Convergence of symbolic regression using metaheuristics.

Ben Cardoen
ben.cardoen@student.uantwerpen.be—bcardoen@sfu.ca

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 expression, determine important features by their usage in the expression, and gain 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 paper will study convergence of a GP-SR implementation on selected use cases known for bad convergence. We revisit the constant optimization problem by hybridizing with 3 metaheuristics: PSO, DE and ABC. We test our approach using benchmark problems known to be hard problems for SR. Our results show that while hybridization can offer orders of magnitude improvement in fitness values, care has to be taken when to apply the optimizer in order to avoid overfitting. Neither of the three algorithms was optimal for all problem instances.

2 Introduction and design

Our tool, Convergence of SR using Metaheuristics (CSRM), is implemented in Python3. The language offers portability, rich libraries and fast development cycles. The disadvantages are speed and memory usage compared to compiled languages (e.g. C++) or newer scripting languages (e.g Julia). Furthermore, Python's use of a global interpreter lock makes shared memory parallelism infeasible. Distributed programming is possible using MPI.

Algorithm 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. We do not know in advance whether all features are needed to build the expression. The goal of the algorithm is to find f' such that $d(f(X), f'(X)) = \epsilon$ results in ϵ minimal. F is the process we wish to approximate with f'. Not all distance functions are equally suitable for this purpose. A simple root mean squared error (RMSE) has an issue of scale: the range of this function is [0, $+\infty$), which makes comparisons problematic, particularly 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, but there is no single recommended way to do so. In this work we use a distance function based on the Pearson Correlation Coefficient r. Specifically, we define

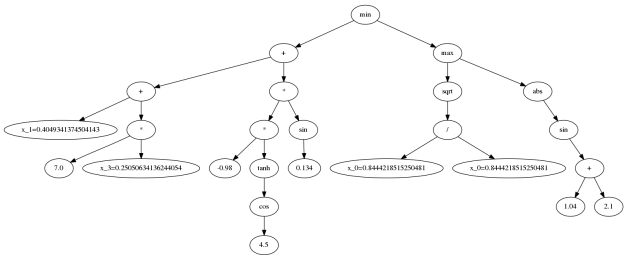
$$d(Y, Y') = 1 - |\frac{\sum_{i=0}^{n}(y_i - E[Y]) * (y'_i - E[Y'])}{\sqrt{\sum_{j=0}^{n}(y_j - E[Y])^2 * \sum_{k=0}^{n}(y'_k - E[Y'])^2}}| \quad (1)$$

R has a range of [-1, 1] indicating negative linear and linear correlation between Y and Y' respectively, and 0 indicates no correlation. The distance function d 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'. The use of the Pearson correlation coefficient as a fitness measure is not new, a variant of this approach has been used in [28].

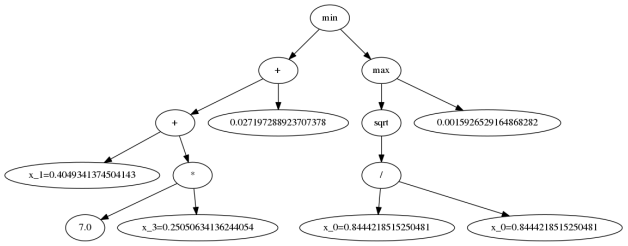
Genetic Programming Implementation We use Genetic Programming (GP) [17] to optimize the symbolic regression problem w.r.t. the distance function above. The GP algorithm controls a population of expressions represented as trees. These are initialized, evolved and selected to simulate evolution. The algorithm is subdivided into a set of phases. Each phase initializes the population with a seed provided by an archive populated by previous phases or by the user. A phase is subdivided in runs, where each run selects a subset of the population and applies GP operators. If this leads to fitness improvement it replaces the expressions in the population. At the end of a phase the best expressions are stored in an archive to seed subsequent phases, and are communicated to other processes executing the same algorithm with a differently seeded population. The next phase will then seed its population using the best of all available expressions. We use a vanilla GP implementation, with 'full' initialization method [17]. Expression trees are generated with a specified minimal depth. During evolution, expressions are limited to a specified depth. We use mutation and crossover in sequence. Mutation replaces a randomly selected subtree with a randomly generated tree. This introduces new information and leads to exploration of the search space. Crossover selects two trees based on fitness and swaps randomly selected subtrees between them. This leads to exploitation of the search space. The selection for crossover is random, biased by fitness value. A stochastic process decides if crossover is applied pairwise (between fitness ordered expressions in the population) or at random. The initialization of expression trees can lead to invalid expressions for the given domain. The probability of an invalid expression increases exponentially with the depth of the tree. A typical example of an invalid tree is division by zero. While some works opt for a guarded implementation, where division is altered to return a 'safe' value if the argument is zero, we elect to discard invalid expressions and replace them with a valid expression. We implement an efficient bottom up approach to construct valid trees where valid subtrees are merged. In contrast to a top down approach this detects invalid expressions early on and avoids unnecessary evaluations of generated subtrees. Nevertheless, the initialization constitutes a significant computational cost in the set up of each phase and in the application of the mutation operator.

