Convergence of distributed symbolic regression.

Ben Cardoen
ben.cardoen@student.uantwerpen.be

May 7, 2017

1 Abstract

Symbolic regression (SR) fits symbolic expression to a dataset of expected values. Amongst its advantages over other techniques are the ability for a practitioner to interpret the resulting expression, determine important features by their usage in the expression, and an insight into the behavior of the resulting model (e.g. continuity, derivation, extrema). SR combines a discrete combinatoric problem (combining base functions) with a continuous optimization problem (selecting and mutating constants). One of the main algorithms used in SR is genetic programming. The convergence characteristics of SR using GP are still an open issue. 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 new topology with the aim of finding a new balance between structured and random diffusion. 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 validate our work on several test problems, and a real world use case.

2 Introduction

3 Design

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

3.1 Algorithm

3.1.1 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.

3.1.3 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 3.8 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 3.3.1 we will cover the problem these functions solve.

Optimizer CSRM provides an interface to continuous optimizers. In section **??** 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.

3.1.4 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.

3.2 Fitness

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

3.1.2 Control flow

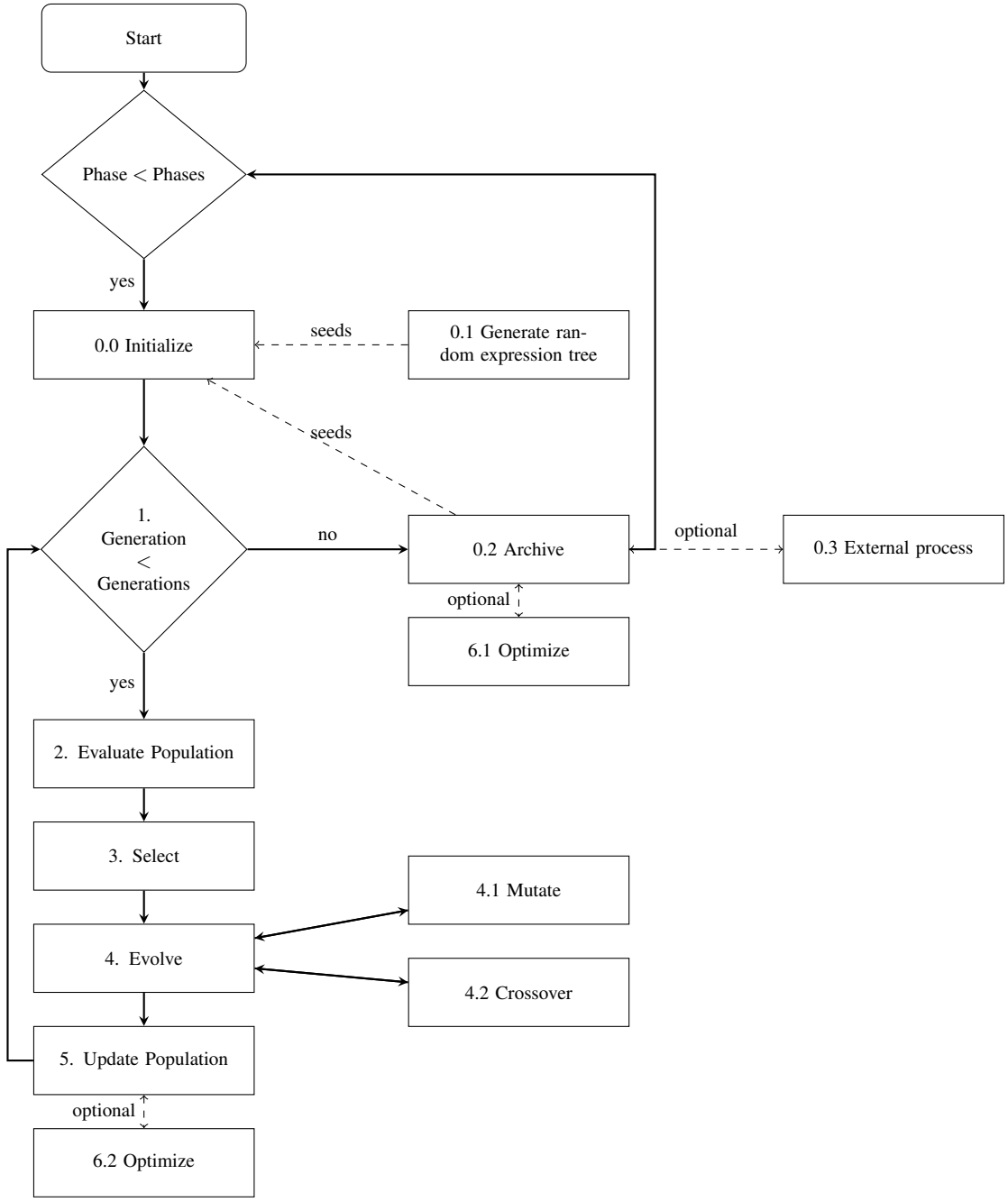


Figure 1: CSRM control flow.

