LLM-Meta-SR: In-Context Learning for Evolving Selection Operators in Symbolic Regression

Hengzhe Zhang[1], **Qi Chen**[1], **Bing Xue**[1], **Wolfgang Banzhaf**[2], **Mengjie Zhang**[1]
[1]Centre for Data Science and Artificial Intelligence & School of Engineering and Computer Science,
Victoria University of Wellington, PO Box 600, Wellington 6140, New Zealand
[2]Department of Computer Science and Engineering,
Michigan State University, East Lansing, MI 48824, USA
{hengzhe.zhang, qi.chen, bing.xue, mengjie.zhang}@ecs.vuw.ac.nz, banzhafw@msu.edu

Abstract

Large language models (LLMs) have revolutionized algorithm development, yet their application in symbolic regression, where algorithms automatically discover symbolic expressions from data, remains constrained and is typically designed manually by human experts. In this paper, we propose a meta learning framework that enables LLMs to automatically design selection operators for evolutionary symbolic regression algorithms. We first identify two key limitations in existing LLM-based algorithm evolution techniques: a lack of semantic guidance and code bloat. The absence of semantic awareness can lead to ineffective exchange of useful code components, and bloat results in unnecessarily complex components, both of which can reduce the interpretability of the designed algorithm or hinder evolutionary learning progress. To address these issues, we enhance the LLM-based evolution framework for meta symbolic regression with two key innovations: a complementary, semantics-aware selection operator and bloat control. Additionally, we embed domain knowledge into the prompt, enabling the LLM to generate more effective and contextually relevant selection operators. Our experimental results on symbolic regression benchmarks show that LLMs can devise selection operators that outperform nine expert-designed baselines, achieving state-of-the-art performance. Moreover, the evolved operator can further improve the state-of-the-art symbolic regression algorithm, achieving the best performance among 26 symbolic regression and machine learning algorithms across 116 regression datasets. This demonstrates that LLMs can exceed expert-level algorithm design for symbolic regression.

1 Introduction

Symbolic regression (SR) is the task of discovering mathematical expressions that accurately model a given dataset. It is widely used in domains such as finance (Shi et al. 2025), education (Shen et al. 2024),chemistry (Chen et al. 2025), and physics (Feng et al. 2025), due to its interpretability and ability to uncover underlying relationships. Formally, given a dataset $\mathcal{D} = \{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{N}$, where $\mathbf{x}^{(i)} \in \mathbb{R}^d$ are input features and $y^{(i)} \in \mathbb{R}$ are target values, the goal is to find a symbolic expression $f^* \in \mathcal{F}$ from a space of candidate expressions $\mathcal{F}$ that minimizes a loss function $\mathcal{L}$:

$$f^* = \arg\min_{f \in \mathcal{F}} \mathcal{L}(f(\mathbf{x}), y). \quad (1)$$

Many SR algorithms have been proposed, including evolutionary (Jiang and Xue 2024; Fong and Motani 2024b), neural (Biggio et al. 2021; Kamienny et al. 2022), and search-based (Kahlmeyer, Fischer, and Giesen 2025; Yu et al. 2025) approaches. In recent years, there has been growing interest in hybrid methods that combine the flexibility of evolutionary algorithms with the learning capabilities of neural networks (Landajuela et al. 2022; Grayeli et al. 2024; Zhang et al. 2025).

In both evolutionary (Jiang and Xue 2024; Fong and Motani 2024b) and neural-evolutionary (Landajuela et al. 2022; Grayeli et al. 2024; Zhang et al. 2025) symbolic regression algorithms, the system follows an iterative optimization process in which solutions are progressively refined by selecting promising candidates. In this process, the selection operator responsible for identifying candidate solutions worth refining, plays a key role in system effectiveness. However, current selection operators are typically manually designed by experts, requiring significant effort and trial-and-error, which can limit the pace of progress. Ideally, an automated algorithm design framework capable of generating effective selection operators for symbolic regression would substantially enhance research and development efficiency in this area.

