Convergence of symbolic regression using metaheuristics

Ben Cardoen
ben.cardoen@student.uantwerpen.be

May 4, 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 an insight into the behavior of the resulting model (e.g. continuity, derivatives, 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 (GP). The convergence characteristics of SR using GP is 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) is studied.

2 Introduction

2.1 Overview

2.2 Symbolic Regression

2.2.1 Compared to other techniques

2.2.2 Interpreting results

2.2.3 Applications

2.3 Convergence

2.4 Genetic Programming

2.5 Constant optimization problem

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 an X = n x k matrix of input data, and an Y = 1 x k vector 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 per feature. 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.

3.1.3 Entities

We will describe briefly the important entities in our design. In Figure 2 the architecture of our tool is shown. Tooling 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.
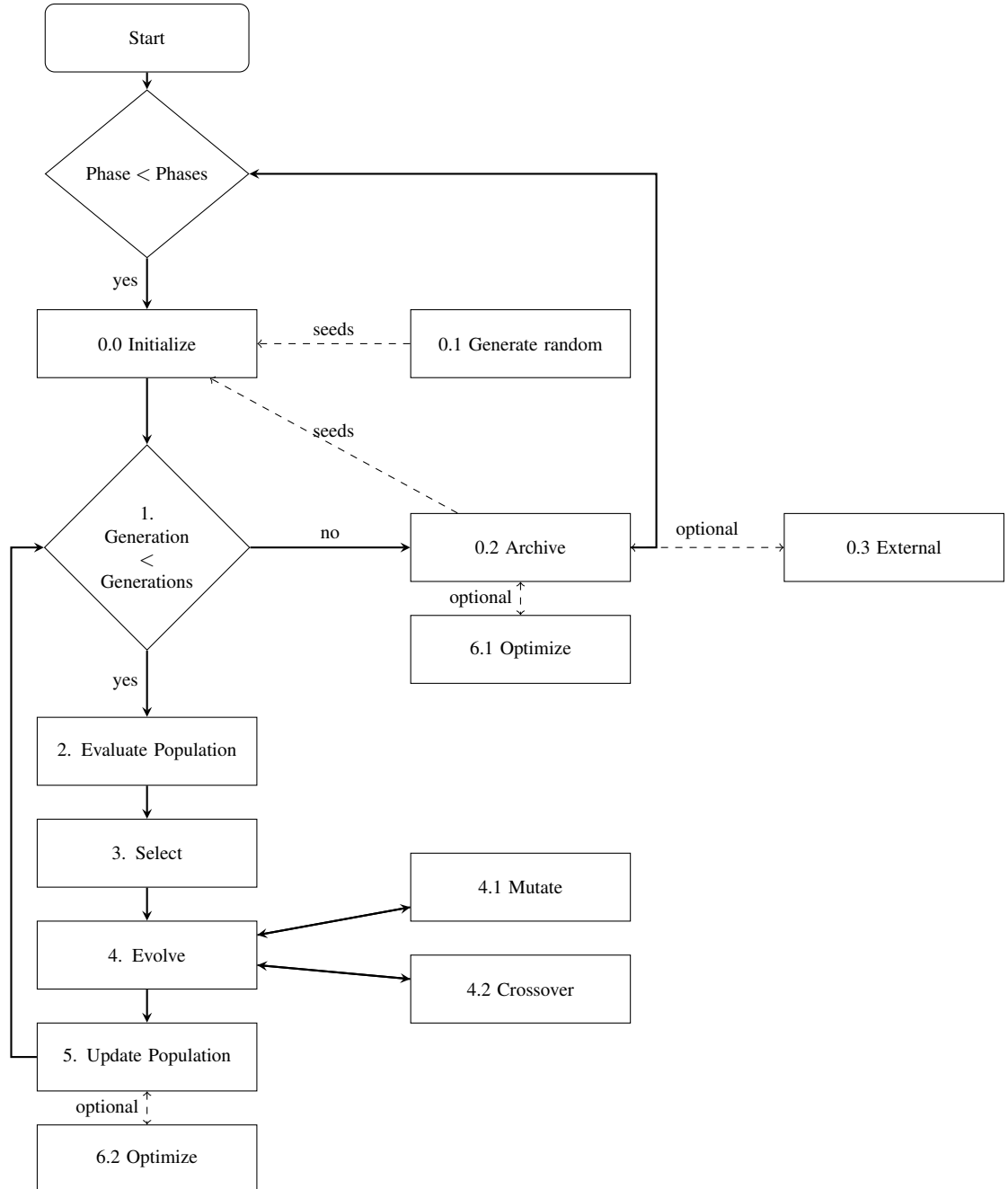
3.1.2 Control flow
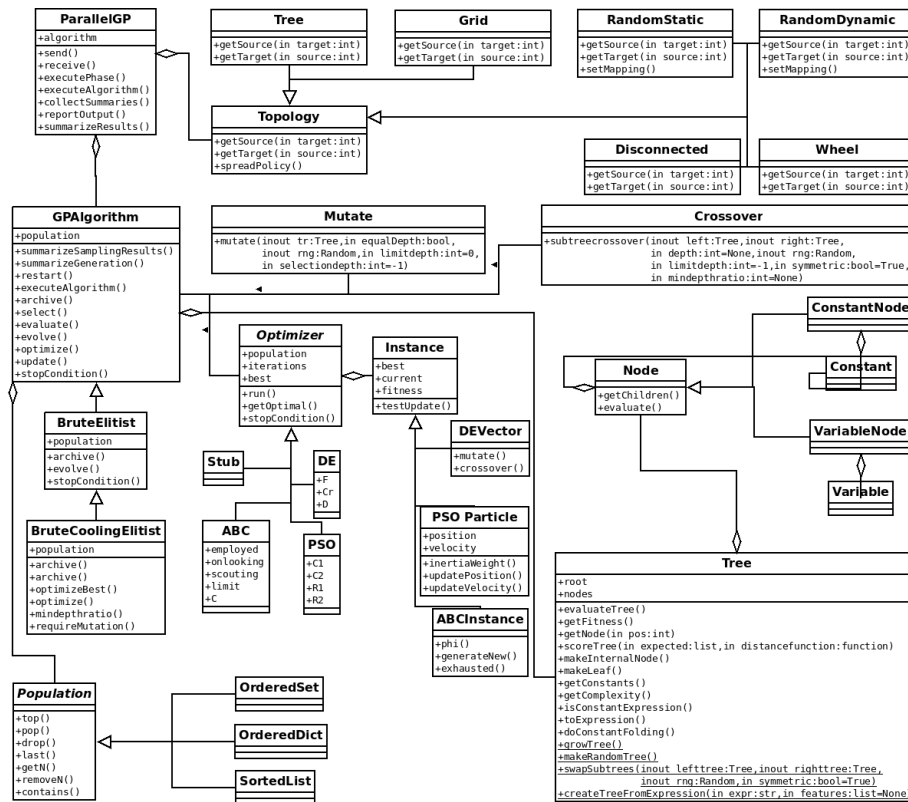

