve the global best after k/2 consecutive iterations, where k is the maximum number of iterations it can execute. The initialization stage adds another n evaluations, in addition to n per iteration. The total cost in fitness evaluations is therefore n(k+1), resulting in a worst case evaluation complexity of O(nk).

Configuration This overview gives the values of each parameter used in CSRM's PSO implementation.

- C_1 = 2
- C_2 = 2 : Setting both to 2 is recommended as the most generic approach [14].
- $w_i = \frac{1+r}{2}$ with r random in [0,1] : In early implementation the inertia weight was kept constant [6] There are a large number of strategies for an inertia weight. Dynamically decreasing inertia may improve convergence significantly. We opt for a random inertia weight as it has been shown [1] to lead to faster convergence. Since our use case requires fast convergence on very limited iterations, this strategy is clearly favored.
- R_1, R_2 r with r random in [0,1]
- population = 50 : PSO is not sensitive to populations larger than this value, providing a robust default value. [15]

CSRM's optimizer does not set constraints on the domain of each constant, it is different for each problem instance. Finding the domain of a constant in the expression tree

requires a domain analysis of the expression tree. With features in the tree involved, finding the exact domain is infeasible, given that some of the datapoints are unknown. It is therefore possible that a particle obtains values outside the valid domain of one or more constants, resulting in an invalid expression tree. This will result in the particle temporarily no longer contributing to the search process.

6.2.3 DE

Differential Evolution is a vector based optimization algorithm, or rather as the name implies, it operates by computing the difference between particles.

Algorithm The algorithm has a population of n vectors, similar to the other algorithms it holds a linear set of values to optimize, one per dimension.

Initialization Similar to our approach in initialization PSO, we perturb a known (sub) optimal solution. A vector stores its current value, and the best value.

Modification Each iteration the algorithm processes all vectors. For each vector $\vec{v}$, three distinct randomly selected vectors are selected. From these 3 vectors a new 'mutated' vector is obtained:

$$\vec{v} = \vec{w} + F(\vec{y} - \vec{z})$$

With $\vec{w}$,$\vec{y}$,$\vec{z}$ randomly chosen and not equal to $\vec{v}$.
The selection occurs with replacement. Several selection schemes exists, and the size of the selection is equally configurable. From this step the algorithm lends its name. The F factor influences the effect of the difference. Then we apply a crossover operation, using vectors x and v and probability parameter Cr. We select a random index j with $j \in [0, |x|)$. We then create a new vector u:

$$u_i = k_i \forall i \in [0, |x|$$

and k_i equal to

$$\begin{cases} v_i & i = j \vee r < Cr \\ x_i & i \neq j \wedge r \geq Cr \end{cases}$$

This is binomial crossover, another frequently used selection operation is exponential crossover.

Selection For a given selected vector $\vec{v}$ and created vector $\vec{u}$ we now test if $\vec{v}$ is a better candidate than $\vec{u}$, in other words has a lower or equal fitness value. Note the distinction here with PSO, the equality test allows DE vectors to cross areas without a gradient. If f($\vec{u}$) $\leq$ $\vec{v}$ the vector is replaced with $\vec{u}$. The global and local best are updated as well. A difference with PSO is that a PSO particle changes regardless of fitness value, whereas in DE the modification is only committed if a better or equal fitness value is obtained. The first approach allows an optimization algorithm to break

free from local optima. DE uses the random selection of other vectors to create a similar effect. If we use the landscape analogy, as long as at least one DE vector is outside a depression in the landscape, but all the others are converging to the suboptimal minimum, DE has a probability to escape a local optima. Unlike PSO DE (in our configuration) does not use the global best in its calculations, sharing of information is completely distributed over the vectors.

Cost For each vector 3 other vectors are used, or restated we create 2 new vectors. Similar to PSO these calculations have a complexity linear in d, the dimensionality of the problem. The fitness function is called once per iteration per vector. Compared to PSO we therefore have the exact same evaluation complexity of O(nk).

Configuration CSRM uses a DE/rand/2/bin configuration. The DE/x/y/z notation reflects its main configuration, where x is the vector perturbed, y is the vectors used in the differential step and z is the crossover operation (binomial). This configuration is referenced [26] as one of the most competitive for multimodal problems with good convergence to the global optimum. Since we start from a probable local optimum the choice for a random vector instead of the global best vector also helps avoid premature convergence. This overview gives the values of each parameter used in CSRM's DE implementation.

- F = 0.6 : F should be in [0.4, 1] Large F values favor exploration, whereas small F values favor exploitation. The value of 0.6 is reported as good starting value. In our problem domain we already have a (sub) optimal solution which we wish to improve, so the risk of premature convergence is present, hence the small bias for exploration.
- Cr = 0.1 : The Cr values should be in [0,0.2] for separable functions, and [0.9, 1] for non separable functions. We cannot assume dependency between the constants, and therefore use a value of 0.1. This results in DE focussing along the axes of each dimension in its search trajectory.

Compared to PSO DE has a low parameter count, optimal values for these parameters can be found in literature [4]. The population size should be t * d with t in [2, 10]. Since we do not know d in advance, and to keep the comparison fair we set the population at 50, allowing for optimal values for up to 25 dimensions (constants). While DE has a small set of parameters, their effect is still quite pronounced. There exists implementations of DE that use self adapting parameters, but this is beyond our scope. It should be noted that CSRM's optimizer has a very small optimization budget (in evaluation cost) and each new problem has potentially new characteristics. We therefore chose for the most robust values and configuration.

7 Experiments

7.1 Reproducability

All benchmarks were performed on a Intel Xeon E5 2697 processor with 64GB Ram, with Ubuntu 16.04 LTS, kernel 4.4.0 as operating system. CSRM is implemented using Python3, the test system uses Python 3.5. The experiments use a fixed seed in order to guarantee determinism. Where relevant the configuration is given. An open source repository holds the project's source code, benchmark scripts, analysis code and plots. The project dependencies are minimal making the project portable across any system that has a working Python3 implementation and pip as an installation manager. In order to run distributed the project requires an MPI implementation, which is available for most platforms.

7.2 Benchmark problems

Recent work on the convergence of GP-based SR [16, 18] featured a set of benchmark problems that pose convergence problems for SR implementations. We reuse these problems in our work in order to study convergence of CSRM's implementation.

7.2.1 Problems

These problems use at most five features. CSRM does not know which features are used, making the problem harder. In other words it assumes each problem is a function of 5 features which may or may not influence the expected outcome. This is an extra test in robustness for the algorithm, while also testing the algorithm's capability as a classifier. These problems are introduced by [18].

$$1.57 + (24.3 * x_3)$$

$$0.23 + 14.2 * \frac{x_3 + x_1}{3.0 * x_4}$$

$$-5.41 + 4.9 * (\frac{x_3 - x_0 + \frac{x_1}{x_4}}{3 * x_4})$$

$$-2.3 + 0.13 * \sin(x_2)$$

$$3.0 + (2.13 * ln(x_4))$$

$$1.3 + 0.13 * \sqrt{x_0}$$

$$213.80940889 - 213.80940889 * e^{-0.54723748542 * x_0}$$

$$6.87 + 11 * \sqrt{7.23 * x_0 * x_3 * x_4}$$

$$\frac{\sqrt{x_0}}{\ln(x_1)} * \frac{e^{x_2}}{x_3^2}$$

$$0.81 + 24.3 * \frac{2.0 * x_1 + 3.0 * x_2^2}{4.0 * x_3^3 + 5.0 * x_4^4}$$

$$6.87 + 11 * \cos(7.23 * x_0^3)$$

$$2.0 - 2.1 * \cos(9.8 * x_0) * \sin(1.3 * x_4)$$

$$32 - 3.0 * \frac{\tan(x_0)}{\tan(x_1)} * \frac{tan(x_2)}{\tan(x_3)}$$

$$22 - 4.2 * ((\cos(x_0) - \tan(x_1)) * \frac{\tanh(x_2)}{\sin(x_3)}$$

$$12.0 - 6.0 * \frac{\tan(x_0)}{e^{x_1}} * (\ln(x_2) - \tan(x_3))$$

7.3 Operators

7.3.1 Cooling

The mutation operator introduces new information in the form of generated subtrees into the population. Mutation ensures exploration, but is an computationally expensive operator. In section 4.3.1 we have discussed the impact on complexity the issue of generating valid expressions has. When we apply mutation we only allow the mutated expression to survive into the next generation if it has a strictly better fitness value. We record the success rate of both mutation and crossover, and their respective gains in fitness. Using this information we discovered that the mutation success rate decreases as the algorithm converges. The change in fitness value introduced by mutation is significant, but for highly fit expressions this change can be to abrupt and lead to worse fitness. We would like to see a shift to exploitation instead of exploration later in the convergence process. While applying mutation without fitness gain will not affect the convergence rate, it will incur significant computational cost. Based on this reasoning we introduce a cooling schedule similar to that used in Simulated Annealing (SA). The schedule tries to predict if mutation is beneficial by a biased random process based on the current fitness of an expression and its generation. As long as the schedule is correct we will improve the runtime without affecting convergence. If the schedule is too strict we will slow down convergence. P_i is defined as the position of expression i in the population, which is ordered on ascending fitness. G_i is the generation for expression i. P and G are the population size and number of generations respectively. The decision whether or not to apply mutation to expression i is then given by :

$$q = \frac{g_i}{2g}$$

$$w = \frac{p_i 2}{p}$$

R, s are uniformly distributed random numbers in [0,1].

$$m = r < q \wedge s < w$$

If m is true, we apply mutation. By making this choice stochastic we introduce a measure of tolerance into the schedule. Unlike SA we use a linear combination and not an exponential distribution.