Large language models (LLMs) have recently emerged as powerful tools for automated heuristic design in various optimization tasks (Romera-Paredes et al. 2024; Liu et al. 2024a; Ye et al. 2024). However, most existing frameworks primarily focus on generating heuristic rules for specific combinatorial optimization tasks. In this paper, we pursue a more ambitious goal: to automatically design a core algorithmic component—specifically, the selection operator—in evolutionary symbolic regression, with the aim of achieving strong performance across a wide range of regression tasks. While frameworks such as FunSearch (Romera-Paredes et al. 2024), EoH (Liu et al. 2024a), and ReEvo (Ye et al. 2024) lay groundwork for algorithm evolution, two challenges remain underexplored in LLM-driven algorithm evolution. The first is the *underutilization of semantic information* in the generated code. Here, we define semantic information as the performance of an algorithm on individual task instances. Typically, only the average performance across all training instances is provided to the LLM to guide code generation (Huang et al. 2025a), which overlooks fine-grained behavior. As illustrated

Figure 1: Illustration of the importance of fine-grained semantic differences between solutions during algorithm evolution.

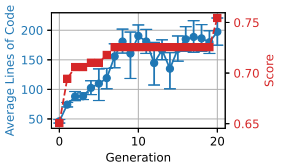

Figure 2: Average code length of solutions in the population over generations without bloat control, and the score of the best solution.

in Figure 1, consider a scenario with three algorithms: the first two perform well on the first dataset but poorly on the second, while the third shows the opposite pattern. Despite their contrasting behaviors, all three algorithms may exhibit similar average performance. However, presenting the first two algorithms together to the LLM offers limited benefit, as they exhibit redundant patterns—both succeeding and failing on the same datasets. In contrast, combining the first and third algorithms allows the LLM to integrate complementary strengths, potentially generating an algorithm that performs well on both datasets. Thus, incorporating semantic information is crucial for effective algorithm evolution.

The second challenge is *complexity bias*, illustrated in Figure 2. LLMs tend to generate overly long or intricate code during optimization, similar to code bloat in genetic programming (Banzhaf et al. 1998). This results in the accumulation of redundant or non-functional logic, which impairs interpretability, wastes a large number of tokens, and slows optimization. As shown in Figure 2, such complexity can lead to performance stagnation, ultimately limiting the effectiveness of evolution.

In this paper, we propose an LLM-driven method for meta symbolic regression to automatically design selection operators [1]. The main contributions are as follows:

- We propose an LLM-driven meta symbolic regression framework to automatically design selection operators using in-context learning. The LLM learns from design and evaluation history to automatically discover generalizable operators that consistently outperform human-designed counterparts across a wide range of unseen datasets.
- We identify the issue of bloat in LLM-based code generation and introduce a bloat control strategy that improves both the interpretability of the evolved code and the effectiveness of the evolutionary process.
- We propose semantic-based feedback and complementary selection mechanisms to fully leverage semantic information during LLM-driven generation, explicitly guiding algorithm evolution by integrating effective building blocks and enhancing learning performance.
- We design ideal properties of selection operators based on domain knowledge to craft more effective prompts that guide LLMs in generating high-quality selection operators.

[1] Source Code: https://anonymous.4open.science/r/LLM-Meta-SR/

2 Background and Related Work

2.1 Selection Operators in Symbolic Regression

There has been significant research on designing selection operators for symbolic regression, including tournament selection (Xie and Zhang 2012), Boltzmann sampling (Shojaee et al. 2025; Romera-Paredes et al. 2024), lexicase selection (La Cava et al. 2019), novelty search (Lehman and Stanley 2010), and complementary phenotype selection (Dolin, Arenas, and Merelo 2002). These operators aim to improve symbolic regression from different perspectives. Lexicase selection highlights the importance of specialists on subsets of training instances (La Cava et al. 2019). Novelty search encourages behavioral diversity to escape local optima (Lehman and Stanley 2010). Complementary phenotype selection focuses on leveraging complementary individuals in crossover and using this information during the selection process (Dolin, Arenas, and Merelo 2002). Despite these advances, there remains a lack of selection operators that simultaneously capture multiple desirable traits, largely because designing such operators remains labor-intensive and trial-and-error-driven.