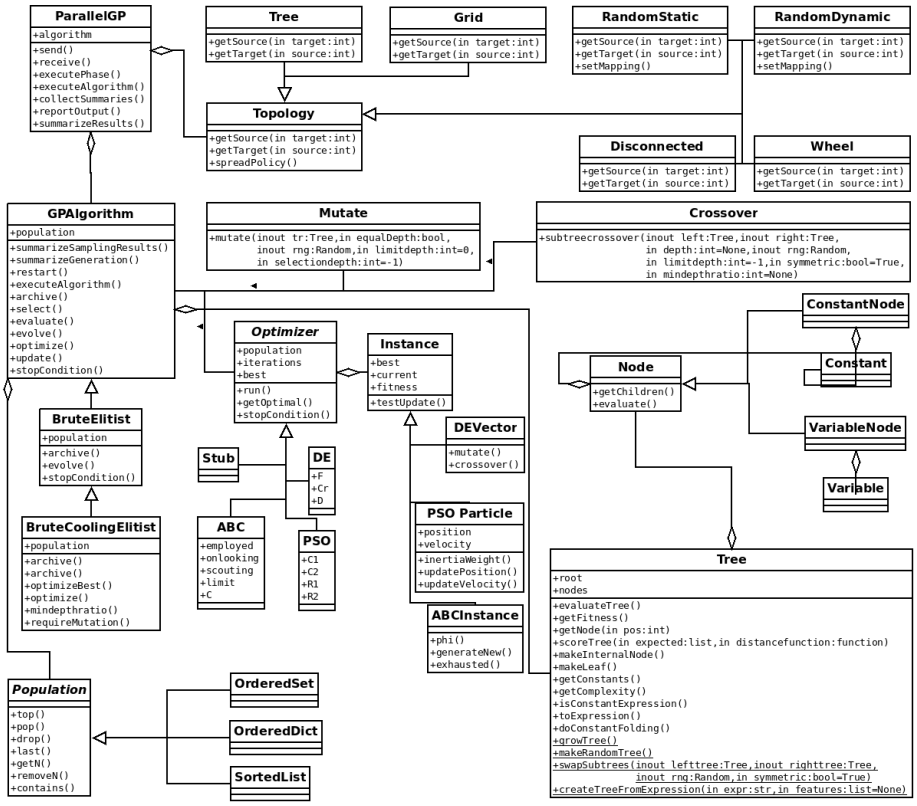


Figure 2: UML of CSRM.

3.2.1 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.
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'.

3.2.2 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 promote 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, consider a population of 5 where the 3 best samples all have fitness scores of 0.134. Our tool will aim to prevent retaining expressions that have identical fitness values. In contrast, in some metaheuristics [5] allowing replacement of solutions with identical fitness values (not duplication, but replacement), can actually help avoiding local minima.

3.2.3 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.

3.2.4 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.

3.3 Initialization

Initialization is done using the 'full' method [4]. 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.

3.3.1 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 [2], 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.

3.4 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.

3.5 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.

3.5.1 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 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 3.3.1 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.

3.5.2 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.

3.6 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.

3.7 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.

3.8 Representation and data structures

3.8.1 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.

3.8.2 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 [3] 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 3.2.2. 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.

3.9 Parameters

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

3.9.1 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 [2] 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 3.3.1 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.

3.9.2 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.

3.9.3 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.

3.9.4 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.

3.9.5 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.

3.10 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 :

