Inexact Simplification of Symbolic Regression Expressions with Locality-sensitive Hashing

Guilherme Seidyo Imai Aldeia
Federal University of ABC
Santo Andre, São Paulo, Brazil
guilherme.aldeia@ufabc.edu.br

Fabrício Olivetti de França
Federal University of ABC
Santo Andre, São Paulo, Brazil
folivetti@ufabc.edu.br

William G. La Cava
Boston Children's Hospital
Harvard Medical School
Boston, Massachusetts, USA
william.lacava@childrens.harvard.edu

ABSTRACT

Symbolic regression (SR) searches for parametric models that accurately fit a dataset, prioritizing simplicity and interpretability. Despite this secondary objective, studies point out that the models are often overly complex due to redundant operations, introns, and bloat that arise during the iterative process, and can hinder the search with repeated exploration of bloated segments. Applying a fast heuristic algebraic simplification may not fully simplify the expression and exact methods can be infeasible depending on size or complexity of the expressions. We propose a novel agnostic simplification and bloat control for SR employing an efficient memoization with locality-sensitive hashing (LHS). The idea is that expressions and their sub-expressions traversed during the iterative simplification process are stored in a dictionary using LHS, enabling efficient retrieval of similar structures. We iterate through the expression, replacing subtrees with others of same hash if they result in a smaller expression. Empirical results shows that applying this simplification during evolution performs equal or better than without simplification in minimization of error, significantly reducing the number of nonlinear functions. This technique can learn simplification rules that work in general or for a specific problem, and improves convergence while reducing model complexity.

CCS CONCEPTS

• **Computing methodologies** → **Symbolic and algebraic algorithms**; • **Mathematics of computing** → *Genetic programming*.

KEYWORDS

locality sensitive hashing, simplification, symbolic regression, genetic programming

1 INTRODUCTION

Symbolic regression (SR) addresses the challenge of jointly optimizing the parameters and structure of a function. Given a set of d-dimensional inputs $\mathcal{X}$ and target outputs $\mathbf{y}$, it solves $\mathbf{y} \approx \hat{\mathbf{y}} = \hat{f}(\mathcal{X}, \hat{\theta})$, typically by minimizing the difference between $\mathbf{y}$ and $\hat{\mathbf{y}}$ while concurrently prioritizing simplicity [18]. Since first proposed by Koza as an application of genetic programming (GP) [15], it has been successfully used in several fields, such as clinical decision support [20], financial modeling [24], aerospace engineering [17], and physical sciences [3].

Despite its success, SR is an NP-hard problem [27], meaning that searching for the best regression model is computationally inefficient unless P=NP is proved. The search space is vast, containing isomorphic mathematical expressions [5], introns (parts of the model that do not influence predictions) [1], bloat (growth in model size with unjustified improvement in loss) [23], and model overparameterization (excessive number of parameters to tune) [9]. While these do not directly affect model accuracy, they can inflate model size, decreasing simplicity and interpretability.

There are several different approaches in the literature for tackling this problem, such as automatic simplification of models [12], parametric rules to rewrite expressions [16], Bayesian loss metric for model selection [4], constrained search space by restricting model structure [8], or application of an exact simplification procedure such as equality saturation [9]. Although all of those have their own benefits, they also have limitations, chiefly the additional computational cost and the need to manually write the algebraic simplification rules.

We propose a technique to dynamically build the simplification rules to address these limitations by memoizing observed equivalences with hashing. In short, we build a hash table where the key is created using locality-sensitive hashing (LHS) [11] based on the prediction vector, and the value is the smallest observed subtree corresponding to that vector. LHS is often used to efficiently retrieve nearest-neighbor points in a database system [13]. By analogy, our proposal transforms expressions into similar expressions within the phenotype (i.e., behavior) space that are smaller.

When integrating and applying the simplification into every expression throughout the generations, we observed benefits such as faster convergence and reduced mean squared error. The size of the returned expression remained the same, but its complexity was significantly reduced when considering the use and chaining of nonlinear functions. Furthermore, we analyzed the expressions identified as equivalent and noticed that the algorithm successfully captures many known algebraic identities.

The paper is organized as follows. Section 2 provides an overview of related work in bloat control, hashing, and simplification methods in SR. Section 3 introduces the application of Locality-Sensitive Hashing (LSH) in our simplification method. Section 4 details simplifying expressions through memoization with LSH. Section 5 outlines the experimental methods, including the SR framework used to test the method. Section 6 presents and discusses empirical findings, focusing on the impact of our approach on solution quality, expression size, and eliminating unnecessary operations. Finally, Section 7 draws conclusive remarks, summarizing key contributions and suggesting future work.

2 RELATED WORK

Helmuth et al. [12] proposed an automatic simplification in the program synthesis context by randomly removing subtrees and accepting the change if that did not affect the final prediction. Burlacu et al. [5] explored the idea of hashing an SR tree using some basic algebraic rules for handling commutative operations, such that two equivalent subtrees are hashed to the same value. Nguyen and Chu explored indirect simplification through mutation [26], pruning a population percentage by replacing subtrees with a newly small random tree with similar semantics.

Some previous work focuses on sets of rules to perform simplification. Kulunchakov [16] defined equivalent algebraic models as those that produce the same outputs over a small range and by evolving expressions with GP to use as replacements. In [9], de França and Kronberger proposed using equality saturation, an enumerative simplification algorithm, to remove redundant parameters of symbolic expressions. They noticed that overparametrization and unnecessary complexity are predominant in current SR algorithms. Some attempts rely on something other than the manipulation of symbols. Bomarito et al. [4] proposed to reduce bloat by using a custom Bayesian fitness metric with regularization.

Contrary to previous work, our approach does not rely on sets of rules or expensive computing resources. We propose a data-driven and computationally efficient way of inexact simplification that can be embedded into any GP framework, being agnostic because it only requires slicing the expressions and replacing subtrees.