Configuration

- population : 20
- minimum depth : 4
- maximum depth : 10
- phases : 20
- generations per phase : 20
- datapoints : 20
- range : [1,5]
- features : 5
- archivesize : 20
- expressions to archive per phase : 4
- optimization strategy : none
- testproblem : 0

The results of applying the cooling schedule vary over the testproblems. We see no clear effect on the fitness value so we focus on the intended goal of the cooling schedule, namely reducing the frequency of mutation applications that do not result in a better fitness value. We would like to have some insight into this process as it unfolds over the generations. Using our statistics we can see exactly how the mutation operator behaves over time.

Results

7.3.2 Depth sensitive

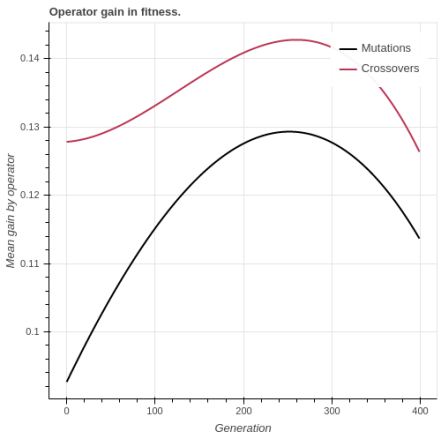
7.4 Constant Folding

7.4.1 Savings

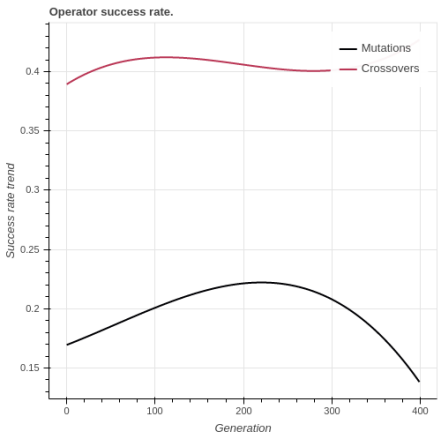
We will analyze the results of our constant subtree folding technique discussed in section 6.1.3.

Configuration

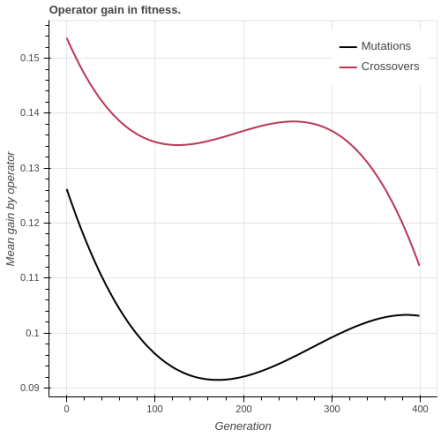
- population : 20
- minimum depth : 4
- maximum depth : 10



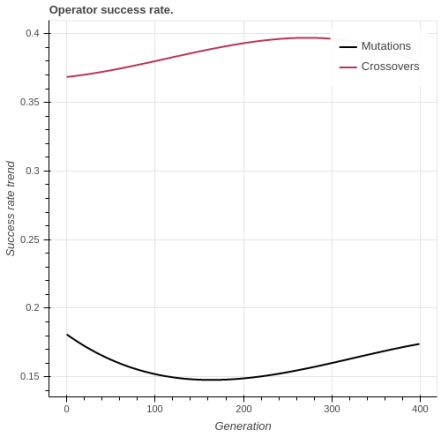
(a) Mutation gain with cooling schedule.



(b) Mutation success rate with cooling schedule.

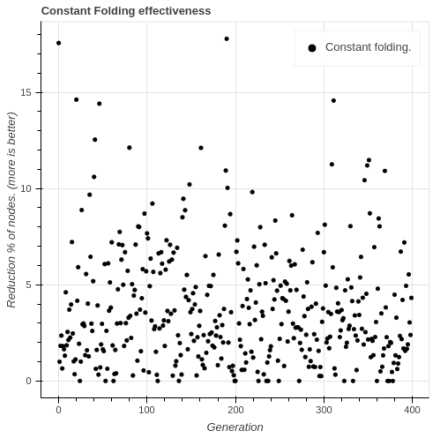


(c) Mutation gain without cooling schedule.

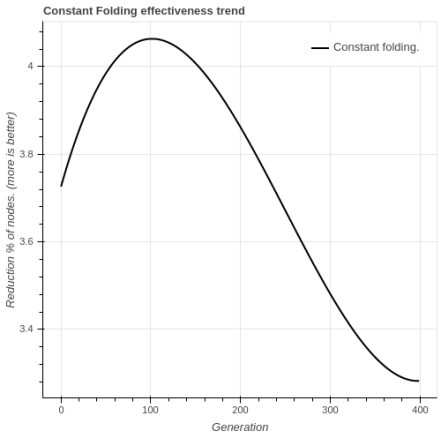


(d) Mutation success rate without cooling schedule.

Figure 15: Effect of cooling schedule on mutation success rate and gain.



(a) Folding savings over generations.



(b) Folding savings trend over generations (cubic fit).

Figure 16: Constant subtree folding savings over generations for testproblem 1.

- phases : 20
- generations per phase : 20
- datapoints : 20
- range : [1,5]
- features : 5
- archivesize : 20
- expressions to archive per phase : 4
- optimization strategy : none
- testproblem : 1

The results for constant folding are similar for all testproblems, we select a single problem in order to visualize the constant folding process over time instead of reporting the results at the end.

Discussion

Measure If we can collapse j subtrees holding k_j nodes in a tree with n nodes, we define the savings as

$$s = \frac{\sum_i = 0^j k_i - 1}{n} * 100$$

In other words, s is the percentage of nodes with which a tree is reduced in size. We calculate the mean for the entire generation of this value. In Figure 16 we see that

the savings have a high variance, but tend to decrease slightly. On average we expect between 1 and 5 % in savings. In addition to the savings these results give us an indication as to how the algorithm introduces constant subtrees. If we would not apply the savings we would see an incremental gain in constant subtrees. Even though the gains are relatively small the high number of generations would make this tendency problematic. Constant folding prevents this from occurring, but as we have discussed previously it can also hinder convergence by slowing down the search process for the 'right' constant. Even though a constant subtree can be represented by a single constant, it holds more information than that single constant alone and provides a kind of 'constant repository' that can be used by the operators to more quickly find fitter expressions. On the other hand if the constant subtrees grow so large as to dominate the tree the convergence can be compromised as the tree has no more place left for base functions using features. A delicate balance between the two is required here. In future work we could experiment with such a balance by delaying the constant folding until x generations have passed, instead of our current approach where we apply it after every generation.

7.5 Constant optimization

We look at the effect constant optimization using different algorithms has on different configurations of the tool. The measures used in the comparison are best fitness on training and test data, mean fitness on training and test data, and optimization cost.

7.5.1 Test problem

To verify our implementation for the optimizers we use a simple test problem and observe for each optimizer if it is able to optimize this instance to a known optimal value.

$$f(x_0, x_1, x_2) = 1 + x_1 * \sin(5 + x_2) * x_0 + (17 + \sin(233 + 9))$$

We give each optimizer a population of 50, 50 iterations and compare the results for 10 runs, displaying best value obtained, mean, and standard deviation of the fitness values compared to the known best value.

Best fitness In Figure 17 we see that DE outperforms PSO and ABC with several orders of magnitude. The best fitness value obtained was 2.22 e-16. As smaller but significant difference is present between PSO and ABC. This result is somewhat surprising given that fact that ABC is allowed to perform more evaluations in its configuration. From our previous discussion 6.2.2,6.2.3,6.2.1 we can conclude that for this test problem DE is clearly preferable as it obtains the best result at minimum cost. ABC has almost double the cost compared to PSO and DE, with PSO and DE having an equal cost in evaluations. The results on this testproblem do not necessarily mean that in the application of the three optimizers the results will be identical. Here we have a known optimal solution and want to observe how fast the optimizers converge to it. When we optimize evolved expressions we do not know what the optimal solution is. The

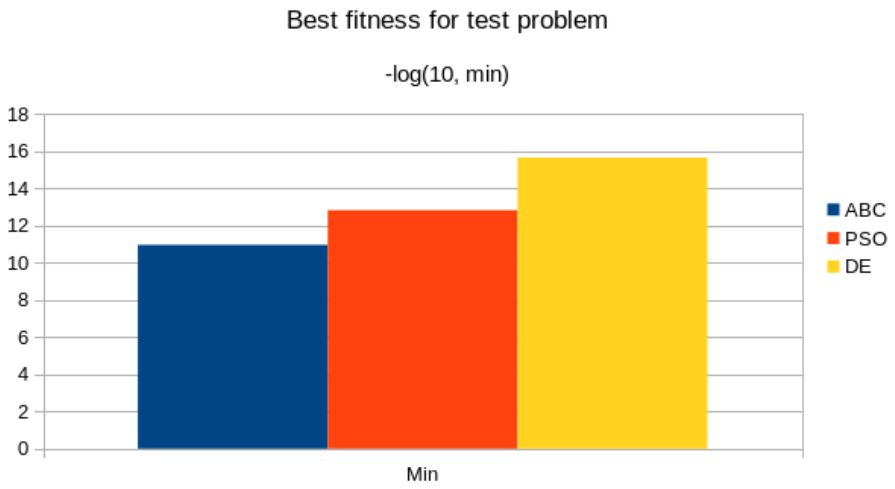


Figure 17: Logarithmic value of best fitness for each optimizer.

problem statement is different, and so the convergence behavior is likely to differ as well.