Figure 1: CSRM control flow.

ParallelGP
+algorithm
+send()
+receive()
+executePhase()
+executeAlgorithm()
+collectSummaries()
+reportOutput()
+summarizeResults()

Tree
+getSource(in target:int)
+getTarget(in source:int)

Grid
+getSource(in target:int)
+getTarget(in source:int)

RandomStatic
+getSource(in target:int)
+getTarget(in source:int)
+setMapping()

RandomDynamic
+getSource(in target:int)
+getTarget(in source:int)
+setMapping()

Topology
+getSource(in target:int)
+getTarget(in source:int)
+spreadPolicy()

Disconnected
+getSource(in target:int)
+getTarget(in source:int)

Wheel
+getSource(in target:int)
+getTarget(in source:int)

GPAlgorithm
+population
+summarizeSamplingResults()
+summarizeGeneration()
+restart()
+executeAlgorithm()
+archive()
+select()
+evaluate()
+evolve()
+optimize()
+update()
+stopCondition()

Mutate
+mutate(inout tr:Tree,in equalDepth:bool,
    inout rng:Random,in limitdepth:int=0,
    in selectiondepth:int=-1)

Crossover
+subtreecrossover(inout left:Tree,inout right:Tree,
    in depth:int=None,inout rng:Random,
    in limitdepth:int=-1,in symmetric:bool=True,
    in mindepthratio:int=None)

ConstantNode

Optimizer
+population
+iterations
+best
+run()
+getOptimal()
+stopCondition()

Instance
+best
+current
+fitness
+testUpdate()

Node
+getChildren()
+evaluate()

Constant

BruteElitist
+population
+archive()
+evolve()
+stopCondition()

DEVector
+mutate()
+crossover()

VariableNode

Stub

DE
+F
+Cr
+D

BruteCoolingElitist
+population
+archive()
+archive()
+optimizeBest()
+optimize()
+mindepthratio()
+requireMutation()

ABC
+employed
+onlooking
+scouting
+limit
+C

PSO
+C1
+C2
+R1
+R2

PSO Particle
+position
+velocity
+inertiaWeight()
+updatePosition()
+updateVelocity()

Variable

Tree
+root
+nodes
+evaluateTree()
+getFitness()
+getNode(in pos:int)
+scoreTree(in expected:list,in distancefunction:function)
+makeInternalNode()
+makeLeaf()
+getConstants()
+getComplexity()
+isConstantExpression()
+toExpression()
+doConstantFolding()
+growTree()
+makeRandomTree()
+swapSubtrees(inout lefttree:Tree,inout righttree:Tree,
    inout rng:Random,in symmetric:bool=True)
+createTreeFromExpression(in expr:str,in features:list=None)

ABCInstance
+phi()
+generateNew()
+exhausted()

Population
+top()
+pop()
+drop()
+last()
+getN()
+removeN()
+contains()

OrderedSet

OrderedDict

SortedList

Figure 2: UML of CSRM.

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 3.3.1 we will cover in depth the problem solved here.

Optimizer   CSRM provides an interface to continuous optimizers. In section 4 we will go deeper into the specific algorithms used. Each algorithm shares a population, instances of which can be subclassed if needed. 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

3.2.1 Distance function

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

$$Y = f(X)$$

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

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

e is 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 $[0, +\infty)$, 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

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

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 [11] 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 specimens based on training data X_t, with X_t a random sample from input data X of size j = r * k. R is the sampling ratio, the ratio between training and test data. K is the total amount of datapoints available per feature. 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, what we would like to know is how the algorithm's progress is related to the value. For example, if we add 10 more generations, or increase the population by a factor 1.5, do we gain or lose in predictive quality of the expressions? To define this we use a correlation between the fitness values using the training data and those of the full data. This provides an immediate link between the two sets of values. 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 gives us good measure of how the algorithm will converge on a good solution on the full data.

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 [8]. The algorithm has a parameter initial depth, each new expression in the population is created using that depth.

3.3.1 Invalid expressions

Generating a random expression is done by generating random subtrees and merging them. A critical point to observe is that randomly generating 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, one can return a predefined value that will still result in a valid tree. The downside of 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. 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. Our approach is 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 [4], 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 linearly. 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 a bottom up construction, where we generate small subtrees, evaluate them and if and only if valid merge them into a larger tree, allows for early detection of invalid expressions. A downside of this approach is the repeated evaluation of the subtrees. This 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 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.