3 LOCALITY-SENSITIVE HASHING

Locality-sensitive Hashing (LSH) was first introduced in [11] as a technique designed to efficiently find approximate neighbors in high-dimensional spaces through the usage of hash functions that preserve local proximity [13]. The main idea is to hash input vectors so that similar vectors are more likely to be mapped to the same hash bucket, while very distinct vectors are less likely to do so. Figure 1 illustrates this process.

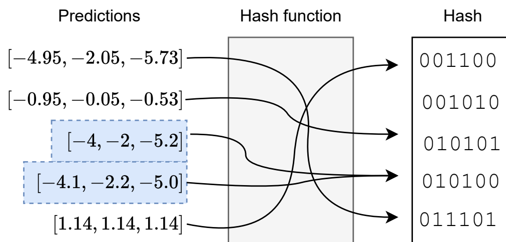

Figure 1: Given a set of vectors (i.e. the predictions), similar predictions are mapped to the same hash. The two highlighted vectors present close values in this example and are mapped into the same hash.

Here, we employ the SimHash LSH method as a proof of concept [6], as it has a straightforward implementation and requires few lines of code to be incorporated into existing frameworks. This results in a trade-off between computational efficiency and accuracy, beneficial in scenarios where exact similarity search becomes impractical [13] due to the curse of dimensionality [7].

Given a hash size of b bits and a set of d-dimensional data $\{x_i \in \mathbb{R}^d\}$, we first stipulate the hash size as b bits. Then, we create a plane $\mathbf{P} \in \mathbb{R}^{b \times d}$ where each $\mathbf{P}_{(i,j)} \sim \mathcal{N}(0, 1)$.

Whenever we want to query a data point x_i, we first calculate the matrix multiplication $\mathbf{P} \cdot x_i$, resulting in the vector $\mathbf{q} \in \mathbb{R}^{b \times 1}$, and the hash $h(x_i)$ is calculated as

$$h(x_i)_j = \begin{cases} 1 & q_j > 0 \\ 0 & \text{otherwise} \end{cases}$$

This forms a bit string that is used as the key to the hash table. The main property of this hash function is that the probability of two hashes being equal, i.e., $Pr[h(x) == h(y)] = 1 - \frac{\theta(x,y)}{\pi}$, is proportional to their cosine similarity, and cosine similarity is proportional to the normalized l_2-euclidean distance.

Every hash table entry will store all queried objects with that same hash value. After inserting many objects, similar objects will be clustered around the same keys. The higher the number of bits used to generate the hash key, the larger the number of less dense clusters, increasing the accuracy of the similarity estimation.

4 SIMPLIFYING EXPRESSIONS BY MEMOIZATION

The idea of inexact simplification consists of having a simplification table where the key is the SimHash (as described in §3), and the values are a list of expressions hashed to that particular key.

In the first step, we evaluate the expression, keeping a trace of the evaluation at every node. At this point, every node will contain a vector of predictions corresponding to the evaluation of that subtree. In the next step, we traverse the tree again [1] hashing these vectors into the binary string key. Finally, if this key contains any element in the hash table and the closest value to this subtree is within a threshold, we replace this subtree with the smallest tree in this table entry. If there is no entry for this key, we create a new entry with this subtree. Figure 2 depicts the proposed algorithm to use LSH to simplify symbolic expressions.

We force every constant vector to have the same hash by setting the prediction to zero if the variance of the vector is zero. This avoids growing the hash table unnecessarily. The threshold also alleviates the fact that nonlinear equations can have non-unique solutions, so they are considered equivalent as long as the predictions are within the threshold. Algorithm 1 describes the initialization of the table, and Algorithm 2 describes the simplification process.

When simplifying the expression, we can traverse the tree either top-down or bottom-up since we apply the replacements on the fly. The order of traversal can generate different simplifications. The advantage of the top-down approach is that it prunes large subtrees first (if it finds a hash entry in the first levels). This may lead to fewer nodes being visited as it may stop at an earlier level. On the other hand, the bottom-up traversal may require more steps, as a node simplified at the bottom level may trigger a new simplification at

[1] In practice, we only need to traverse the tree once.

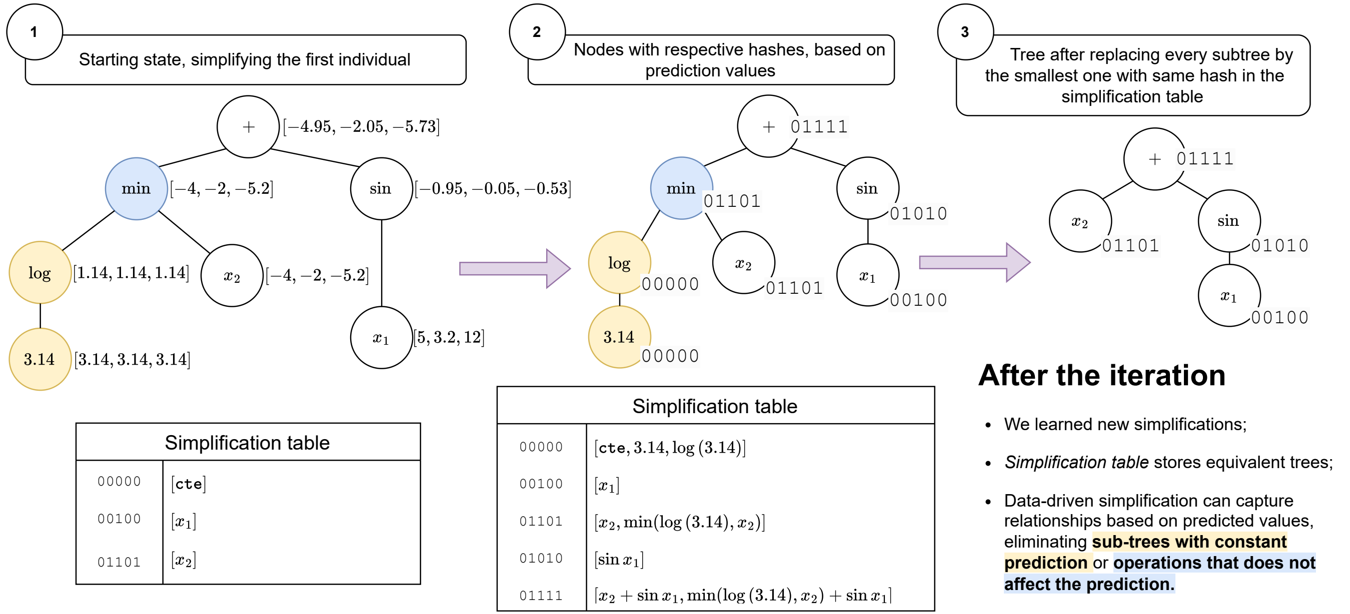

Figure 2: In the first stage (1), we have a simplification table with only the problem variables plus the constant. Every node is seen as the root of a subtree and can generate a prediction vector. The second stage (2) uses the predictions to get hash values for each node, updating the simplification table. Finally, we get the simplified tree by replacing the nodes with the smallest subtree of the same hash in the simplification table.