2.2 LLMs for Symbolic Regression

Early work on LLMs for symbolic regression focused on specialized language models (Kamienny et al. 2022; Biggio et al. 2021), often assisted by Monte Carlo Tree Search (Kamienny et al. 2023; Shojaee et al. 2024) or evolutionary algorithms (Zhang et al. 2025). With the advent of general-purpose LLMs, their use in symbolic regression has attracted increasing attention (Shojaee et al. 2025), either through the FunSearch framework (Romera-Paredes et al. 2024) or via integration with evolutionary algorithm frameworks (Grayeli et al. 2024). In these studies, LLMs effectively replace traditional crossover and mutation operators to generate candidate solutions, but selection operators are still manually designed. This motivates the exploration of automated selection operator design using LLMs.

Some recent work incorporates dataset descriptions into symbolic regression, referring to this as "semantics" (Liu, Huynh, and van der Schaar 2025). However, this differs from the concept of semantics in genetic programming (Moraglio, Krawiec, and Johnson 2012), where it refers to the behavior of a program. In LLM-based program evolution, the fine-grained behavior of candidate programs is often overlooked and reduced to aggregate scores, which can obscure meaningful information and hinder evolutionary progress.

3 Algorithm Framework

This work introduces a meta learning framework that leverages large language models (LLMs) within a meta-symbolic regression setting to automate the design of selection operators. The key components are as follows.

3.1 Solution Representation

Each solution is a piece of code representing a selection operator. The input to the selection operator is a list of individuals. Each individual contains a list representing squared error, a list of predicted values, and a list of nodes, which can be used to compute the height and depth of the symbolic tree. The evolutionary status, specifically the ratio between the current and total generations, is also provided. The expected output of the selection operator is a list of promising symbolic trees. The structure of the selection operator is presented in Code 7 of the supplementary material.

3.2 Meta-Evolution Workflow

The meta-evolution algorithm consists of two nested loops: an outer meta-evolution loop and an inner symbolic regression loop. The outer loop generates new selection operators, while the inner loop evaluates their performance. An overview of the workflow is shown in Figure 3.

Let $\mathcal{P}^{(t)} = \{O_1^{(t)}, O_2^{(t)}, \ldots, O_N^{(t)}\}$ denote the population of selection operators at generation t, where N is the population size and each $O_i^{(t)}$ is a selection operator. The fitness of each operator $O_i^{(t)}$, denoted $f(O_i^{(t)})$, is evaluated based on its performance in the symbolic regression loop. The meta-evolution process includes the following components:

- **Population Initialization:** The initial population $\mathcal{P}^{(0)}$ is generated by prompting the LLM to produce N random selection operators. The details of the initialization prompts are provided in Appendix K of the supplementary material.
- **Synthetic Evaluation:** Since LLM-generated code may contain syntax errors or infinite loops, we run code on several basic test cases to filter out invalid operators. Each $O_i^{(t)} \in \mathcal{P}^{(t)}$ is evaluated in a synthetic testing environment and discarded if it contains syntax/runtime errors, exceeds a 300-second execution time, or is more than $100\times$ slower than the fastest operator in the batch. The test cases are provided in Appendix F of the supplementary material.
- **Solution Evaluation:** For each selection operator $O_i^{(t)} \in \mathcal{P}^{(t)}$, its fitness is computed as $f(O_i^{(t)}) = \frac{1}{T}\sum_{j=1}^{T} \texttt{SR}(O_i^{(t)}, \mathcal{D}_j)$, where $\mathcal{D}_1, \ldots, \mathcal{D}_T$ are symbolic regression datasets, and $\texttt{SR}(O_i^{(t)}, \mathcal{D}_j)$ denotes the symbolic regression performance of $O_i^{(t)}$ on dataset $\mathcal{D}_j$. In brief, each dataset is split into a training and validation set. Symbolic regression is performed on the training set, and the fitness of the selection operator is computed based on its performance on the validation set. Details of the solution evaluation procedure are provided in Appendix A of the supplementary material.
- **Survival Selection:** A subset of operators is retained to form the next generation $\mathcal{P}^{(t+1)}$. To control bloat, a multi-objective survival selection strategy is employed, as described in Section 3.4.
- **Solution Selection:** Two parent operators $O_a^{(t)}, O_b^{(t)} \in \mathcal{P}^{(t)}$ are selected for recombination. The selection strategy incorporates semantic information, as detailed in Section 3.3.
- **Solution Generation:** Based on the selected parents, N candidate operators are generated per generation. Of these, $N - M$ are generated via crossover and M via mutation:
  - **Operator Crossover:** For each pair $(O_a^{(t)}, O_b^{(t)})$, a new operator is generated as $O_{\text{new}}^{(t+1)} = \text{LLM\_Crossover}\left(O_a^{(t)}, O_b^{(t)}\right)$. The goal is to generate a new piece of code that combines effective building blocks from the selected parent operators, guided by their performance scores on the evaluated datasets.
  - **Operator Mutation:** Let $O^* = \arg\max_{O_i^{(t)} \in \mathcal{P}^{(t)}} f(O_i^{(t)})$ be the best-performing operator in generation t (Ye et al. 2024). Mutated variants are then generated as $O_{\text{mut}}^{(t+1)} = \text{LLM\_Mutate}(O^*)$, with the objective of generating novel code based on the elite operator.