3.5.1 Mutation

Mutation works by selecting 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 mutation 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 process3.3.1 apply here as well. The mutation operator will use an invalid subtree, at the cost of having to 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 inserts them into 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 last reflects in part 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 a 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 by seeding, from other processes, or the best expressions at the end of a phase. 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 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 nearly non existent, 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 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 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 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. 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. The problem with this is that this process is highly inefficient and uses nodes that could otherwise be used to approximate the expected values. 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 Populationsize

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 gives (for this tool) a good balance.

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 in both. 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.

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 more fair evaluation of the algorithm and optimizations used.

3.10 Incremental support

Our tool support 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 data-points. 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 5. 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 rendered solution is what is expected. The size of the search space makes detecting edge cases impossible. 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 run-time statistics ranging from fitness values per expression for each point in the execution,
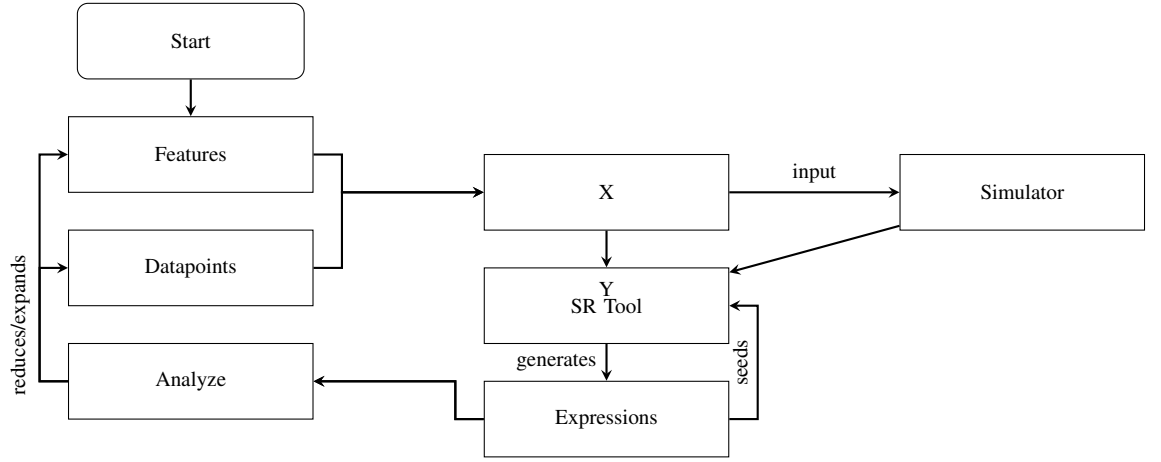
Figure 3: Incremental DOE using CSRM and a simulator.

convergence behavior over time, depth and complexity of the expressions, cost of evaluations, effectiveness of operators, correlation between training en 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 [3]. In Figure 4 a selection of the collected statistics on a testfunction is shown. The third testproblem 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 Constant optimization