Algorithm 1 initialize_table

Require: Training data points $(\mathcal{X}^t, \mathrm{y}^t)$, single node constant tree `cte`
Ensure: simplification table `st`

1: `st` $\leftarrow$ empty mapping of (key : value)
2: **for** `terminal` $\in$ {`cte`} $\cup \{x \in \mathcal{X}^t\}$ **do**
3: `pred` $\leftarrow$ `terminal.predict`$(\mathcal{X}^t)$
4: `lhs.index(pred)`
5: `hash`, $d \leftarrow$ `lhs.query(pred)`
6: `st[hash] = [terminal]`
7: **return** `st`

Algorithm 2 hash_simplify

Require: individual `ind`, simplification table `st`, lhs instance `lhs`, tolerance τ
Ensure: simplified version of the individual n

1: **for** `subtree` $\in$ `ind` **do**
2: `pred` $\leftarrow$ `subtree.predict`$(\mathcal{X})$
3: **if** Var(`pred`) = 0 **then**
4: `pred` $\leftarrow$ `pred` $\times$ 0.0
5: `hash`, $d \leftarrow$ `lhs.query(pred)`
6: **if** `hash` $\in$ `st` and $d \leq \tau$ **then**
7: `st[hash]` $\leftarrow$ `st[hash]` $\cup$ {`subtree`}
8: `subtree` $\leftarrow$ argmin size(`tree`) for `tree` $\in$ `st[hash]`
9: **else if** `hash` $\notin$ `st` **then**
10: `lhs.index(pred)`
11: `st[hash] = [subtree]`
12: **return** `ind`

the upper level. Nevertheless, this fine-grained simplification may lead to larger simplifications after simplifying the smaller branches.

This simplification will not always return an algebraically equivalent expression but rather an approximation. By imposing a maximum distance threshold, we mitigate this issue by guaranteeing that this procedure will replace subtrees with similar semantics. This is an advantage for practical applications since we favor simpler expressions at the expense of minor differences in the predictions.

5 METHODS

We implemented a standard evolutionary algorithm using the DEAP framework [10]. The individuals are randomly initialized using the PTC2 method [22]. Then, iteratively, we perform a fixed number of generations of tournament selection with a tournament size of 3. Possible variations operators are the crossover and mutations *insert node*, *remove node*, *replace node*, and *replace subtree*.

Three variants were implemented: *without simplify*; *bottom up*, and *top down*. Simplification is done on the initial population and every offspring generated by the variation operators — this way, we guarantee that every individual is simplified at least once. The individuals also go into the nonlinear parameter optimization method Levenberg-Marquardt [21, 25] algorithm using Scipy [28], which was shown to be effective in previous SR algorithms [2, 14, 29]. After simplifying, we repeat the parameter optimization.

Our hash method is probabilistic (due to plane initialization); thus, the hash size is set to avoid collisions. We tried to increase the hash size by powers of two until the simplification table would not have any collision at its initialization step. During initial experiments, we found that if the initial table has a hash collision for the features or constant, the simplification may replace features with a constant, which is undesirable unless a feature is almost constant.

The threshold was set to a fixed value based on the smallest train MSE error in preliminary experiments over all datasets and set to 0.01, so it is one order of magnitude below all the train errors. Population size and number of generations were set so most runs would finish under a 3 wall clock hours. Table 1 describes the hyper-parameters used in the experiments.

Table 1: Symbolic regression algorithms hyper-parameters.

Parameter	Value
pop size (S)	80
max gen (G)	200
max depth ($\max_d$)	7
max size ($\max_s$)	128 (2^7)
tolerance (τ)	$1e-2$
hash_len	256 bits
probabilities	1/5 for each variation operator
objectives	[error (MSE), size (# nodes))]
Function set	[$+, -, *, \div, \lvert\cdot\rvert, \cos^{-1}, \sin^{-1}, \tan^{-1}$, $\cos, \sin, \tan, e^{(\cdot)}$, `min`, `max`, $\log, \log(1+\cdot), \exp(1+\cdot), \sqrt{\lvert\cdot\rvert}, (\cdot)^2$]

We used a selection of datasets to analyze the effect of simplification throughout the evolutionary process. Table 2 shows the name and dimensionality of the datasets used. We should note that our approach is agnostic to GP implementation and the choice of LSH. We have chosen the basic implementations to highlight the effects of such simplifications.

Table 2: Dimensionality of the six datasets used to perform an in-depth analysis.

Dataset	# samples	# features
Airfoil	1503	5
Concrete	1030	8
Energy Cooling	768	8
Energy Heating	768	8
Housing	506	13
Yacht	308	6

The loss function is the mean squared error (MSE) between the predictions $\widehat{\mathbf{y}} = f(\mathcal{X})$ and observed values $\mathbf{y}$:

$$\text{MSE}(\widehat{\mathbf{y}}, \mathbf{y}) = \frac{1}{d}\sum_{i=1}^{d}(\widehat{y}_i - y_i)^2. \tag{1}$$

As a measurement of simplicity, we also use the concept of complexity by La Cava et al. [19], defined for a node n with k arguments as the recursive combination of complexity of children with its root/head node:

$$C(n) = c_n * (\sum_{a=1}^{k} C(a)). \tag{2}$$

By assigning high complexity values to hard-to-interpret operations, we can measure —recursively— how this propagates through the tree. The complexity for the operators is described in Table 3.