Distribution of fitness In Figures 18 and 19 we see that both the mean and standard deviation follow the same pattern as seen for the minimum fitness value with DE leading the others by several orders of magnitude. With all three distributions behaving similarly, this result provides a more solid foundation for our conclusions that for this problem DE is indeed the better optimizer.

7.5.2 Optimizer experiments setup

We test the 15 expressions with the following configuration:

- population : 20
- minimum depth : 4
- maximum depth : 10
- phases : (2, 5, 10)
- generations per phase : 20
- datapoints : 20
- range : [1,5]
- features : 5

Figure 18: Logarithmic scaled mean fitness for each optimizer.

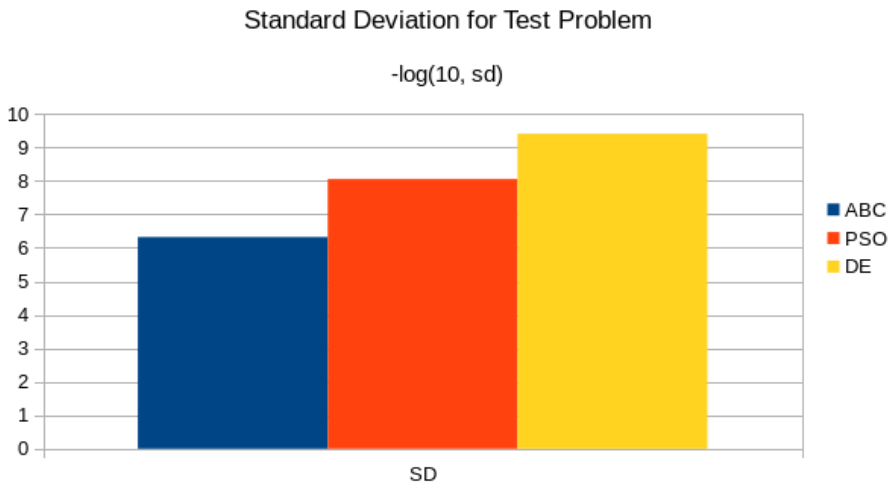


Figure 19: Logarithmic scaled standard deviation fitness for each optimizer.

- archivesize : 20
- expressions to archive per phase : 4
- optimization strategy : optimize expressions archived at end of phase

7.5.3 Measures

We compare the relative gain in fitness compared to not using an optimizer for all expressions. In other words, if m_n is a measure obtained by the algorithm without the optimizer, and m_a the same measure with the optimizer, we then define the relative gain as :

$$g_{ma} = \frac{m_n}{m_a}$$

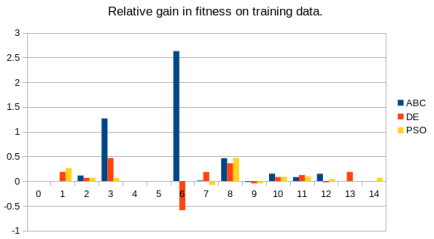
If m_a is zero, we use $-log_{10}(m_n)$ to represent the gain. If both are zero, the gain is obviously 1. A value of g > 1 indicates the ratio with which the optimizer improves the result. A g value < 1 indicates a regression. These 15 functions have wildly varying convergence behavior. In order to make sense of the data, we then apply a log scale :

$$g_{lma} = -\log_{10}(g_{ma})$$

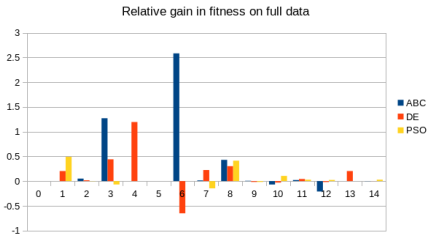
A value of $g_{lma} > 0$ indicates improvement, with the units transformed to orders of magnitude. A zero value indicates no improvement is registered, and negative values indicate regression. As measures we use the best fitness value on the training data, and the best on the full data set. We take the mean of the fitness of the 5 best expressions on training and the full data as well. This last measure gives us an indication on how the optimization process acts on the 'best' set of the population. Note that in our configuration, the 4 best expressions are always optimized.

7.5.4 2 Phases

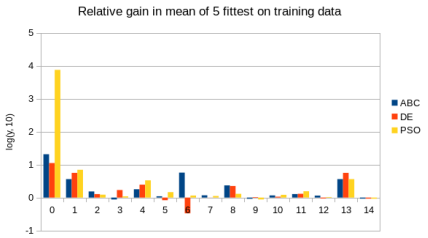
In Figure 20 we see the performance of the algorithms on training data. In Table 1 we see that for problems 0, 4, 5 there is no improvement possible (e.g. 0 zero fitness value), which explains the absence of any value in the figure. We see that for the training fitness data the improvements are significant, with ABC scoring an increase of 2.5 orders of magnitude for problem 6. For the other problems the increase is still large, especially given that our fitness function has a range of [0,1]. We also observe the significant regression for problem 6. This is likely caused due to overfitting. The algorithm in question (DE) optimizes the 4 best candidates of the last phase (1), but it is possible that these optimized expressions actually form a local optimum. By archiving these the convergence of the algorithm is hindered in the next phase. Note that DE allows equality updates, where expressions with the same fitness values are accepted as better. The same behavior occurs in a far less significant effect for expressions 7 and 9. A second explanation is our implementation of the population. The algorithm enforces distinct fitness values for all expressions. In an edge case it is possible that these optimized samples form a barrier, preventing other expressions from evolving past them. The optimized expressions in effect trap the rest of the population, which



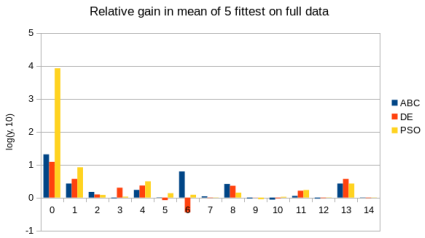
(a) Relative gain in best fitness of training data



(b) Relative gain in best fitness of full data



(c) Relative gain in mean fitness of training data



(d) Relative gain in mean fitness of full data

Figure 20: Relative gain of optimizer after 2 phases.

given our low generation count can explain this behavior. The mean fitness of the 5 best expressions shows significant improvements. Important to observe is the similarity between the two plots, the correlation between fitness values on training and full data is strong. This was a concern in the setup of the experiments. The optimizers could introduce overfitting on the training data. This risk is mitigated by the relatively low number of iterations each optimizer has been allocated. For the minimum fitness on the full data ABC outperforms the others. For the mean evaluation PSO is a good candidate. In this stage of the experiments, there is no single algorithm that is ideal for all problems. This once again confirms the NFL theorem [27].

Table 1: Relative Gain in minimum fitness on training data after 2 phases.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ABC	1.000e+00	1.000e+00	1.295e+00	1.848e+01	1.000e+00	1.000e+00	4.253e+02	1.030e+00	2.897e+00	9.615e-01	1.415e+00	1.207e+00	1.404e+00	1.000e+00	9.970e-01
DE	1.000e+00	1.536e+00	1.163e+00	2.923e+00	1.000e+00	1.000e+00	2.584e-01	1.526e+00	2.294e+00	8.972e-01	1.210e+00	1.327e+00	9.397e-01	1.536e+00	9.960e-01
PSO	1.000e+00	1.839e+00	1.169e+00	1.150e+00	1.000e+00	1.000e+00	1.000e+00	8.356e-01	2.947e+00	8.971e-01	1.226e+00	1.247e+00	1.096e+00	1.000e+00	1.172e+00

Table 2: Gain in minimum fitness on full data after 2 phases.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ABC	1.000e+00	1.000e+00	1.122e+00	1.865e+01	1.000e+00	1.000e+00	3.831e+02	1.034e+00	2.691e+00	1.016e+00	8.528e-01	1.051e+00	6.169e-01	1.000e+00	9.938e-01
DE	1.000e+00	1.597e+00	1.039e+00	2.757e+00	1.565e+01	1.000e+00	2.233e-01	1.674e+00	2.004e+00	9.617e-01	9.337e-01	1.103e+00	9.584e-01	1.597e+00	9.970e-01
PSO	1.000e+00	3.098e+00	9.915e-01	8.538e-01	1.000e+00	1.000e+00	1.000e+00	7.145e-01	2.586e+00	9.617e-01	1.274e+00	1.067e+00	1.055e+00	1.000e+00	1.069e+00

Table 3: Relative gain in mean fitness of 5 fittest expressions on training data after 2 phases.