3 Constant optimization

Generating the optimal constant value in an expression tree is a hard problem for GP. In contrast to the discrete problem of selecting and combining from a limited set of base functions, the selection range of a constant value is huge. In this section we cover our approach to this problem. A constant can appear in an expression as a linear weight for a base function or as a leaf. The following clarifies the difference. In $f(x) = \sin(\log(2, x))$ the 2 is a constant value, while in $g(x) = 3.14sin(x) + 4.25cos(x)$ the 3.14 and 4.25 are linear weights. In our implementation, the tree encoding has a hidden linear weight for each node which can be optimized independently. When we refer to constant optimization, we are referring to the process of finding the optimal values of constant values in the expression. The size of the constant set dominates the size of the search space. We intentionally restrict this set to [0,1]. This reduces the size of c from 2^{64} to 2^{23} since we use the IEEE 754 double representation. This restriction in range does not prevent larger constants from being evolved. The algorithm will evolve subtrees combining base functions with constants in order to find more optimal values. An expression tree is constant if all its children are constant expression trees. 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, in the degenerate case where the entire tree is a constant expression. Upon detecting a non constant subtree, the algorithm returns early without a full traversal. Using the checking procedure a tree marked as a constant expression is not allowed in the initialization procedure. It is still possible to create constant expressions by applying crossover. The mutation operator will not generate constant subtrees. Our tool does not prevent constant expressions from forming in this way, the evaluation step following crossover will filter out the constant expressions using evolutionary pressure. A tree can contain subtrees that represent constant expressions. This is an immediate effect of the GP algorithm trying to evolve the optimal 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 and waste memory and evaluation cost. There is a counterargument: parts of a constant



(a) Tree before subtree folding.



(b) Tree after subtree folding.

Figure 1: Constant subtree folding.

subtree can help evolve constants faster than random selection. It is possible that folding such subtrees leads to worse fitness values. We can use a depth sensitive objective function to try to mitigate this effect, but a more direct approach is replacing the subtrees. Using the previous constant expression detection technique we can collect all constant subtrees from a tree and replace it with the constant value they represent, which we will refer to as folding. Constant folding can have a negative effect on convergence in some cases. A constant subtree with depth d represents between d and $2^{d+1} - 1$ constants. Both mutation and crossover can use these constants to generate more expressive trees. Constant folding is a trade-off between this expressive power and time and space complexity. More importantly, it is a necessity in order to optimize the constant values with a continuous optimizer. It reduces the dimensions of the optimization problem significantly. In Figure 1 we see the effect of applying the constant subtree folding.

Optimizers Using a continuous optimizer in combination with GP is a known solution [21, 7] to the constant optimization problem. Selecting which algorithm to combine with GP is a difficult question and, to our knowledge, no comparative study is available. Given a tree with k constant leaves and all constant subtrees folded, we would like to find the optimal values for those constants. In most optimization problems an initial solution is not provided, only the problem domain and an objective function. However, we do have an initial solution, namely that generated by GP. Instead of choosing random points in the search space, we therefore opt to perturb this initial solution. In order to make a fair comparison between the algorithms, they are, to the extent possible configured similarly. Based on the optimal value for PSO [10] we use a default population of 50. The iterations are set at 50. We can apply the optimizer on each expression each iteration or only on a selection of the best expressions at each iteration or only at the end of a phase on all, a selection, or only the best expressions. In our experiments we will show the effect on convergence of these choices.