Table 3: Complexity of each operator.

Complexity	Operators
2	$+, -$, `cte`
3	$*$, `max`, `min`, $(\cdot)^2$, $\lvert\cdot\rvert$
4	$\div$, $\sqrt{\lvert\cdot\rvert}$, $e^{(\cdot)}$
5	$\exp(1+\cdot)$, cos, sin, tan
6	$\cos^{-1}$, $\sin^{-1}$, $\tan^{-1}$
8	$\log(1+\cdot)$
9	log

Each method was run 30 times with different split seeds for each dataset on the same hardware. The data was divided into three partitions: 50% as train (visible to the algorithm to perform the parameter and function optimization); 25% as validation (used to assess the loss during the evolution, but not used during train); and 25% test (held-out data used to obtain the final values for the experiments). The validation split is also used to pick the final model returned by the algorithm.

Statistical comparisons, when reported, use the non-parametric Wilcoxon test with Holm-Bonferroni correction, and all comparisons made are explicitly depicted in the figures. Table 4 shows the annotations used to show statistical significance in the plots. When an annotation is shown as "* (ns)", it shows a statistical significance of one asterisk, but after correction, it becomes non-significant.

Table 4: p-value annotations and their correspondences, after applying the alpha correction.

Annotation	p-value
ns	$5\times10^{-2} < p \leq 1.0$
*	$1\times10^{-2} < p \leq 5\times10^{-2}$
**	$1\times10^{-3} < p \leq 1\times10^{-2}$
***	$1\times10^{-4} < p \leq 1\times10^{-3}$
****	$p \leq 1\times10^{-4}$

All data and the source code for implementations, experiments, and post-processing analysis are available at https://github.com/gAldeia/hashing-symbolic-expressions.

6 RESULTS AND DISCUSSION

In this section we analyze different aspects of the experiment's results. First, we verify whether the simplification accelerates the convergence of GP to a good local optimum; next, we compare the number of simplifications performed by each traversal strategy; we show the distribution of the average size, complexity, and goodness-of-fit of the final solutions. We also show the relative

difference of the solutions under these same criteria when paired with the individual runs. Finally, we compare the differences in runtime and explore some hand-picked simplification rules generated throughout the runs.

6.1 Convergence

The data was split into training and validation sets. While the model used the training partition to perform parameter and model optimization, the validation partition was used to compute the MSE with unseen data. Figure 3 reports the minimum validation error during the evolution, estimated using the mean of 30 runs.

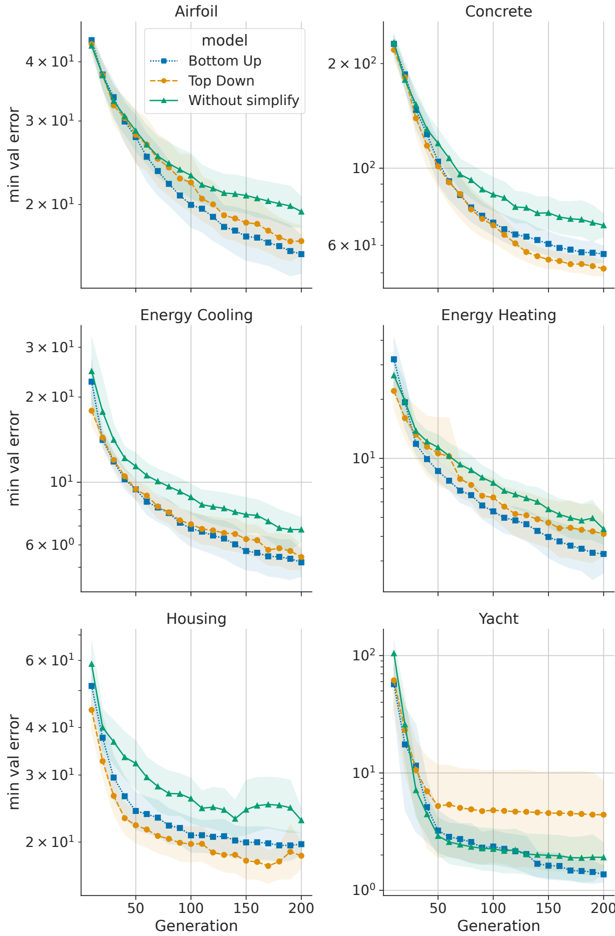

Figure 3: Validation error of the best individual during the evolution. y axis is in log-scale.

These results show that the versions with simplification often outperform the traditional GP throughout the run, leading to a better local optimum. The only exception is the Yacht dataset, in which the top-down strategy performs worse than the bottom-up and the version without simplification. In every other dataset, both traversal strategies present similar performance.

The two traversal strategies, as conjectured before, perform a different number of simplifications during the evolutionary process, as we can see in Figure 4. The bottom-up strategy performs around 50% more simplifications with a little intersection in the estimated confidence interval. We believe that the number of simplifications appears to diminish in later generations as we effectively remove bloated and redundant subtrees; however, we also start to have more specialized and complex models, which are harder to simplify.