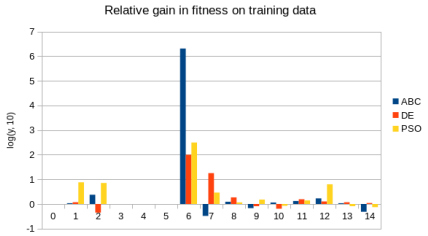
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ABC	1.335e+01	1.113e+00	4.824e+00	5.543e-01	3.479e+02	4.395e+02	7.059e-02	4.477e-01	1.215e+00	9.073e-01	2.214e+00	9.169e-01	1.564e+00	1.113e+00	4.064e-01
DE	3.620e+01	1.299e+00	1.115e+00	1.128e+00	1.204e-01	5.385e+08	3.202e+01	1.338e+00	1.835e+00	1.154e+00	1.268e+00	1.626e+00	1.246e+00	1.299e+00	7.247e-01
PSO	5.046e+02	8.017e+00	2.302e+00	4.220e-01	1.665e+01	2.237e+01	2.144e-01	6.974e-01	1.240e+00	9.647e-01	1.408e+00	1.347e+00	5.933e+00	7.152e-01	8.523e-01

Table 4: Relative gain in mean fitness of 5 fittest expressions on full data after 2 phases.

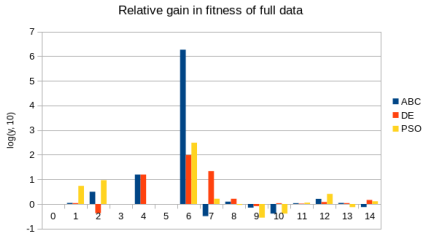
meanfullfitness															
algorithm	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ABC	1.382e+01	2.306e-01	5.950e+00	5.551e-01	3.264e+02	5.091e+02	7.308e-02	4.463e-01	1.188e+00	9.214e-01	4.665e-01	7.197e-01	1.463e+00	2.306e-01	8.096e-01
DE	3.244e+01	1.315e+00	1.068e+00	1.228e+00	1.153e-01	5.392e+08	3.234e+01	1.348e+00	1.685e+00	1.153e+00	1.193e+00	1.441e+00	1.210e+00	1.315e+00	1.199e+00
PSO	4.920e+02	6.794e+00	2.531e+00	4.117e-01	1.721e+01	9.994e+00	2.146e-01	6.486e-01	1.113e+00	7.060e-01	5.830e-01	1.324e+00	2.613e+00	8.126e-01	1.094e+00

7.5.5 5 Phases

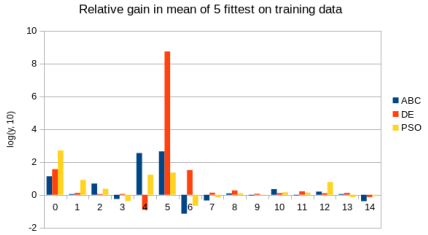
With 5 phases we see in Figure21 a more diverse effect. While ABC scores exceptionally good on problem 6, in sharp contrast with the 2 phase experiment, we see that PSO scores overall better for the training data. These results are logarithmic scaled, an improvement in fitness of factor 10 results in a value of 1 in the plots. When it comes to improving the mean of the best 5 expressions, PSO is a stable choice if we disregard the outlier values for problem 5. The correlation between training and full fitness scores is good for both measures. This demonstrates that the optimizer is not in this experiment introducing overfitting on the training data. The adverse effect of the optimizer on some test problems is still present. For the best fitness values on the full data DE is the better candidate. While ABC scores exceptionally high on problem 6, DE scores better overall. When we look at the mean there is no clear winner.



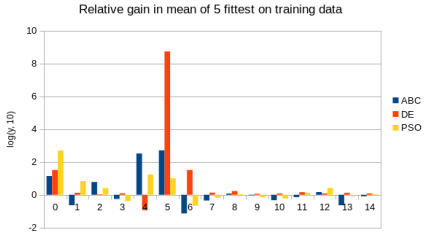
(a) Relative gain in best fitness of training data



(b) Relative gain in best fitness of full data



(c) Relative gain in mean fitness of training data



(d) Relative gain in mean fitness of full data

Figure 21: Relative gain of optimizer after 5 phases.

Table 5: Relative Gain in minimum fitness on training data after 5 phases.

algorithm	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ABC	1.000e+00	1.074e+00	2.371e+00	1.000e+00	1.000e+00	1.000e+00	2.050e+06	3.277e-01	1.230e+00	6.779e-01	1.146e+00	1.321e+00	1.690e+00	1.074e+00	4.837e-01
DE	1.000e+00	1.176e+00	4.459e-01	1.000e+00	1.000e+00	1.000e+00	9.987e+01	1.790e+01	1.853e+00	8.129e-01	6.428e-01	1.553e+00	1.266e+00	1.176e+00	1.095e+00
PSO	1.000e+00	7.591e+00	7.112e+00	0.000e+00	1.000e+00	1.000e+00	3.117e+02	2.904e+00	1.159e+00	1.506e+00	8.442e-01	1.396e+00	6.302e+00	8.146e-01	7.435e-01

Table 6: Gain in minimum fitness on full data after 5 phases.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ABC	1.000e+00	1.103e+00	3.145e+00	1.000e+00	1.565e+01	1.000e+00	1.846e+06	3.220e-01	1.243e+00	7.106e-01	4.082e-01	1.080e+00	1.613e+00	1.103e+00	7.483e-01
DE	1.000e+00	1.087e+00	4.240e-01	1.000e+00	1.565e+01	1.000e+00	9.756e+01	2.158e+01	1.629e+00	8.213e-01	1.079e+00	1.040e+00	1.202e+00	1.087e+00	1.454e+00
PSO	1.000e+00	5.426e+00	9.234e+00	0.000e+00	1.000e+00	1.000e+00	3.070e+02	1.634e+00	9.216e-01	2.786e-01	4.071e-01	1.131e+00	2.563e+00	7.444e-01	1.288e+00

Table 7: Relative gain in mean fitness of 5 fittest expressions on training data after 5 phases.

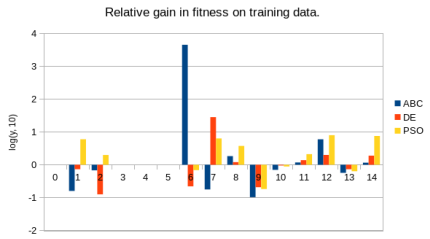
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ABC	2.064e+01	3.632e+00	1.536e+00	8.774e-01	1.788e+00	1.096e+00	5.726e+00	1.173e+00	2.348e+00	9.209e-01	1.163e+00	1.280e+00	1.151e+00	3.632e+00	9.300e-01
DE	1.117e+01	5.624e+00	1.277e+00	1.696e+00	2.468e+00	8.307e-01	3.333e-01	9.953e-01	2.249e+00	1.038e+00	1.067e+00	1.305e+00	9.416e-01	5.624e+00	9.302e-01
PSO	7.494e+03	6.977e+00	1.225e+00	1.091e+00	3.335e+00	1.461e+00	1.154e+00	1.118e+00	1.300e+00	8.794e-01	1.212e+00	1.559e+00	1.041e+00	3.632e+00	9.242e-01

Table 8: Relative gain in mean fitness of 5 fittest expressions on full data after 5 phases.

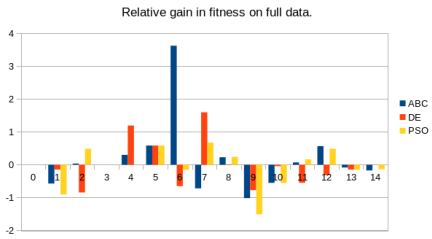
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ABC	2.048e+01	2.672e+00	1.487e+00	9.516e-01	1.709e+00	1.021e+00	6.217e+00	1.093e+00	2.589e+00	9.346e-01	8.686e-01	1.126e+00	9.308e-01	2.672e+00	9.816e-01
DE	1.213e+01	3.687e+00	1.244e+00	1.997e+00	2.324e+00	8.352e-01	3.572e-01	9.731e-01	2.288e+00	1.009e+00	9.496e-01	1.616e+00	9.641e-01	3.687e+00	9.668e-01
PSO	8.382e+03	8.286e+00	1.204e+00	1.071e+00	3.111e+00	1.362e+00	1.219e+00	9.817e-01	1.411e+00	8.939e-01	1.059e+00	1.698e+00	1.018e+00	2.672e+00	9.840e-01

7.5.6 10 Phases

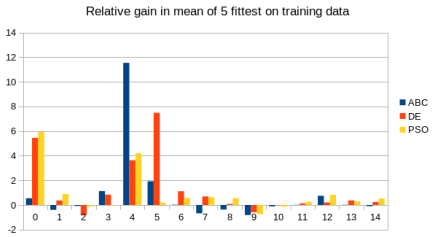
If we observe the convergence after 10 phases we see a more pronounced effect. In Figure 22 we see that for several problems the optimizers are no longer improving w.r.t the unoptimized algorithm. This only holds for the best values, for the mean values the improvements are still significant. It becomes clear that the optimizer can force the algorithm into a local optimum from which it becomes hard to escape. The correlation between fitness results on the training data and full data is starting to weaken as well, in comparison to the experiments with 2 and 5 phases. If we look at the fitness values for the full data DE is the more stable of the three algorithms. When it regresses its losses are smaller than the others, while its gains are strongest on the most problems. For the mean fitness of the full data a similar argument can be made, with the exception of problem 2 where DE fails severely. Another aspect is that after 100 generations the fitness values are extremely small, in the order of 1e-15. We measure the relative gain with respect to the algorithm without an optimizer, but as the fitness values decrease rounding errors start to influence the calculations more and more. The fitness values are approaching the floating point epsilon values. For our implementation epsilon is set at 2.22 e-16. For problem 0, a minimum fitness value of 0 is found after 2 phases. For others far more iterations are needed. We need to make a trade-off in order to be able to compare all 15 problems. Giving each problem an equal budget in iterations is the more fair approach. Another approach is implementing a stop condition that halts within a certain distance of a desired fitness treshold, but this approach is fraught with issues. There is no guarantee exactly how many iterations are needed. This approach requires knowing the problem 'hardness' in advance, but by the very definition of our problem statement we do not know how hard our problem is. We do not know the optimal value, if there is a singular optimal value. In general the topology of the search space SR tries to traverse is not known. A practitioner with a real world problem faces the same issues. A more robust approach is stating in advance how much resources the algorithm can use in its search, and terminate if that budget is exhausted. The exact definition of resource is nuanced. We can use time, but this depends to a large extent on the implementation. A more solid measure is the number of fitness evaluations. Even this is not a constant measure, not all evaluations are equal in computational complexity.