ABC Artificial Bee Colony [8] is a relatively new nature inspired algorithm. It is not limited to continuous optimization, and has even been used for symbolic regression itself [9]. A key advantages over other algorithms is a low parameter count. ABC is good at exploration (thanks to its scouting phase) though sometimes lacking in exploitation. To resolve this, ABC can be combined with more exploitative algorithms [18]. ABC uses three distinct phases per iteration. It 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 replaces it if an improvement in fitness is obtained. If this fails for a preset number of iterations, the solution is marked exhausted and a new one scouted. Scouting in this context is generating a new potential solution. After the employed phase, the onlooking particles decide, based on a fitness weighted probability, which solutions should be further optimized. In contrast to the employed particles the onlooking particles swap out solutions each iteration. Finally, exhausted solutions are replaced by scouted solutions. The algorithm initializes its solution set by multiplying each constant with a random number $\in [-1, 1]$. Each instance records its best solution. We use the initial value of each constant as the mean of a normal distribution and generate values within 2 standard deviations. A configurable scaling factor is in-

troduced, in our test problems we use 20. This value is a trade-off between exploration and an increasing probability for generating invalid solutions. A solution is updated if its fitness improves. The new fitness weights for the roulette wheel selection are calculated and the global best is updated. Unlike PSO, a solution is only updated if an improvement is measured. In contrast to DE, equal fitness values do not lead to an update. An equality update allows an optimization algorithm to cross zero gradient areas in the fitness landscape. The influence solutions have on each other in ABC is not as great as in DE. In PSO, the entire position, in all dimensions, is updated. In DE this depends on a parameter. This distinction can have a large impact on convergence and is determined by the correlation or separability of each constant. We do not know in advance if our problem instances are separable or not. The configuration of our tool for ABC is:

- limit = 0.75 * onlookers / 2 : If a solution does not improve after this many iterations, it is marked exhausted. This limit is scaled by the number of dimensions per instance.
- population = 50 solutions.
- onlookers = 25 : The number of onlookers. Setting this value to half that of the employed finds a balance between exploitation and evaluation cost.
- employed = 50
- scouts = 25 : This is a maximum number of scouts used after a solution is exhausted.

This configuration is guided by the findings in [8]. Compared to PSO and DE,a single iteration in ABC will have more optimization per solution, but on fewer dimensions.

PSO Particle Swarm Optimization [10] is one of the oldest population based metaheuristics. It consists of particles that share information with each other about the global best solution. Each particle is assigned a position in the n-dimensional search space. A particle's position is updated using its velocity which is influenced by information from the global best and the local best. The concept of inertia is used to prevent velocity explosion [3]. Each particle is given a random location at start. Similar to our ABC imlementation, we perturb the known solution with a random value $\in [0, 1]$. To avoid premature convergence each particle is assigned a small but non-zero velocity. Without this velocity all particles are immediately attracted to the first global best. The algorithm updates all particles in sequence by updating each velocity using a weighted sum of local and global best, then updating its position using the velocity. The configuration of our tool for PSO is:

- C_1, C_2 = 2, weights for global and local best position differences, recommended in [11].
- $w_i = \frac{1+r}{2}$ with r random $\in [0, 1]$: A random inertia weight leads to faster convergence for a generic problem set [1].
- R_1, R_2 r with r random in [0,1], random weights for the global and local best positions.
- population = 50 : PSO is not sensitive to populations larger than this value [12].

CSRM's optimizer does not impose constraints on the domain of each constant. Finding the domain of a constant in the expression tree requires a domain analysis of the expression tree. Finding the exact domain is infeasible, given that some of the data points for features are unknown (e.g. validation or test data). It is thus possible that a particle obtains values outside the valid domain of one or more constants, resulting in an invalid expression tree. If so, the particle temporarily no longer contributes to the search process.