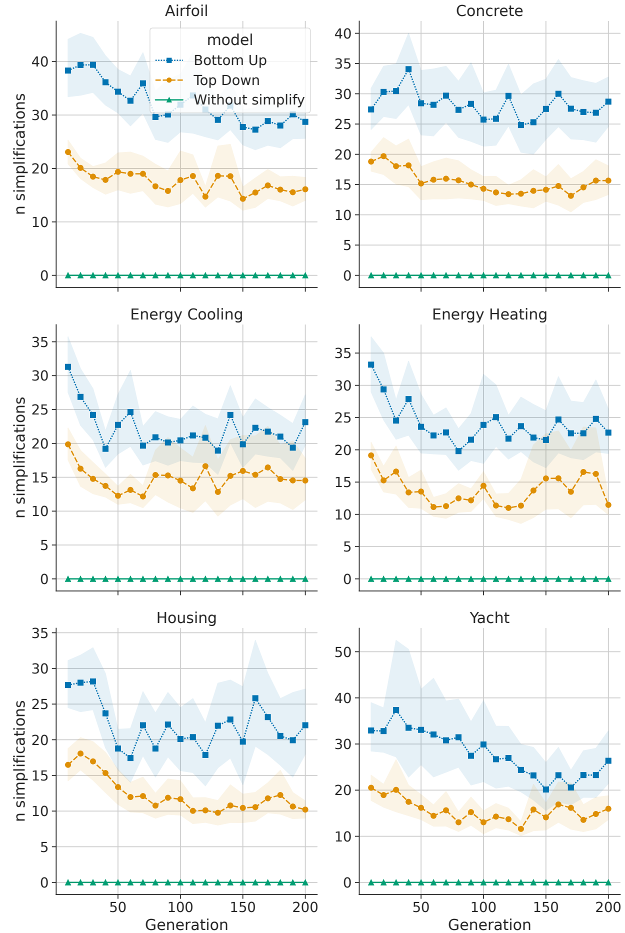

Figure 4: Number of simplifications performed in each generation.

6.2 Goodness-of-fit and size trade-off

At the end of the run, every algorithm picks the final model with the best performance on the validation split. Figures 5, 6, and 7 report the size, complexity, and MSE on the test partition, respectively.

Regarding the size, we cannot reject the null hypothesis that there are not differences between the approaches — a counter-intuitive result, as the simplification is expected to reduce the size of the expressions. However, looking at the complexity boxplots, we observe a smaller variation (for the best) using simplification strategies and the rejection of the null hypothesis with a p-value between $[10^{-3}, 10^{-2}]$. As the models are simplified during the evolution, it

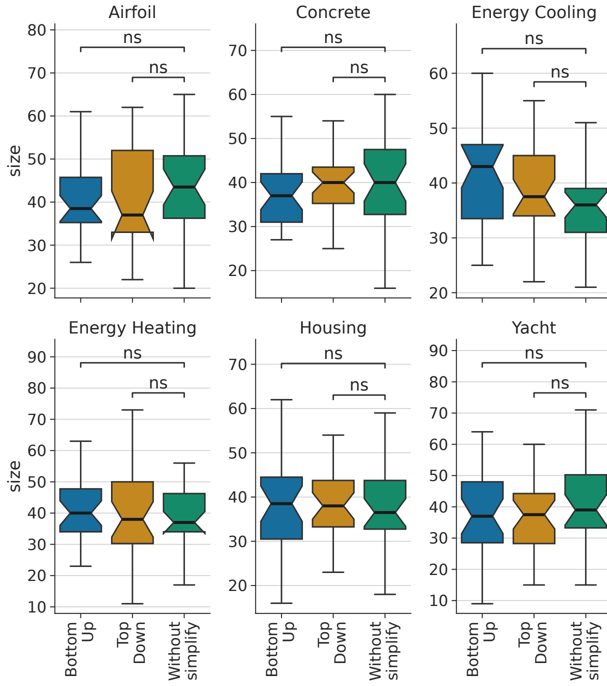

Figure 5: Final size of the solutions found by each method. The simplification methods showed no statistically different significance related to without any simplification.

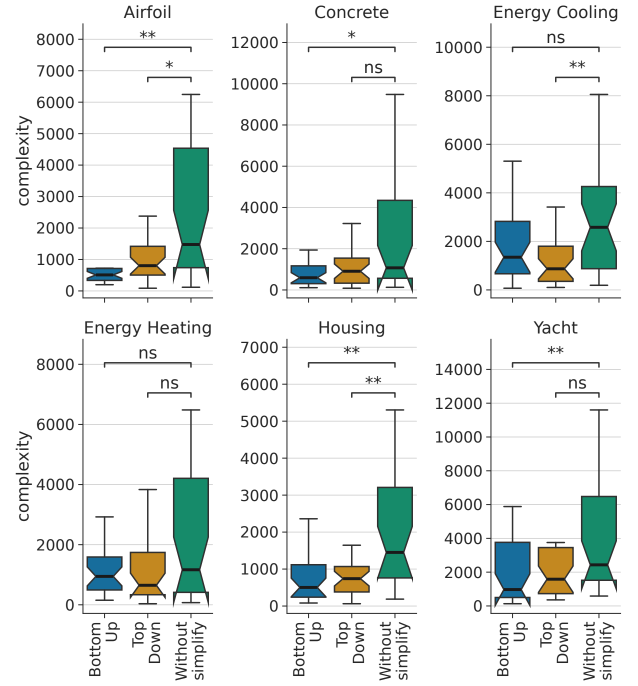

Figure 6: Final complexity of the solutions. There are several cases where simplification can improve complexity.

frees up space (within the limit of maximum nodes) to accommodate more useful sub-expressions. As a by-product, the generated expressions are also less complex when considering the definition of recursive complexity. We also hypothesize that the simplification opens up space for further improving the expressions, and the evolutionary process efficiently takes care of that by generating expressions of the same size but with better performance.

We find differences in complexity for the datasets Airfoil, Concrete, Energy Cooling, Housing, and Yacht. Even though they are equivalent in size, the simplification process led to finding solutions with better complexity without explicitly trying to minimize it. Note that the simplified expression does not exhibit higher errors than the original. Regarding the test set MSE, Airfoil, Concrete, and Energy Cooling datasets show statistically significant improvements in the bottom-up strategy. The top-down strategy shows improvements only for Concrete.

The simplification strategy shows only small differences and no clear better option in terms of the results, except for the number of simplifications. The bottom-up strategy will always iterate through the entire tree, while the top-down strategy can cut large branches right away, ending up with a smaller number of simplifications.

We believe that Yacht was the most challenging dataset due to the test MSE error scale shown in Fig. 7, as it is closer to the threshold than any other algorithm. Parameter optimization, especially the inexact simplification threshold, seems important to be adjusted based on the error scale, and we plan to investigate it further.