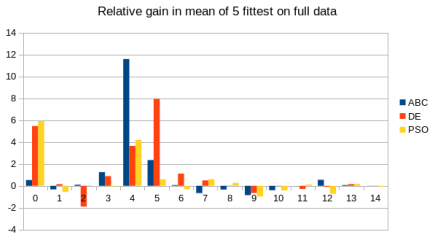
(a) Relative gain in best fitness of training data



(b) Relative gain in best fitness of full data



(c) Relative gain in mean fitness of training data



(d) Relative gain in mean fitness of full data

Figure 22: Relative gain of optimizer after 10 phases.

Table 9: Relative Gain in minimum fitness on training data after 10 phases.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ABC	1.000e+00	1.591e-01	6.759e-01	1.000e+00	1.000e+00	1.000e+00	4.508e+03	1.771e-01	1.831e+00	1.029e-01	6.983e-01	1.176e+00	5.945e+00	5.674e-01	1.158e+00
DE	1.000e+00	7.304e-01	1.254e-01	1.000e+00	1.000e+00	1.000e+00	2.197e-01	2.816e+01	1.201e+00	2.068e-01	9.480e-01	1.377e+00	1.989e+00	7.304e-01	1.891e+00
PSO	1.000e+00	5.976e+00	1.980e+00	1.000e+00	1.000e+00	1.000e+00	6.857e-01	6.335e+00	3.739e+00	1.822e-01	8.852e-01	2.105e+00	7.945e+00	6.334e-01	7.526e+00

Table 10: Gain in minimum fitness on full data after 10 phases.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ABC	1.000e+00	2.679e-01	1.087e+00	1.000e+00	2.000e+00	3.846e+00	4.224e+03	1.908e-01	1.694e+00	9.627e-02	2.807e-01	1.178e+00	3.699e+00	8.255e-01	6.709e-01
DE	1.000e+00	7.126e-01	1.432e-01	1.000e+00	1.565e+01	3.846e+00	2.232e-01	3.965e+01	1.031e+00	1.687e-01	9.052e-01	2.827e-01	4.813e-01	7.126e-01	1.005e+00
PSO	1.000e+00	1.246e-01	3.083e+00	1.000e+00	1.000e+00	3.846e+00	7.019e-01	4.738e+00	1.736e+00	3.091e-02	2.797e-01	1.459e+00	3.081e+00	7.083e-01	7.305e-01

Table 11: Relative gain in mean fitness of 5 fittest expressions on training data after 10 phases.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ABC	3.530e+00	4.032e-01	8.301e-01	1.346e+01	3.503e+11	8.400e+01	1.061e+00	2.112e-01	4.251e-01	1.531e-01	7.788e-01	1.034e+00	5.595e+00	1.041e+00	7.932e-01
DE	2.813e+05	2.333e+00	1.492e-01	6.914e+00	4.242e+03	3.098e+07	1.305e+01	4.957e+00	1.210e+00	2.547e-01	9.384e-01	1.339e+00	1.614e+00	2.333e+00	1.742e+00
PSO	8.346e+05	7.871e+00	7.495e-01	9.256e-01	1.573e+04	1.539e+00	3.743e+00	4.297e+00	3.560e+00	1.781e-01	7.192e-01	1.838e+00	6.612e+00	1.991e+00	3.372e+00

Table 12: Relative gain in mean fitness of 5 fittest expressions on full data after 10 phases.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ABC	3.389e+00	4.674e-01	1.263e+00	1.779e+01	3.846e+11	2.257e+02	1.163e+00	2.171e-01	4.563e-01	1.408e-01	3.961e-01	1.020e+00	3.568e+00	1.170e+00	1.027e+00
DE	2.901e+05	1.426e+00	1.270e-02	7.677e+00	4.445e+03	8.874e+07	1.310e+01	3.143e+00	1.057e+00	2.200e-01	9.597e-01	5.134e-01	8.172e-01	1.426e+00	1.039e+00
PSO	8.487e+05	2.720e-01	1.113e+00	8.726e-01	1.601e+04	3.942e+00	4.913e-01	4.000e+00	1.855e+00	1.070e-01	3.816e-01	1.327e+00	1.873e-01	1.466e+00	8.262e-01

7.5.7 Cost

The cost in terms of evaluations is linear in the number of phases. For each phase, 4 expressions are optimized with O(nk) complexity. In our configuration n=50, k=50. The SR algorithm itself requires O(mg) evaluations per phase, where m is the population (20) and g the generations per phase (20). The question whether or not the cost of the optimizer is worth the gain in fitness is in the end one for the practitioner to answer. There is no guarantee that improvement will take place, but from the results on the above hard problems we can still expect significant improvements up to several orders of magnitude.

7.6 Distributed

We now apply our tool to the testproblems in a distributed setup.

7.6.1 Experiment setup

- population : 20
- minimum depth : 4
- maximum depth : 8
- phases : 20
- generations per phase : 20
- datapoints : 20
- range : [1,5]
- features : 5
- archivesize : 20
- expressions to archive per phase : 4
- optimization strategy : none
- communication size m : 2, 4
- topology : Tree, Random, Grid
- spreadpolicy : distribute
- processes n : 25

Discussion From section 5.5 we know that simply using m for all topologies will lead to unintended communication patterns. We test Tree and Random with m = 2, and Grid with m = 4. This results in the following number of messages per link :

- Grid : 1
- Tree : 1
- Random : 2

The total number of messages sent per phase is then :

- Grid : 4 * n = 200
- Tree : 1 * n = 25
- Random : 2 * n = 50

The value of m = 2 for Tree and m = 4 for Grid follows from our discussion in 5.5. The Random topology in this configuration has a single outgoing link per process, resulting in 2 messages per link. This configuration forms a balance between the different strategies. For 25 processes the grid topology is a simple square of 5x5. The tree topology with 25 processes is a non full binary tree with depth 4. The random topology is highly dependent on the seed. In Figure 23 we see that in this instance the topology has become a disconnected graph of 3 cycles. This highlights the risk of using a random topology, without extra constraints the characteristics of the communication pattern are unknown and can be undesirable.

7.6.2 Measures

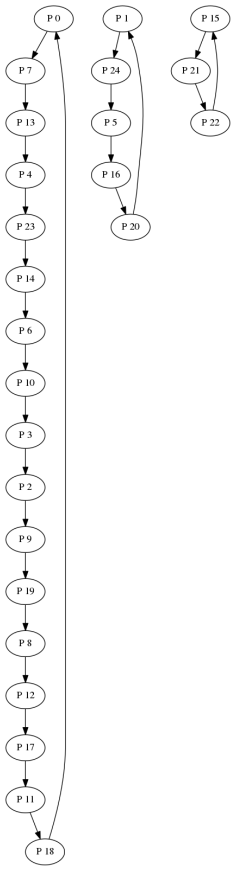
When the experiment ends the best 20 expressions from all processes are collected and scored. We measure in best fitness on the training data, and best fitness on the validated data. As we have seen in there is a subtle difference here compared to the sequential approach. While each process has a subset of the training and validation data, in the end we score against the full dataset. Finally we record the mean fitness values of the best 5 expressions, both on the training and validation data. These are the same measures as used by the optimizer experiment. The mean is restricted to the upper quarter of the population specifically to measure how the best expressions are distributed. This measure records the convergence more accurately as the fittest expressions drive the convergence rate.

Calculation The fitness values fluctuate strongly between the test problems and even between topologies. We apply a negative logarithmic scale :

$$f_t = -\log_{10}(f)$$

where f is either the best fitness value or the mean. Then we scale the results relative to the values obtained for the tree topology in order to measure relative gain or loss in orders of magnitude.

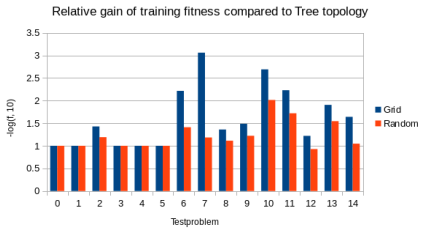
$$v_t = \frac{f_t}{f_{tree}}$$



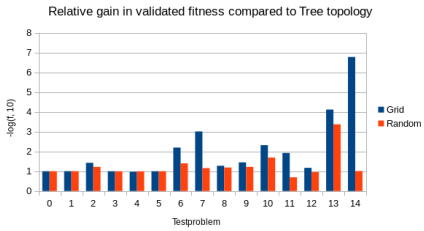
(a) Random topology for 25 processes with one outgoing link per process.

(b) Tree topology for 25 processes.