DE Differential Evolution is a vector based optimization algorithm, operating by computing the difference between particles. The algorithm has a population of n vectors and holds a linear set of values to optimize, one per dimension. Similar to our approach in initializating PSO and ABC, we perturb a known (sub)optimal solution. A vector stores its current value, and the best value. CSRM uses a DE/rand/2/bin configuration. The notation indicates a random selection of (2) vectors with binomial crossover. This configuration is referenced [25] as one of the most competitive for multimodal problems with good convergence characteristics to the global optimum. Each iteration the algorithm processes all vectors. First a mutation step obtains a new vector based on the difference

of 2 randomly selected, with replacement. Then we apply a binomial crossover operation. Finally we compare with the new vector and replace it if it has the same or better fitness, in contrast to PSO and ABC. The equality update allows DE to traverse zero gradient areas in the search space. The configuration of our tool for DE is:

- F = 0.6 : Weight determines the mutation step. The value of 0.6 is reported as a good starting value[4].
- Cr = 0.1 : The Cr value influences the crossover step. It should be in [0,0.2] for separable functions, and [0.9, 1] for non-separable functions. We cannot assume dependency between the constants, and therefore use 0.1. This results in DE focussing along the axes of each dimension in its search trajectory.

Compared to PSO DE has a low parameter count, optimal values for these parameters can be found in literature [4]. The population size should be t * d with t in [2, 10]. Since we do not know d in advance, and to keep the comparison fair we set the population at 50, allowing for optimal values for up to 25 dimensions (constants).

4 Experiments

An open source repository holds the project's source code, benchmark scripts, analysis code, plots and extra material. CSRM is implemented using Python3, the test system uses Python 3.5. All benchmarks were performed on an Intel Xeon E5 2697 processor with 64GB Ram, with Ubuntu 16.04 LTS operating system. The experiments use a fixed seed in order to guarantee determinism. Recent work on the convergence of GP-based SR [15, 16] featured a set of benchmark problems, using at most five features. CSRM does not know which features are used, making the problem harder.

Setup We test 15 functions with the configuration:
- population : 20
- minimum depth : 4; maximum depth : 10
- phases : (2, 5, 10); generations per phase : 20
- features 5; datapoints : 20
- range : [1,5]
- size of archive : 20
- expressions to archive per phase : 4
- optimization : optimize at end of phase

Measures For all functions, we compare the gain in fitness with and without using an optimizer. If m_n is a measures the gain without the optimizer, and m_a with the optimizer, we define the relative gain as $g_{ma} = \frac{m_n}{m_a}$. If m_a is zero, we use $-log_{10}(m_n)$ to represent the gain. If both are zero, the gain is 1. A value of g > 1 indicates the ratio with which the optimizer improves the result. A g value < 1 indicates a regression. The 15 functions have wildly varying convergence behavior. In order to make sense of the data, we then apply a log scale $g_{lma} = -\log_{10}(g_{ma})$. As measures we use the best fitness value on the training data, and the best on the full data set. We take the mean of the fitness of the 5 best expressions on training and the full data as well. This last measure gives us an indication on how the optimization process acts on the 'best' set of the population. Note that in our configuration, the 4 best expressions are always optimized.

2 Phases Figure 2d exhibits the performance of the optimizers on training data. We see that for these the improvements are significant, with ABC scoring an increase of 2.5 orders of magnitude for problem 6. For the other problems the increase is still large, especially given that our fitness function has a range of [0,1]. We also observe the significant regression for problem 6. This is likely due to overfitting. The algorithm in question (DE) optimizes the 4 best candidates of the last phase, but it is possible that these optimized expressions actually form a local optimum for the training data with poor fitness for the validation data. By archiving these, the convergence is hindered in the next phase. The same behavior occurs to some extent for expressions 7 and 9. A second explanation can be found in our implementation of the population. The algorithm enforces distinct fitness values for all expressions. In an edge case it is possible that these optimized samples form a barrier, preventing other expressions from evolving past them. The optimized expressions in effect trap the rest of the population, which given our low generation count can explain this behavior. The mean fitness of the 5 best expressions shows significant improvements. Notice the strong correlation between fitness values on training and full data. This was a concern in the setup of the experiments. The optimizers could introduce overfitting on the training data. This risk is mitigated by the relatively low number of iterations