  The prompts used for crossover and mutation are provided in Appendix K. Solution generation leverages the in-context learning capability of LLMs (Gao and Das 2024), expecting that LLMs can generate better solutions by analyzing historical execution results.

This process continues until a predefined number of generations T_{max} is reached. The operator with the highest fitness value across all generations is selected as the final selection operator.

3.3 Semantics-Aware Evolution

Semantic-based Selection: In the meta-evolution scenario, the goal of the crossover operator is to combine the strengths of two LLM-generated selection operators. As introduced in Section 1, crossover benefits more from combining solutions with complementary strengths than from pairing generalists.

To this end, we propose a semantics-aware selection strategy, with the pseudocode provided in Algorithm 1. Because each selection operator $O_i \in \mathcal{P}^{(t)}$ has been evaluated across d datasets and is associated with a score vector $\mathbf{s}_i = [s_{i,1}, \ldots, s_{i,d}]$, we can compute a complementarity score for each candidate in the population. The process begins by randomly selecting the first parent $O_a \in \mathcal{P}^{(t)}$. Random selection, as used in ReEvo (Ye et al. 2024) and HSEvo (Dat, Doan, and Binh 2025), helps mitigate premature convergence (Dat, Doan, and Binh 2025). Then, for each candidate $O_i \in \mathcal{P}^{(t)}$, we compute a complementarity score:

$$\mu_i = \frac{1}{d}\sum_{j=1}^{d} \max(s_{a,j}, s_{i,j}), \tag{2}$$

which measures the potential combined performance of O_a and O_i across all datasets. The second parent O_b is retrieved as the operator with the highest complementarity score, i.e., $O_b = O_{i^*}$ where $i^* = \arg\max_i \mu_i$. This process can be viewed as retrieval-augmented generation (RAG), where complementarity serves as the similarity function to retrieve the most complementary code from the population to guide generation.

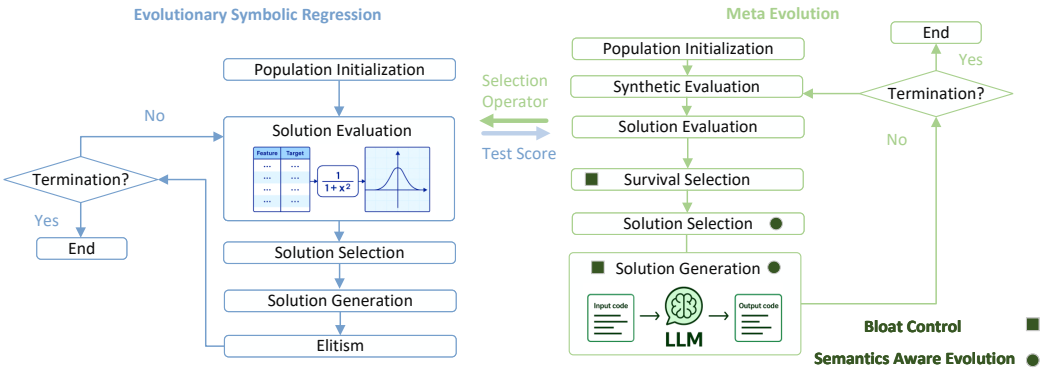

Figure 3: Workflow of LLM-driven selection operator evolution.