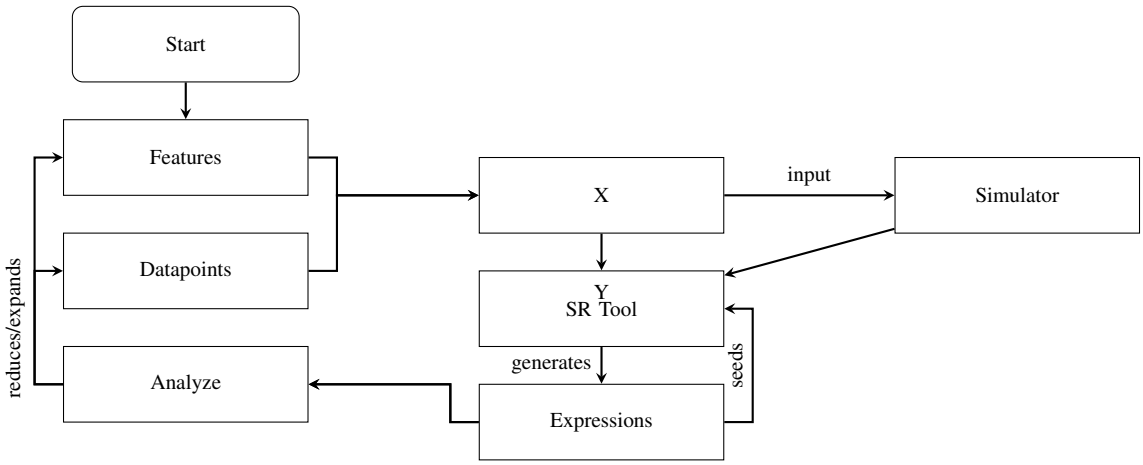


Figure 3: Incremental DOE using CSRM and a simulator.

- 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 **??**. The following diagram illustrates a DOE hypercube design using our tool and a simulator.

3.11 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 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 [1]. In Figure 4 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.

3.12 Conclusion

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

4 Experiments

5 Related Work

6 Conclusion

7 Future work

References

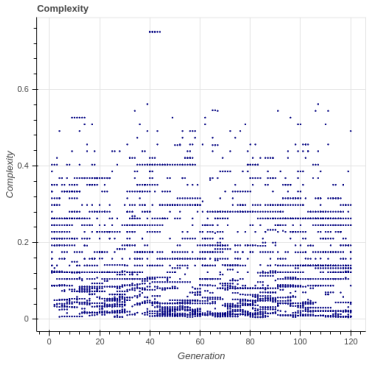
[1] GANSNER, E. R., AND NORTH, S. C. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE 30*, 11 (2000), 1203–1233.

[2] 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.

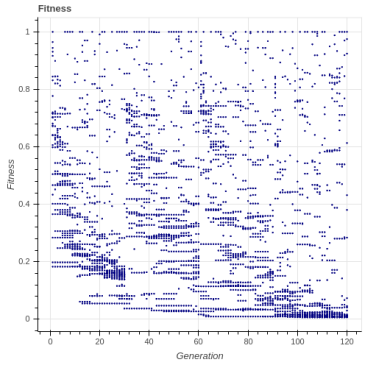
[3] JENKS, G., ET AL. SortedContainers: Sorted list, dictionary and set types for Python, 2014–. [Online; accessed ¡today¿].

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

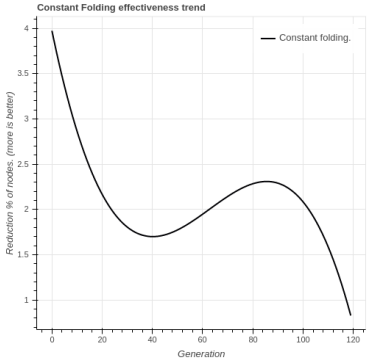
[5] STORN, R., AND PRICE, K. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization 11*, 4 (1997), 341–359.



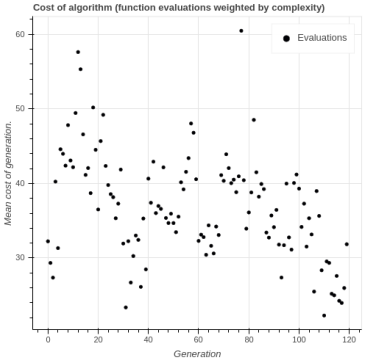
(a) Scaled complexity over generations.



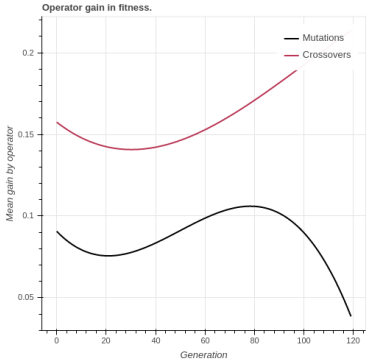
(b) Fitness values over generations.



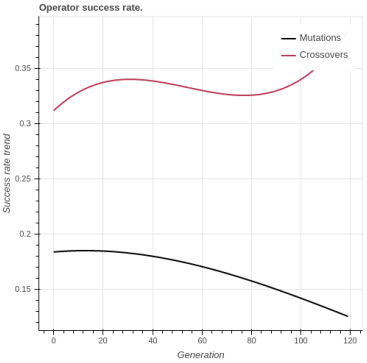
(c) Constant folding savings.



(d) Mean evaluation cost.



(e) Operator gain.



(f) Operator success rate.

Figure 4: Selection of visualizations generated by CSRM.