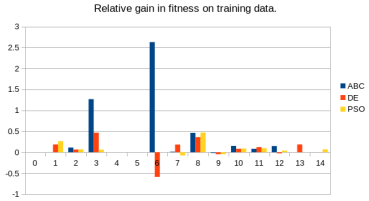
each optimizer has been allocated. For the minimum fitness on the full data, ABC outperforms the others. For the mean evaluation PSO is a good candidate. In this stage of the experiments, there is no single algorithm that is ideal for all problems. This once again confirms the NFL theorem for metaheuristics which states that no metaheuristic is optimal for all problem instances [29].

5 Phases With 5 phases we see in Figure2h a more diverse effect. ABC scores exceptionally well on problem 6, in sharp contrast with the 2 phase experiment. PSO scores better overall for the training data. When it comes to improving the mean of the best 5 expressions, PSO is a stable choice if we disregard the outlier values for problem 5. The correlation between training and full fitness scores is good for both measures. This demonstrates that the optimizer is not (in this experiment) introducing overfitting on the training data. The adverse effect of the optimizer on some test problems is still present. For the best fitness values on the full data DE is the better candidate. While ABC scores exceptionally high on problem 6, DE scores better overall. When we look at the mean there is no clear winner.

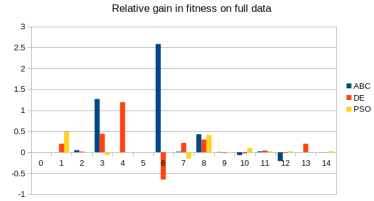
10 Phases If we observe the convergence after 10 phases we see a more pronounced effect. In Figure 2l we see that for several problems the optimizers are no longer improving w.r.t. the unoptimized algorithm. This only holds for the best values. For the mean values the improvements are still significant. It becomes clear that the optimizer can force the algorithm into a local optimum. The correlation between fitness results on the training data and full data is starting to weaken as well, compared to the experiments with 2 and 5 phases. Looking at the fitness values for the full data, DE is the more stable algorithm. When it regresses its losses are smaller than the others, while its gains are strongest on the most problems. For the mean fitness of the full data a similar argument can be made, with the exception of problem 2 where DE fails severely. Another aspect is that after 100 generations the fitness values are extremely small, in the order of 1e-15. We measure the relative gain with respect to the algorithm without an optimizer, but as the fitness values decrease rounding errors start to influence the calculations more and more. The fitness values are approaching the floating point epsilon values. For our implementation epsilon is set at 2.22 e-16. For problem 0, a minimum fitness value of 0 is found after 2 phases. For others far more iterations are needed. We need to make a trade-off in order to be able to compare all 15 problems. Giving each problem an equal budget in iterations is the more fair approach. Another approach is implementing a stop condition that halts within a certain distance of a desired fitness threshold, but this approach is fraught with issues. There is no guarantee exactly how many iterations are needed. This approach requires knowing the problem 'hardness' [22] in advance, but by the very definition of our problem statement we do not know how hard our problem is.

5 Related Work

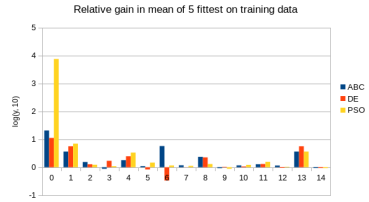
Symbolic regression fits a model, given a set of input values, to a known output. Other machine learning techniques such as Neural Networks, Support Vector Machines and Random Forests have the same functionality. Where SR distinguishes itself is in the white box nature of the model. The convergence characteristics of SR are an active field of study [15]. New approaches in SR such as GPTIPS and FFX [24, 19] focus on multiple linear regression, where a linear combination of base functions is generated. While GPTIPS still uses GP, FFX is completely deterministic and eschews GP. In the recent work of [30] SR is compared with these approaches on a series of benchmark functions. The authors concluded that while SR can have a slower convergence rate compared to conventional machine learning algorithms, the difference is not that large. Even though SR is usually associated with GP, there exists a wide variety of alternative implementations. A non GP approach using elements from Grammatical Evolution (GE) [20], several genetic algorithm techniques and continuous optimizers (DE, PSO) has been presented in [14] with promising results in terms of convergence rate and accuracy. The more recently introduced ABC algorithm has also been used for SR [9]. Ant Colony Optimization [6] has been used to generate programs [23], the same functionality that allows GP to be used for SR. Several approaches to the constant optimization problem exist. The traditional solution of generating random constants [17] remains a slow approach given the size of the search space. In [27] a structure based