6.3 Relative change

Even though the differences in MSE seem modest, those are considering the mean and median of the distribution. A better way to verify the benefits of the simplification strategy is to pair the 30 runs based on their seed and calculate the percentage of variation between the simplification methods and without any simplification (i.e., the baseline). This way, the variation between datasets can be aggregated, and an overall measurement and a one-sided t-test can be applied to verify whether the mean of the relative change is different from zero. Figure 8 reports the variations for the bottom-up and top-down strategies.

A t-test for the mean of the distributions was performed for each subfigure with 180 degrees of freedom (6 datasets and 30 runs each).

Regarding size, we see the distribution with a median of 2.43% for bottom-up and −2.13% for top-down, but both presented a p-value greater than 0.05, so we cannot reject the null hypothesis. Complexity shows a median of −104.06% and −111.34%, with p-values of 2.94×10^{-7} and 1.59×10^{-6} for bottom-up and top-down, respectively. This indicates that for each individual run, we observe a reduction in complexity compared to the baseline. For MSE, the medians are −18.21% and −20.32%, and the p-values are 8.165×10^{-7} and 1.85×10^{-5}. Again, we can reject the null hypothesis and conclude the simplification reduces the error by 20% on average.

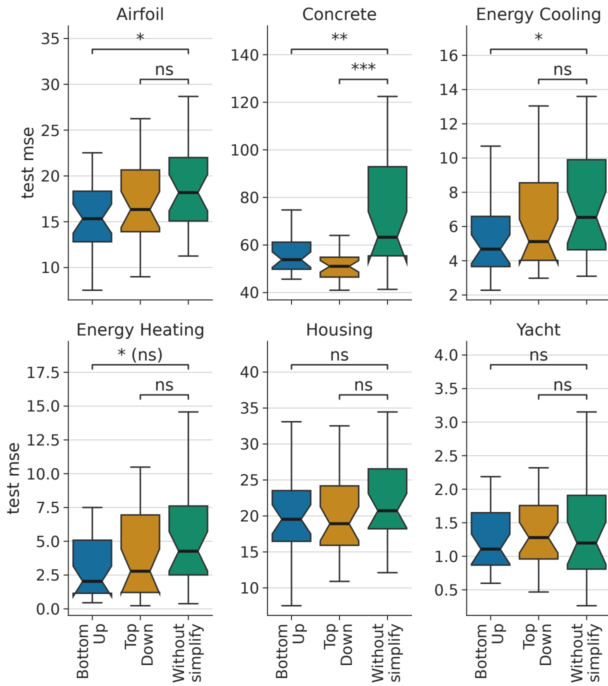

Figure 7: MSE on the test partition. Simplification led to better performance in error, with less complex models.

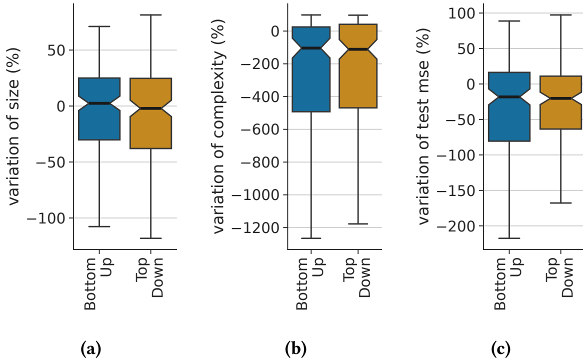

Figure 8: Percentage variation of size (8a), complexity (8b) and MSE on test (8c) compared to without any simplification.

6.4 Analysis of hashes and individuals

We performed a single run on the Yacht dataset to obtain a few insights on how simplification is replacing nodes. We chose it because it has no statistically significant differences in size and error and is also low-dimensional (for the sake of the example). At the end of the run, we had a total of 5144 hash entries created from 7324 expressions. This subsection will review some of the entries of the simplification table. An entry is a tuple of a hash and a list of equivalent trees. The hash will be truncated to simplify the discussion, and expressions will be shown as string-formatted versions of the expressions, ordered by smallest to largest — meaning that any of the subtrees would be replaced by the first.

Let us take a look into the first case:

```
1010110111001101...
 - square(x_5)
 - multiply(x_5, x_5)
 - absolute(square(x_5))
 - maximum(square(x_5), x_0)
 - maximum(add(-15.455, x_1), square(x_5))
```

We first notice that, without explicit rules, our method started to replace `multiply(x_5, x_5)` by `square(x_5)`, a shorter representation. It also learned that the absolute value of the square is always positive, so `absolute(square(x_5))` could also be simplified to `square(x_5)`. Some simplifications there are based on data: that x_5^2 always dominates x_0 in the `max` function. Algebraic rules would not capture this simplification.

Some other interesting cases arise. Here, we have a permutation of arguments on a 4-ary commutative operation, as well as the chain of commutative operations:

```
1110001011011111...
 - multiply(x_1, x_7, x_0, x_4)
 - multiply(x_0, x_4, x_1, x_7)
 - multiply(x_1, x_0, multiply(x_4, x_7))
```

Finally, some other cases are the simplification of redundant operations (as in `minimum(x_2, x_2)`), the usage of the identity value of an operation as argument (as in `add(0.0, x_2)`), and the chaining of f with its f^{-1} (as in `log(exp(x_2))`):

```
1010101111001001...
 - x_2
 - absolute(x_2)
 - minimum(x_2, x_2)
 - minimum(x_2, x_6)
 - minimum(x_2, x_5)
 - add(0.0, x_2)
 - log(exp(x_2))
 - sqrtabs(square(x_2))
 - maximum(-523.249, x_2)
```

Figure 9 shows the smallest expression found by the bottom-up strategy and with GP without simplification for the Yacht dataset.

The bottom-up model obtained an MSE on the test set of 1.98 with 9 nodes, and GP without simplification obtained a test error of 2.08 using 15 nodes. We can see more complex constructs when not applying simplification, as well as the chaining of $\sqrt{\cdot}$ and $(\cdot)^2$.

The yacht dataset has errors that are one order of magnitude greater than the simplification threshold, and besides not showing improvements in error, it had improvements in complexity. By analyzing one single run, we proved that the benefits of our simplification methods extend beyond size and error. The method was able to minimize complexity without being explicitly told to do so.