4.1 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.
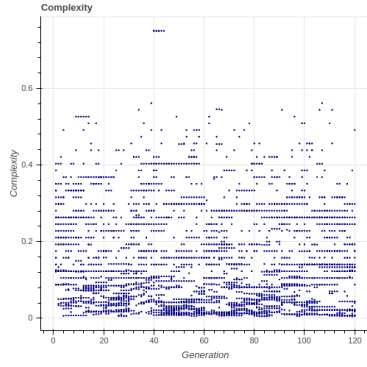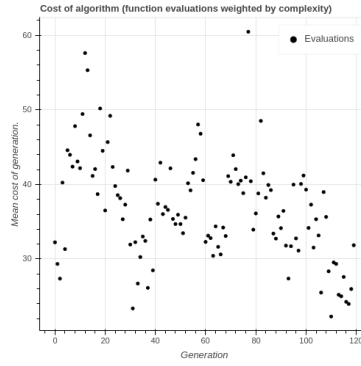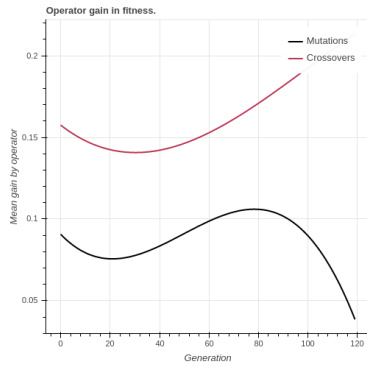
(a) Scaled complexity over generations.
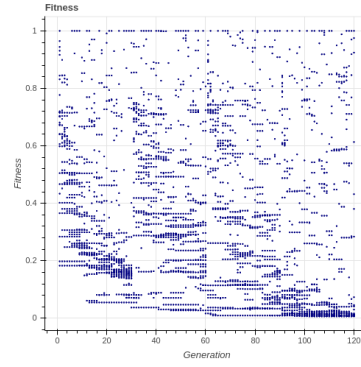
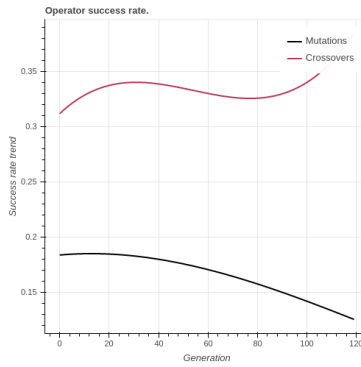(b) Fitness values over generations.
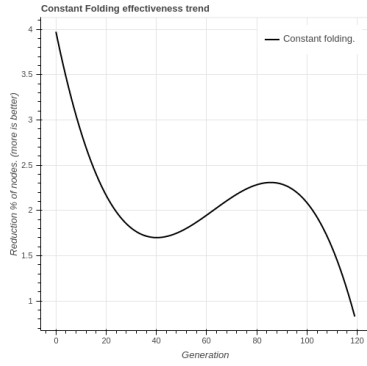
(c) Constant folding savings.

(d) Mean evaluation cost.

(e) Operator gain.

(f) Operator success rate.

Figure 4: Selection of visualizations generated by CSRM.

15

4.1.1 Initialization revisited

During initialization it is possible that a tree represents a constant expression. Such an expression is only a valid approximation if no feature has an influence on Y, which is an extremely unlikely edge case. A constant expression is of no use for the algorithm, but without measures to prevent or remove these they will still be formed. Ideally such an expression will have a high (worse) fitness value than non constant expression, 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 constant, but this right is replaced by either mutation or crossover with a constant expression then the node 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 remove the constant expressions immediately.

4.1.2 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. We will evaluate these effects in the experiments section.

Constant subtree folding　It is possible to 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. This leads to a saving nodes, possibly 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 tree. Constant folding requires an O(n) detection check, since the
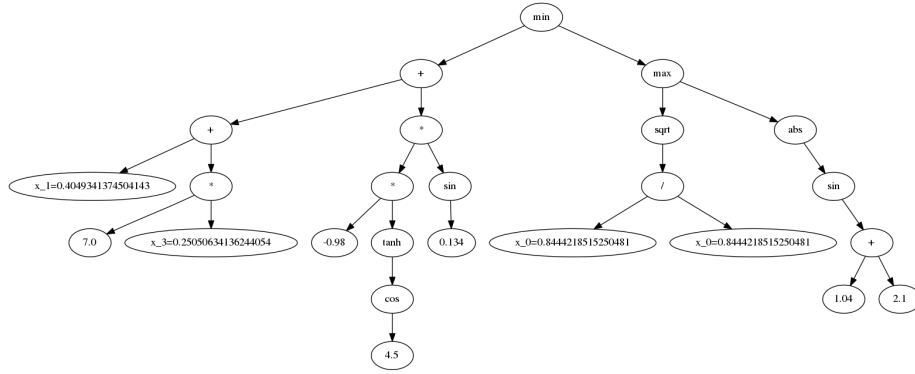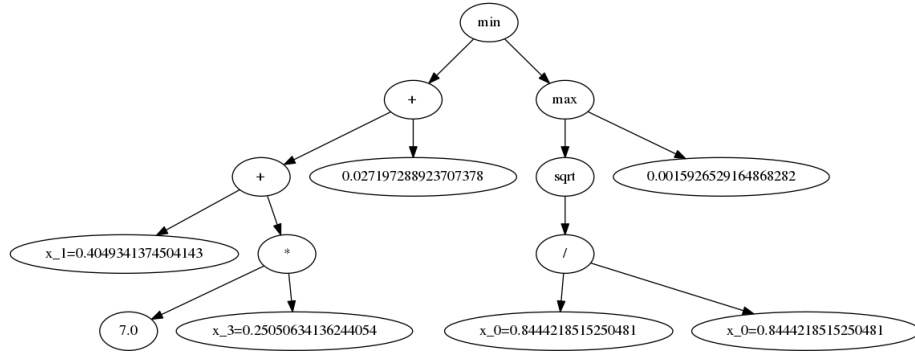
Figure 5: Tree before subtree folding.