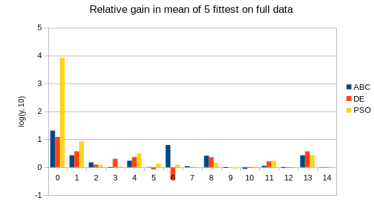
(a) Relative gain in best fitness of training data - 2 phases



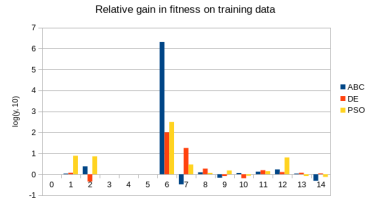
(b) Relative gain in best fitness of full data - 2 phases



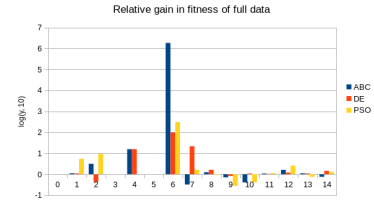
(c) Relative gain in mean fitness of 5 best on training data - 2 phases



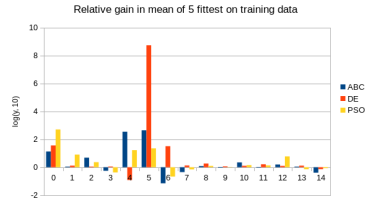
(d) Relative gain in mean fitness of 5 best on full data - 2 phases



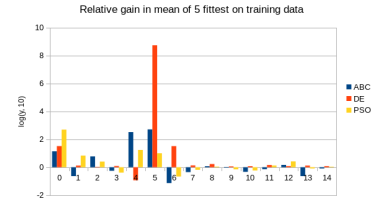
(e) Relative gain in best fitness of training data - 5 phases



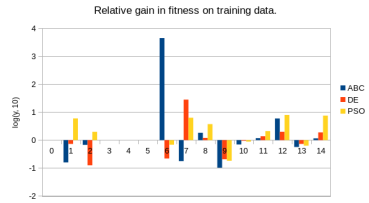
(f) Relative gain in best fitness of full data - 5 phases



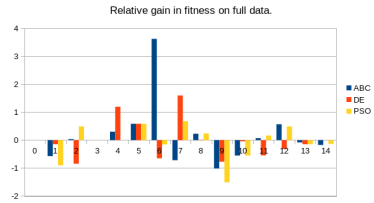
(g) Relative gain in mean fitness of 5 best on training data - 5 phases



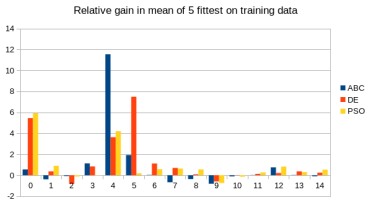
(h) Relative gain in mean fitness of 5 best on full data - 5 phases



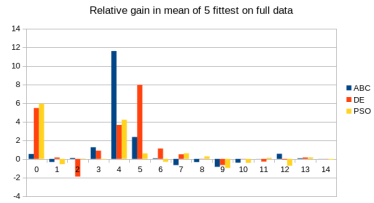
(i) Relative gain in best fitness of training data - 10 phases



(j) Relative gain in best fitness of full data - 10 phases



(k) Relative gain in mean fitness of 5 best on training data - 10 phases



(l) Relative gain in mean fitness of 5 best on full data - 10 phases

Figure 2: Relative gain of optimizer after 2,5,10 phases.