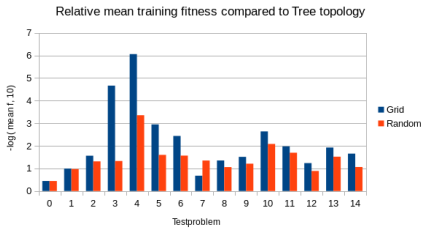
Figure 23: Tree and random topologies used in the experiment.



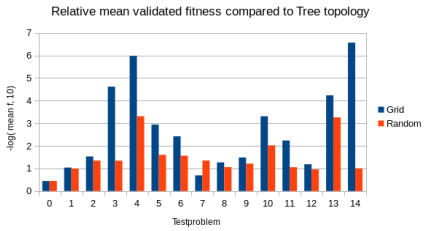
(a) Relative gain in best fitness of training data



(b) Relative gain in best fitness on full data.



(c) Relative gain in mean fitness on training data.



(d) Relative gain in mean fitness on full data.

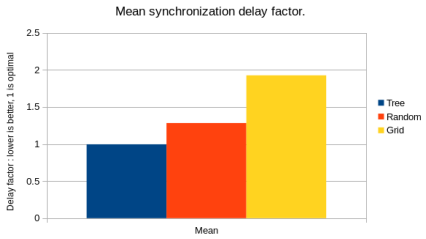
Figure 24: Convergence differences between topologies.

We will compare each topology to the tree topology to make an ordered comparison easier.

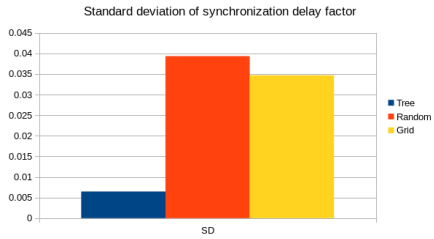
7.6.3 Results

Convergence In Figure 24 we see how the topology affects convergence. The first observation we make is that the first 5 testproblems, with exception of the second, all have identical values for best fitness on training and validation data. The processes converged to zero fitness values for these problems, hence the identical results. The training fitness results in Figure 24a indicate that the grid and random topology have superior convergence characteristics compared to the tree topology, with grid outperforming random on several testproblems. When we look at the fitness values on the validation data in Figure 24b we see a more nuanced result. The grid topology is overall still the best choice, but the random topology has worse results for problems 11 and 12. Next we look at the mean fitness values on training and validation data. Interestingly enough for problem 0 both random and grid score far worse than the tree topology. Overall the grid topology is still better for most problems, with the exception of problem 7. Note the similarity in pattern in the results here between training and validation data indicating that the predictive capability of the results is still good. If overfitting would have taken place we would see a reverse pattern in the results for the validation data.

Overhead Given that the Tree, Random and Grid topologies have different synchronization and memory constraints we now look at their real life overhead and discuss



(a) Mean synchronization delay factor.



(b) Standard deviation of synchronization delay factor.

Figure 25: Synchronization overhead introduced by topologies.

the results.

Synchronization From our discussion in 5.4 we know that cycles in the topology will lead to excessive synchronization and even serialization. We measure the mean execution time for testproblem 6. The convergence characteristics using the three topologies differed significantly making this a good testcase. The processes will communicate 25 times. If the runtime of a single phase is too long, the overhead of communication will become hard to measure. If it is too short the overhead dominates the entire runtime. This second case is one we should try to avoid, it will unfairly penalize topologies with cycles forcing them to serialize. The runtime of a phase is dependent on the generations, population and depth ranges of the expressions. Ideally we would like for a practitioner to choose these parameters based on the problem at hand and not constrained by synchronization considerations. We compare the three topologies and use the disconnected or 'none' topology as a reference point, which has zero synchronization overhead. This last topology has an ideal speedup of n, where n is the processcount, compared to a sequential process. From the synchronization overhead we can then derive the speedup each of the topologies is able to offer. In practice even the 'none' topology will have some synchronization overhead, as the root process has to collect all results from the other processes. In Figure 25 we see that the tree topology has a near zero delay introduced by the synchronization. This is due to the delay tolerance we have built in in our implementation as seen in Figure 8. The random topology has a mean delay factor of 1.3, the grid topology scores worst with a mean delay factor of nearly 2. This is easily translated in terms of speedup. A tree topology will have near linear speedup, a grid will have a speedup roughly half of that value and a random topology will have a speedup bracketed between those two. The standard deviation for the tree topology is significantly smaller indicating that a tree topology will have a far more predictable speedup.

Memory Memory overhead is hard to measure in a language with a garbage collector. We can estimate the overhead by calculating the needed memory in function of depth and topology used. Let the depth be constrained by $[d_i, d_m]$, with n processes, m communicationsize and a distribution spreading policy. If we let $d_a = \frac{d_i+d_m}{2}$ be the

average depth, then the memory requirements on average for each topology are then given by

- Tree : $d_a \frac{m}{2} n$
- Random : $d_a \frac{m}{n}$
- Grid : $d_a \frac{m}{4} 4n$

M $>= 4$ for a grid if we use distribute spreading policy. This leads us to an important observation. The Tree topology can communicate expressions with an average depth that is 2 times greater than the one used by the grid with the same memory usage. This factor is important, an increase in depth has an exponential effect on the complexity of the entire algorithm but also allows for more complex solutions. In addition an increased depth tolerates more bloat without losing accuracy.

7.7 Conclusion

7.7.1 Operator cooling schedule

7.7.2 Constant folding

7.7.3 Optimizers

The experiments with the optimizers highlight several issues. There are a wide number of strategies and parameters that influence the effect of the optimizer. We also see that optimizers can hinder the algorithm in its convergence. This is not a general conclusion, but dependent in part on our design choices and the trade-offs made. If we only optimize the final outcome of the algorithm it is obvious that no fitness regression is possible. Only when we apply optimization in the archiving stage are there subtle effects at play that allow for such edge cases. The cost of applying the optimizers is significant. In our implementation the cost is known beforehand, but the gain is not. This holds true in general for this GP SR algorithm. While we can empirically investigate the convergence of a number of problems, there is no known limit to this process. In general, when the optimizers are used the improvements made are far greater than the loss in edge cases. We have tested 3 distinct algorithms as optimizers in order to test which is best. Unfortunately no such algorithm exists. The NFL theorem [27] suggests as much. What we can see is that ABC and DE offer, for our problem set, the best results. Best in this context means the overall highest gain with the lowest losses at an equal cost to the other algorithms. The hardness of the SR problem indicates that, unfortunately, there are an infinite number of problem statements that will have different convergence characteristics. While hybridization of the GP algorithm with other algorithms is a viable strategy, it also substantially increases the number of parameters.

7.7.4 Distributed

We have seen that a tree topology is able to offer a near linear speedup at the cost of a lower convergence rate. The random and grid topologies on the other hand offer better convergence rates but will have a longer runtime in order to achieve those values. A

tree topology is able to operate on expressions with average depth twice that of the grid topology making it an attractive choice for practitioners. The random topology finds a balance between the characteristics of the tree and grid, but is non deterministic in its communication patterns. The resulting convergence and runtime will be influenced by each new random communication pattern, leading to uncertainty for the practitioner.

8 Use Case

In this section we will use our tool on a real world use case.

8.1 Problem statement

We want to use symbolic regression on the output of a simulator. The parameter space of the simulator is huge, and a single simulation instance is computationally expensive. We would like to obtain symbolic expressions relating parameters values to simulator output. These expressions can be used to gain insight into which parameters are correlated, which have a larger effect on output and which are irrelevant. Given the huge parameter space of the simulator a Design Of Experiment is constructed where the coverage of the parameter space is maximized while minimizing the number of combinations for all parameters. We then execute the simulator on each configuration. This is an embarrassingly parallel problem, where each configuration can be executed independent of the others. We then combine the output of all configurations and feed them into a tool where we can apply symbolic regression or another machine learning technique in order to extract a model that approximates the simulator. The problem with this approach is twofold. We have to wait until the simulator has completed all configurations and the SR tool executes on a large dataset. It has no known starting point so effectively performs a blind search in a huge search space. We can avoid both issues by using partial results from the simulator as input for the SR tool. The results from these partial samples ideally will provide a good starting point for the incrementally growing dataset. The last assumption only holds if the sequence of completed configurations is a good sample of the full dataset. We can enforce this sequence by ordering the configurations but this is non trivial. Configurations will not have the same computational load for all simulations. Consider for example the population parameter in a epidemiological simulator. A linear increase in runtime is at least expected if we increase this parameter. Even if we take this into account in our scheduling, the parallel execution of any number of tasks is never completely deterministic. The order of configurations is not only important to avoid a bias for the full design, which would lead to overfitting. The initialization problem 4.3.1 reappears here. If the initial partial set of configurations is biased the probability is quite high that the resulting solutions are invalid for the complete data set.

8.2 Design of Experiment

8.2.1 Experiment configuration

We construct a DOE with 3 parameters, 30 points in total.

- initial depth 3
- maximum depth 6
- p: population 20
- g: generations per phase 60
- f: phases 30
- archive 4 best per phase
- d: datapoints : 10, 20, 30

The total cost in fitness evaluations is then given by: 20*60*30*d. We compare 3 approaches. First we run the CSRM tool on the entire dataset. This is the classical approach, the tool is not seeded and so starts a blind search. In a real world setting this would mean waiting until the simulator has run all 30 configurations. Then we split the data into incremental sections. After 10 configurations have completed we start the tool on this dataset. The best 4 results are saved to disk, then we run the dataset with 20 configurations and use the results from the previous run as a seed. The overlap between the two datasets will mitigate the initialization problem. Finally we use the results of the 20-point dataset as seed for the 30 point run. The cost of running the 10 and 20 point runs to use as seed for the 30 point run is equal to the cost of the 30 point run. Our last approach is given the 30 point run double the amount of phases. This means that it has the same number of fitness evaluations as the 10-20-30 combination. We compare all three to see which gains are made and at what cost.