6.5 Execution time

Figure 10 reports the execution time for each algorithm.

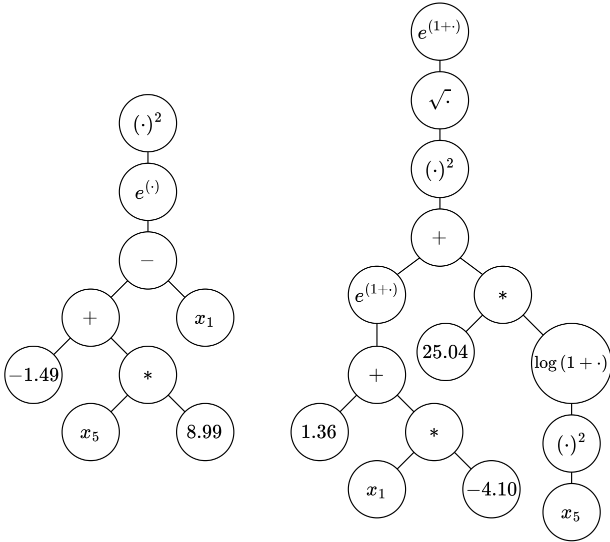

Figure 9: Smallest expression found by bottom-up (left) and without simplification (right) for the Yacht dataset. The MSE for each expression was 1.98 using the bottom-up simplification and 2.08 without simplification.

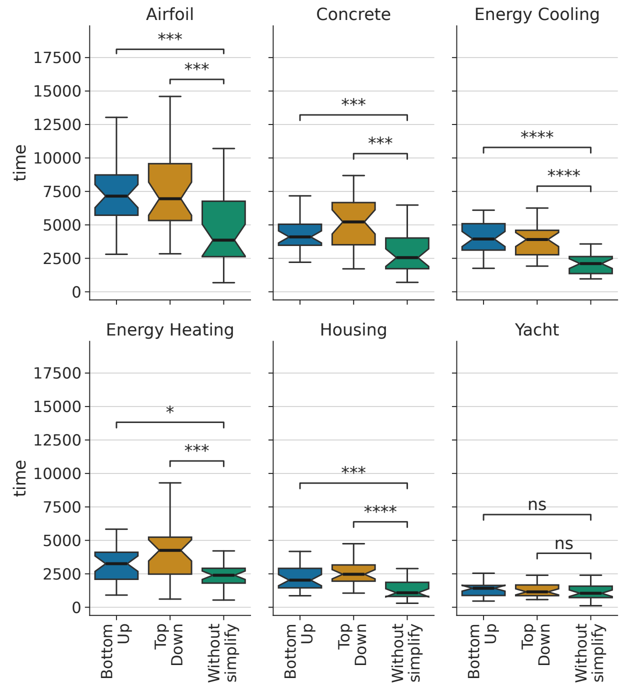

Figure 10: Variation of execution time (in seconds) for each method. The subplots are sharing the y axis.

We notice that the simplification methods take more time to run than without any simplification, with statistical differences for all but the last dataset. This happens because we perform a nonlinear optimization twice for every individual, as they are re-optimized after the simplification, indicating that the method would benefit from caching recent information to avoid re-optimizing the entire tree after every simplification.

7 CONCLUSIONS

In this paper, we propose an inexact simplification method using LSH to learn substitution rules that are either well-known algebraic identities or simplifications specific to the dataset domain. LSH efficiently stores and queries expressions that behave similarly in the phenotypical space, enabling us to apply the simplification process for every sampled expression throughout the evolution.

With this approach, we can experimentally analyze the influence of simplification and bloat control on the evolutionary process. The empirical results showed that, on average, the versions with simplification returned more accurate expressions of the same size but with less complex construct. Comparing runs departing from the same seed, we observed a median of 20% reduction in the mean squared error when using simplification.

This approach has some drawbacks. For example, the hash function can only guarantee that similar expressions are clustered with high probability. We can eventually include expressions with distinct behavior in the wrong cluster, but this should be rare. Another issue is that the simplification is inexact, so it can perform substitutions that slightly change the behavior of the function or that are exact only on the training data.

On the other hand, since this is a data-driven approach, it can learn the rules on the fly without any need to pre-determine the algebraic identities. It can also learn rules specific to the dataset (e.g., x_1 is always smaller than x_2).

As for the next steps, we intend to improve the influence of simplification by making a "warm-up" step, where random individuals are generated and used to initialize the table; we will also evaluate the influence of performing simplification sporadically, only for a selection of individuals, or only at the final generation; and how does the threshold affect the results. We will also test some other hash functions with better guarantees than SimHash. Finally, we will integrate this approach into a high-performance GP implementation to make it possible to run more detailed tests, also including other simplification methods into the benchmark.

ACKNOWLEDGMENTS

W.G.L. was supported by National Institutes of Health (NIH) grant R00-LM012926, and Patient Centered Outcomes Research Institute (PCORI) ME-2020C1D-19393. F.O.F. is supported by Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) grant 2021/12706-1, and Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) grant 301596/2022-0. G.S.I.A. is supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) finance Code 001 and grant 88887.802848/2023-00.

REFERENCES

[1] Michael Affenzeller, Stephan M Winkler, Gabriel Kronberger, Michael Kommenda, Bogdan Burlacu, and Stefan Wagner. 2014. Gaining deeper insights

in symbolic regression. *Genetic Programming Theory and Practice XI* (2014), 175–190.