approach is reported to improve the random constant generation. Each constant is represented not by a single leaf node but by an evolving subtree. Apart from improved convergence, this approach also avoids hybridization of the original GP algorithm. There is a similarity with our approach, where we fold constant subtrees instead of evolving these separately. Their approach underlines our statement that evolving subtrees to generate constants can be quite effective. Our folding approach prevents vanilla GP from doing this, their approach splits constant generation from the GP algorithm but then reuses the same tree evolution techniques to evolve constants. In [7] the concept of numeric mutation is introduced where constants are mutated separately from the remainder of the tree using an implementation based on a temperature-biased normal distribution inspired by simulated annealing [13]. Another approach is made by applying GE to generate constants [5] where grammars are used to generate and evolve constants. Examples in the field of hybridization are DE [2] and PSO [16]. Advances such as semantically aware operators [26] and modularity, e.g. Koza's Automatically Defined Functions [17], are not applied here.

6 Conclusion

We have implemented a baseline GP SR tool based on a tree representation that allows for inspection of the convergence process through extensive statistics and visualization. The constant generation and optimization problem was tackled using a two-pronged approach. We apply constant folding to reduce the size of the expression trees. This approach, although it can slow the generation of constants, is vital for the second stage of our approach. We hybridize GP with 3 continuous optimization algorithms and compare the results. We find that, while in general convergence improves, the application of the continuous optimizer should be chosen such that overfitting is not introduced. As expected from the NFL theorem no single algorithm was optimal for all problem instances, validating both the test problems we used and the implementation of the algorithm. The tool's modular architecture allows it to be extended with new metaheuristics.

References

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

[2] CERNY, B. M., NELSON, P. C., AND ZHOU, C. Using differential evolution for symbolic regression and numerical constant creation.

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

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

[5] DEMPSEY, I., O'NEILL, M., AND BRABAZON, A. Constant creation in grammatical evolution. *International Journal of Innovative Computing and Applications 1*, 1 (2007), 23–38.

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

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

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

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

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

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

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

[13] KIRKPATRICK, S., GELATT, C. D., AND VECCHI, M. P. Optimization by simulated annealing. *SCIENCE 220*, 4598 (1983), 671–680.

[14] KORNS, M. F. *Abstract Expression Grammar Symbolic Regression*. Springer, New York, 2011, pp. 109–128.

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

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

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

[18] 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.

[19] MCCONAGHY, T. FFX: Fast, Scalable, Deterministic Symbolic Regression Technology. In *Genetic Programming Theory and Practice IX*, R. Riolo, E. Vladislavleva, and J. H. Moore, Eds. Springer New York, 2011, pp. 235–260.

[20] O'NEIL, M., AND RYAN, C. *Grammatical Evolution*. Springer US, Boston, MA, 2003, pp. 33–47.

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

[22] POLI, R., AND VANNESCHI, L. Fitness-proportional negative slope coefficient as a hardness measure for genetic algorithms. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation* (New York, NY, USA, 2007), GECCO '07, ACM, pp. 1335–1342.

[23] ROUX, O., AND FONLUPT, C. Ant programming: or how to use ants for automatic programming. In *Proceedings of ANTS* (2000), vol. 2000, Springer Berlin, pp. 121–129.

[24] SEARSON, D. P., LEAHY, D. E., AND WILLIS, M. J. Gptips:an open source genetic programming toolbox for multigene symbolic regression.

[25] 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.

[26] UY, N. Q., HOAI, N. X., ONEILL, M., MCKAY, R. I., AND GALVÁN-LÓPEZ, E. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines 12*, 2 (2011), 91–119.

[27] VEENHUIS, C. B. *Structure-Based Constants in Genetic Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 126–137.

[28] WILLEM, L., STIJVEN, S., VLADISLAVLEVA, E., BROECKHOVE, J., BEUTELS, P., AND HENS, N. Active learning to understand infectious disease models and improve policy making. *PLOS Computational Biology 10*, 4 (04 2014), 1–10.

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

[30] ZEGKLITZ, J., AND POSÍK, P. Symbolic regression algorithms with built-in linear regression. *CoRR abs/1701.03641* (2017).