Figure 6: Tree after subtree folding.

entire tree needs to be traversed. The folding operation itself is at worst O(n), if the entire tree is constant.

Edge cases In Figure 6 we 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 the figure we see a similar case in the right subtree, for 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

17

cannot be prevented from occurring. Another similar 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 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 mathematical software to simplify the expressions, and then convert them back to tree representations. Applying 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, they do not lead to a more efficient finding of the 'right' constant values. For this we need a continuous optimizer, algorithm designed specifically to optimize real valued problems.

4.2   Optimizers

Hybridizing GP   Using a real valued optimizer in combination with GP is a known solution [10, 2]. 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 (since all constant subtrees are folded), we would like to find the most optimal values for those constants resulting in a lower (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 problem 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 3.3.1 reappears here. We could pick random values in the search space, but these are likely to generate invalid trees. We need a balance between exploration and exploitation here.

Domain   Each tree instance has a completely different domain for each constant. We cannot therefore 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 [7] we use a default population of 50, and 50 iterations. 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.

4.2.1   ABC

Artificial Bee Colony [5] is a relative new nature inspired optimization algorithm. It is not limited to continuous optimization, and has even been used for symbolic regression itself [6]. 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 [9].

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 is quite often inappropriate. 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 shoud 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.

Configuration   We use a solution set of 50, a population of 25 employed particles, 25 onlookers and a single scout. These parameters are suggested in [5]. A higher scout value is not warranted, since we already have a good starting point. More exploration would also result in the initialization problem dominating the runtime cost of the optimizer.

4.2.2 PSO

Particle Swarm Optimization [7] 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, in our application a perturbed instance of the constant values we wish to optimize. 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 [1]. 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 4.2.2. Finally the global best is recorded.

Modification The algorithm updates all particles in sequence, then records the new global best. The velocity v at iteration i of a particle is updated using:

$$v_{i+1} = w_i * v_i + C_1 * (p_i - g_i) * R_1 + C_2 * (p_i - G_i) * R_2.$$

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 :

$$x_{i+1} = x_i + v_i$$

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, 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 (of the particle) and the global best (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. This allows it to cross areas where no gradient is observed, for example between two peaks 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. In our implementation we will halt the algorithm if it can't improve the global best after k/2 iterations, where k is the maximum number of iterations it can execute. The total cost in fitness evaluations is therefore O(nk).

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

- C_1 2

- C_2 2

- $w_i = \frac{1+r}{2}$ with r random in [0,1]

- R_1, R_2 r with r random in [0,1]

4.2.3 DE

Algorithm

Configuration

5 Experiments

5.1 Operators

5.2 Constant Folding

5.2.1 Savings

5.2.2 Effect on convergence

5.3 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.

5.3.1 2 Phases

Fitness

Cost

5.3.2 5 Phases

Fitness

Cost

5.3.3 10 Phases

Fitness

Cost

5.4 Conclusion

6 Related Work

7 Conclusion

8 Future work

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

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

[9] 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.

[10] 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.

[11] 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.