[2] Guilherme Seidyo Imai Aldeia and Fabrício Olivetti de França. 2022. Interaction-Transformation Evolutionary Algorithm with Coefficients Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Boston, Massachusetts) *(GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 2274–2281. https://doi.org/10.1145/3520304.3533987

[3] Dimitrios Angelis, Filippos Sofos, and Theodoros E. Karakasidis. 2023. Artificial Intelligence in Physical Sciences: Symbolic Regression Trends and Perspectives. *Archives of Computational Methods in Engineering* 30, 6 (April 2023), 3845–3865. https://doi.org/10.1007/s11831-023-09922-z

[4] G. F. Bomarito, P. E. Leser, N. C. M. Strauss, K. M. Garbrecht, and J. D. Hochhalter. 2022. Bayesian model selection for reducing bloat and overfitting in genetic programming for symbolic regression. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, Boston Massachusetts, 526–529. https://doi.org/10.1145/3520304.3528899

[5] Bogdan Burlacu, Lukas Kammerer, Michael Affenzeller, and Gabriel Kronberger. 2021. Hash-Based Tree Similarity and Simplification in Genetic Programming for Symbolic Regression. https://doi.org/10.1007/2F978-3-030-45093-9_44 arXiv:2107.10640 [cs].

[6] Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. 380–388.

[7] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. 2001. Searching in metric spaces. *ACM Comput. Surv.* 33, 3 (sep 2001), 273–321. https://doi.org/10.1145/502807.502808

[8] F. O. de Franca and G. S. I. Aldeia. 2021. Interaction–Transformation Evolutionary Algorithm for Symbolic Regression. *Evolutionary Computation* 29, 3 (09 2021), 367–390. https://doi.org/10.1162/evco_a_00285 arXiv:https://direct.mit.edu/evco/article-pdf/29/3/367/1959462/evco_a_00285.pdf

[9] Fabricio Olivetti de Franca and Gabriel Kronberger. 2023. Reducing Overparameterization of Symbolic Regression Models with Equality Saturation. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Lisbon, Portugal) *(GECCO '23)*. Association for Computing Machinery, New York, NY, USA, 1064–1072. https://doi.org/10.1145/3583131.3590346

[10] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research* 13 (jul 2012), 2171–2175.

[11] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. [n. d.]. Similarity Search in High Dimensions via Hashing. ([n. d.]).

[12] Thomas Helmuth, Nicholas Freitag McPhee, Edward Pantridge, and Lee Spector. 2017. Improving generalization of evolved programs through automatic simplification. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, Berlin Germany, 937–944. https://doi.org/10.1145/3071178.3071330

[13] Omid Jafari, Preeti Maurya, Parth Nagarkar, Khandker Mushfiqul Islam, and Chidambaram Crushev. 2021. A Survey on Locality Sensitive Hashing Algorithms and their Applications. http://arxiv.org/abs/2102.08942 arXiv:2102.08942 [cs].

[14] Michael Kommenda, Bogdan Burlacu, Gabriel Kronberger, and Michael Affenzeller. 2019. Parameter identification for symbolic regression using nonlinear least squares. *Genetic Programming and Evolvable Machines* 21, 3 (Dec. 2019), 471–501. https://doi.org/10.1007/s10710-019-09371-3

[15] John R Koza. 1994. Genetic programming as a means for programming computers by natural selection. *Statistics and computing* 4 (1994), 87–112.

[16] Kulunchakov. 2017. Creation of parametric rules to rewrite algebraic expressions in Symbolic Regression. *Machine Learning and Data Analysis* 3, 1 (2017), 6–19. https://doi.org/10.21469/22233792.3.1.01

[17] William La Cava, Kourosh Danai, Lee Spector, Paul Fleming, Alan Wright, and Matthew Lackner. 2016. Automatic identification of wind turbine models using evolutionary multiobjective optimization. *Renewable Energy* 87 (2016), 892–902. https://doi.org/10.1016/j.renene.2015.09.068 Optimization Methods in Renewable Energy Systems Design.

[18] William La Cava, Patryk Orzechowski, Bogdan Burlacu, Fabricio de França, Marco Virgolin, Ying Jin, Michael Kommenda, and Jason Moore. 2021. Contemporary Symbolic Regression Methods and their Relative Performance. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, J. Vanschoren and S. Yeung (Eds.), Vol. 1. Curran.

[19] William La Cava, Tilak Raj Singh, James Taggart, Srinivas Suri, and Jason H. Moore. 2019. Learning concise representations for regression by evolving networks of trees. http://arxiv.org/abs/1807.00981 arXiv:1807.00981 [cs].

[20] William G. La Cava, Paul C. Lee, Imran Ajmal, Xiruo Ding, Priyanka Solanki, Jordana B. Cohen, Jason H. Moore, and Daniel S. Herman. 2023. A flexible symbolic regression method for constructing interpretable clinical prediction models. *npj Digital Medicine* 6, 1 (June 2023), 107. https://doi.org/10.1038/s41746-023-00833-8

[21] Kenneth Levenberg. 1944. A method for the solution of certain non-linear problems in least squares. *Quarterly of applied mathematics* 2, 2 (1944), 164–168.

[22] S. Luke. 2000. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation* 4, 3 (2000), 274–283. https://doi.org/10.1109/4235.873237

[23] Sean Luke and Liviu Panait. 2006. A Comparison of Bloat Control Methods for Genetic Programming. *Evolutionary Computation* 14, 3 (Sept. 2006), 309–344. https://doi.org/10.1162/evco.2006.14.3.309

[24] Jiayi Luo and Cindy Long Yu. 2023. The Application of Symbolic Regression on Identifying Implied Volatility Surface. *Mathematics* 11, 9 (2023). https://doi.org/10.3390/math11092108

[25] Donald W Marquardt. 1963. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the society for Industrial and Applied Mathematics* 11, 2 (1963), 431–441.

[26] Quang Uy Nguyen and Thi Huong Chu. 2020. Semantic approximation for reducing code bloat in Genetic Programming. *Swarm and Evolutionary Computation* 58 (Nov. 2020), 100729. https://doi.org/10.1016/j.swevo.2020.100729

[27] Marco Virgolin and Solon P. Pissis. 2022. Symbolic Regression is NP-hard. (2022). https://doi.org/10.48550/ARXIV.2207.01018 Publisher: arXiv Version Number: 3.

[28] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[29] Tony Worm and Kenneth Chiu. 2013. Prioritized Grammar Enumeration: Symbolic Regression by Dynamic Programming. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation* (Amsterdam, The Netherlands) *(GECCO '13)*. Association for Computing Machinery, New York, NY, USA, 1021–1028. https://doi.org/10.1145/2463372.2463486