8.3 Results

8.3.1 Fitness improvement

In Figure 26 we see that the fitness is improved by using the best results of the previous run on a partial data set. We have deliberately split our data set to expose a risk here. If we run the tool with 20 datapoints, seeded by a run of 10 datapoints we see that the validated fitness actually decreases compared to a non seeded run. The ratio between new and known data is too large, leading to overfitting. If we seed the best results from the 20-point run into a 30 point configuration we see that both the training and validated fitness values significantly improve. The 30 point run with 60 phases has the same computational cost as the 10-20-30 runs combined, but has lower convergence. We see that convergence is slowing, with training fitness improving by a factor of 1.1, but validation fitness worsens. This is a typical example of overfitting. The combined 10-20-30 run increases validation fitness with a factor of 1.13.

8.3.2 Convergence behavior

Fitness distribution In Figures 272829 we see how the convergence process evolves over time. If we compare the fitness plot we clearly see that the seeded process has a different distribution compared with the unseeded runs. Around generation 1000 the

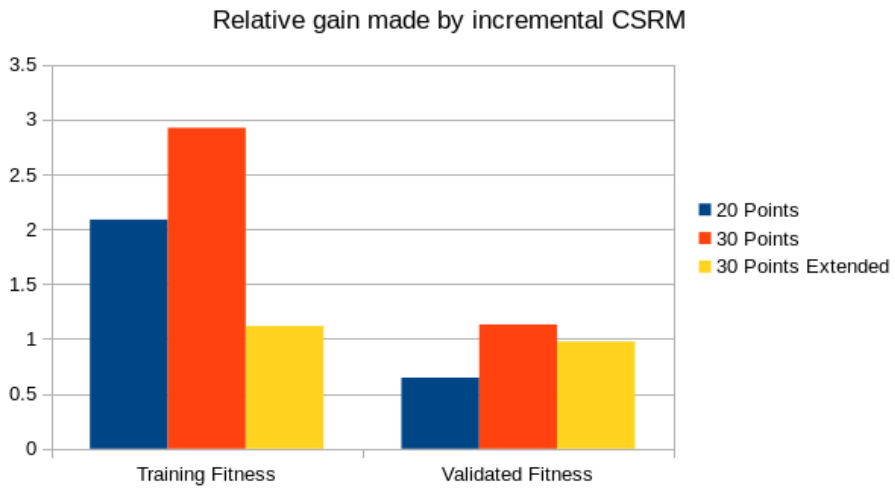


Figure 26: Incremental fitness gain in CSRM.

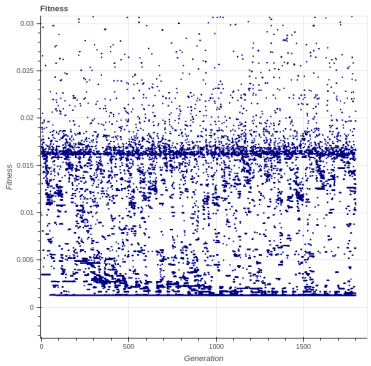
optimal fitness values have converged to their optimal. Doubling the phases has little effect in the fitness distribution.

Operator effectiveness In the plots we see operator success rate and operator gains visualized. The first is a simple counter, if an operator improves the fitness of an expression it is incremented. A trendline is fitted to this data by means of a cubic. This gives us an indication whether an operator is still effective, in particular it gives us the fraction of the population that is improved each generation by the operator. The second plot is the mean gain of an operator, it reflects how much the fitness of each generation is improved by an operator. The distinction is important, if fitness is improved by very small amounts the success rate will be high but the gain low. These statistics give a clear view of the GP process as it executes. Instead of relying only on the end result we can directly observe the effects of the operators. For our comparison we see that there is little difference in gain or success rate between the three runs.

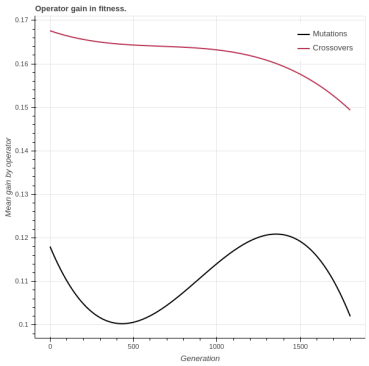
Constant folding The percentage of nodes saved is similar for all three runs. The plots show that constant subtrees are introduced at a constant rate and folded at the end of each generation. The savings plotted also give an indication of the potential increase that could take place if folding was not implemented.

8.4 Conclusion

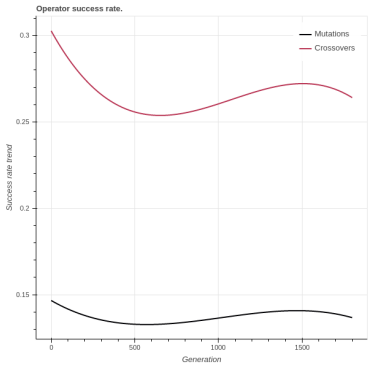
We have seen that incremental use of symbolic regression can, when applied judiciously, increase convergence compared to a blind search with the same computational cost. In addition to obtaining improved results we can integrate it with a simulator that produces output in parallel.



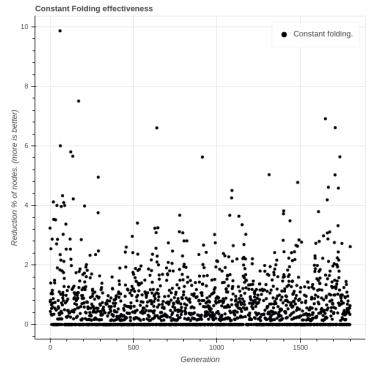
(a) Fitness.



(b) Operator gain.

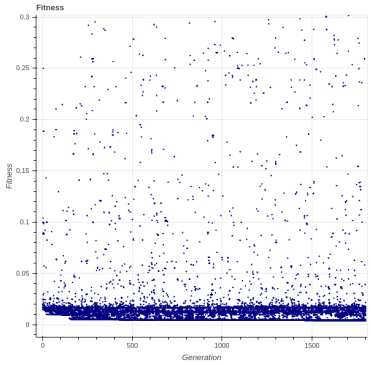


(c) Operator success rate.

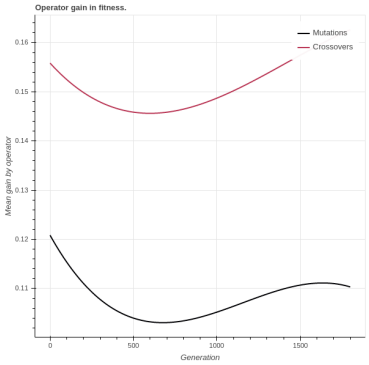


(d) Constant folding savings.

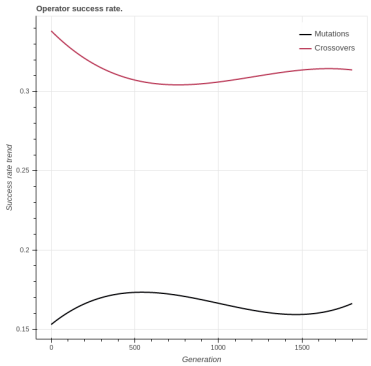
Figure 27: Convergence behavior of incremental symbolic regression with 10-20-30 split.



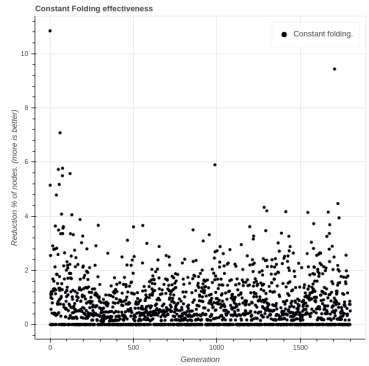
(a) Fitness.



(b) Operator gain.

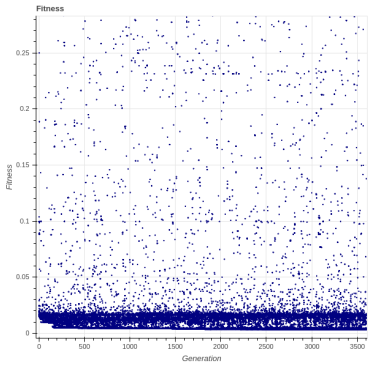


(c) Operator success rate.

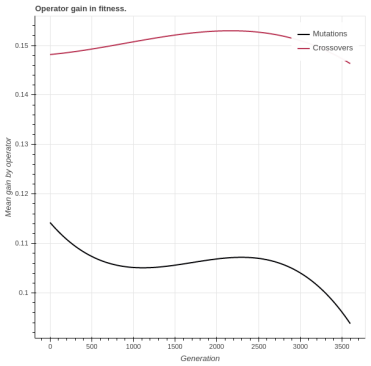


(d) Constant folding savings.

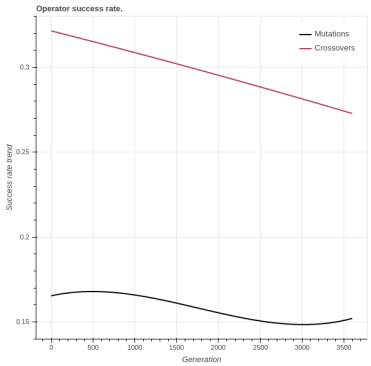
Figure 28: Convergence behavior of incremental symbolic regression without split.



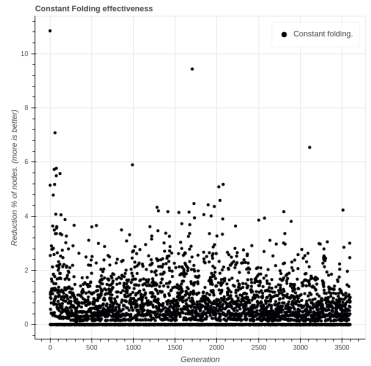
(a) Fitness.



(b) Operator gain.



(c) Operator success rate.



(d) Constant folding savings.

Figure 29: Convergence behavior of incremental symbolic regression with identical computational cost as 10-20-30 split.

9 Related Work

10 Conclusion

11 Future work

Extending constant optimization

Linear weight optimization The constant optimization step applied in our tool has been limited to simple constants in the expression. The tree representing the expression stores a hidden constant for each node that can act as linear weights. We can extend the constant optimization step to include these constants. The advantage is that the expression can be optimized to a greater extent. The disadvantage is the high increase in computation cost. Each constant represents a dimension for the constant optimizer. From our discussion we know that a high dimensionality has a serious impact on the complexity of the optimization step.

Extending folding Using the linear weights representation we could further simplify trees when applying constant folding. The following simple expression

$$f(x) = \sin(\pi \frac{\tan(x)}{2})$$

is represented using a tree with 7 nodes. If we extract $\frac{\pi}{2}$ as a linear weight for the $\tan$ node the tree is reduced to three nodes. The expression is invariant, but the representation is far more compact.

$$f(x) = \sin(\frac{\pi}{2} \tan(x))$$

Detecting and folding such cases is non trivial for more complex expressions.

Distributed set of optimizers We can replace GP with several other combinatorial optimization algorithms and compare convergence. From recent literature we know that ACO, GE and ABC have been used. Using our distributed architecture it would be possible to give each process a different optimization algorithm. This has two advantages. It allows for comparison of different optimization algorithms within the same framework. It would also make the SR tool more robust. Each optimization algorithm has its strengths and weaknesses. We know from the NFL theorem that no optimization algorithm is optimal for all optimization problem instances. A cooperative set of optimizations algorithms could offer an optimal solution for all problem instances by balancing the disadvantages and advantages of each algorithm.

Base functions In this work we have seen how large the impact is from invalid expression on the runtime of the algorithm. If we use a set of base functions where the domain is identical for each, for example the Chebyshev polynomials, and rescale our input set then we could largely avoid the initialization issue.

Policies This work can be extended by several policies. The spreading policies in the topology can be extended with random sampling, new trends in archiving can be applied to the algorithm and the mutation and crossover operators can be similarly extended.

Topologies The inverted tree topology is an interesting alternative to the original tree.

Hyperheuristics Our distributed SR algorithms has a large parameter space, most of which influence convergence. Their optimal values can be problem dependent, correlated to each other and are in general unknown. Optimizing these values requires a new optimization algorithm. Another alternative is a self optimizing variant that uses statistics collected at runtime to modify the parameters in order to find more optimal values.

Random distributions The choice of random distribution in a stochastic algorithm such as most metaheuristics influences the convergence greatly. From generating initial values to perturbing known solutions, selecting target for evolutionary operators, selecting communication partners and so on. The distribution used will have a definite effect on the exploration/exploitation balance in the algorithm. Recent work uses new distributions such as Levy [24] to improve convergence of metaheuristcs. This remains an interesting and open subproblem for symbolic regression.

List of Figures

List of Tables

References

[1] BANSAL, J. C., SINGH, P., SARASWAT, M., VERMA, A., JADON, S. S., AND ABRAHAM, A. Inertia weight strategies in particle swarm optimization. In *Nature and Biologically Inspired Computing (NaBIC), 2011 Third World Congress on* (2011), IEEE, pp. 633–640.

[2] BONYADI, M. R., AND MICHALEWICZ, Z. A locally convergent rotationally invariant particle swarm optimization algorithm. *Swarm Intelligence 8*, 3 (2014), 159–198.

[3] CLERC, M., AND KENNEDY, J. The particle swarm - explosion, stability, and convergence in a multidimensional complex space. *Trans. Evol. Comp 6*, 1 (Feb. 2002), 58–73.

[4] DAS, S., MULLICK, S. S., AND SUGANTHAN, P. Recent advances in differential evolution an updated survey. *Swarm and Evolutionary Computation 27* (2016), 1 – 30.

[5] DORIGO, M., BIRATTARI, M., AND STUTZLE, T. Ant colony optimization. *Comp. Intell. Mag. 1*, 4 (Nov. 2006), 28–39.

[6] EBERHART, R. C., AND SHI, Y. Comparing inertia weights and constriction factors in particle swarm optimization. In *Proc. of the 2000 Congress on Evolutionary Computation* (Piscataway, NJ, 2000), IEEE Service Center, pp. 84–88.

[7] EVETT, M., AND FERNANDEZ, T. Numeric mutation improves the discovery of numeric constants in genetic programming. In *in Genetic Programming. In* (1998), Morgan Kaufmann, pp. 66–71.

[8] GANSNER, E. R., AND NORTH, S. C. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE 30*, 11 (2000), 1203–1233.

[9] GARDNER, M.-A., GAGN, C., AND PARIZEAU, M. Controlling code growth by dynamically shaping the genotype size distribution. *Genetic Programming and Evolvable Machines 16*, 4 (2015), 455–498.

[10] JENKS, G., ET AL. SortedContainers: Sorted list, dictionary and set types for Python, 2014–. [Online; accessed ¡today¿].

[11] KARABOGA, D., AKAY, B., AND OZTURK, C. *Artificial Bee Colony (ABC) Optimization Algorithm for Training Feed-Forward Neural Networks*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 318–329.

[12] KARABOGA, D., OZTURK, C., KARABOGA, N., AND GORKEMLI, B. Artificial bee colony programming for symbolic regression. *Inf. Sci. 209* (Nov. 2012), 1–15.

[13] KENNEDY, J., AND EBERHART, R. Particle swarm optimization, 1995.

[14] KENNEDY, J., AND EBERHART, R. C. Particle swarm optimization. In *Proceedings of the 1995 IEEE International Conference on Neural Networks* (Perth, Australia, IEEE Service Center, Piscataway, NJ, 1995), vol. 4, pp. 1942–1948.

[15] KENNEDY, J. F., KENNEDY, J., EBERHART, R. C., AND SHI, Y. Swarm intelligence, 2001.

[16] KORNS, M. F. *Accuracy in Symbolic Regression*. Springer New York, New York, NY, 2011, pp. 129–151.

[17] KORNS, M. F. *Accuracy in Symbolic Regression*. Springer New York, New York, NY, 2011, pp. 129–151.

[18] KORNS, M. F. *A Baseline Symbolic Regression Algorithm*. Springer New York, New York, NY, 2013, pp. 117–137.

[19] KOZA, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[20] LI, Z., WANG, W., YAN, Y., AND LI, Z. Psabc: A hybrid algorithm based on particle swarm and artificial bee colony for high-dimensional optimization problems. *Expert Systems with Applications 42*, 22 (2015), 8881 – 8895.

[21] MCCONAGHY, T. FFX: Fast, Scalable, Deterministic Symbolic Regression Technology. In *Genetic Programming Theory and Practice IX*, R. Riolo, E. Vladislavleva, and J. H. Moore, Eds., Genetic and Evolutionary Computation. Springer New York, 2011, pp. 235–260. DOI: 10.1007/978-1-4614-1770-5_13.

[22] O'NEILL, M., AND BRABAZON, A. Grammatical differential evolution. In *Proceedings of the 2006 International Conference on Artificial Intelligence, ICAI 2006* (Las Vegas, Nevada, USA, June 26-29 2006), H. R. Arabnia, Ed., vol. 1, CSREA Press, pp. 231–236.

[23] POLI, R., AND GRAFF, M. There is a free lunch for hyper-heuristics, genetic programming and computer scientists. In *Proceedings of the 12th European Conference on Genetic Programming* (Berlin, Heidelberg, 2009), EuroGP '09, Springer-Verlag, pp. 195–207.

[24] RAJASEKHAR, A., ABRAHAM, A., AND PANT, M. Levy mutated Artificial Bee Colony algorithm for global optimization. In *2011 IEEE International Conference on Systems, Man, and Cybernetics (SMC)* (Oct. 2011), pp. 655–662.

[25] SPEARS, W. M., GREEN, D. T., AND SPEARS, D. F. Biases in particle swarm optimization. *Int. J. Swarm. Intell. Res. 1*, 2 (Apr. 2010), 34–57.

[26] STORN, R., AND PRICE, K. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization 11*, 4 (1997), 341–359.

[27] WOLPERT, D. H., AND MACREADY, W. G. No free lunch theorems for optimization. *Trans. Evol. Comp 1*, 1 (Apr. 1997), 67–82.