Algorithm 1: Semantics-Aware Selection

Require: $\mathcal{P}^{(t)} = \{O_1, \ldots, O_N\}$: population where each O_i has score vector $\mathbf{s}_i$

$O_a \leftarrow$ Random sample from $\mathcal{P}^{(t)}$
for each $O_i \in \mathcal{P}^{(t)}$ **do**
  Compute $\mu_i \leftarrow \frac{1}{d}\sum_{j=1}^{d} \max(s_{a,j}, s_{i,j})$
end for
$i^* \leftarrow \arg\max_i \mu_i$
$O_b \leftarrow O_{i^*}$ // Retrieve Complement Operator
return O_b

Semantic Feedback: The semantic selection strategy ensures that crossover is guided by semantically diverse behaviors. Furthermore, to enhance semantic awareness, we provide the full score vector $\mathbf{s}_i$ of dataset-specific scores to the LLM in the solution generation stage, rather than averaging them into a single aggregate value. This enables the LLM to reason explicitly about behavioral differences across tasks.

3.4 Bloat Control

Prompt-based Length Limit: To mitigate code bloat in the evolution of selection operators, we incorporate a length constraint in the prompt during solution generation. For any operator $O_i \in \mathcal{P}^{(t)}$, its code length is denoted by $\ell(O_i)$, measured as the number of non-empty, non-comment lines in the implementation. We choose to constrain the number of lines, rather than the number of tokens, because line count is less sensitive to variations in variable name length. Rather than enforcing a hard upper bound $\ell(O_i) \leq L_{\max}$, which is difficult to control during LLM generation, we embed the maximum length ℓ_{target} directly into the prompt. For instance, the LLM is instructed: "Write a selection operator with code length $\leq \ell_{\text{target}}$." This prompt-guided strategy encourages the model to produce more concise programs.

Multi-Objective Survival Selection: Beyond prompt-based constraints, we employ a multi-objective survival selection strategy based on both operator fitness and code length to further control bloat, since LLMs do not always follow instructions to generate code within a specified length. Each operator O_i is represented as a tuple $(f(O_i), \ell(O_i))$, where $f(O_i)$ denotes its average task performance across T tasks, and $\ell(O_i)$ denotes its code length. The parents for generating the next generation $\mathcal{P}^{(t+1)}$ are selected from the combined set $\mathcal{P}^{(t)} \cup \mathcal{P}^{\text{offspring}}$ using a dominance-dissimilarity selection mechanism (Yao et al. 2025).

The key idea is to compute a dominance score for each operator based on weak Pareto dominance (Lin et al. 2022) and code similarity. An operator O_i is said to weakly Pareto dominate another operator O_j, denoted $O_i \succeq O_j$, if and only if $f(O_i) \geq f(O_j)$ and $\ell(O_i) \leq \ell(O_j)$. For each such pair (O_i, O_j), the score of O_j is penalized by the code similarity $\text{sim}(O_i, O_j)$, computed using the CodeBLEU metric (Ren et al. 2020). The total dominance score of O_j is then defined as $s(O_j) = \sum_{O_i \succeq O_j} -\text{sim}(O_i, O_j)$. Operators are then ranked, and the top-N operators with higher scores are selected, as they are less frequently dominated and exhibit greater dissimilarity from dominating counterparts. This selection strategy encourages a trade-off between maximizing task performance and minimizing code complexity, effectively reducing operator bloat while maintaining diversity and effectiveness.

3.5 Incorporating Domain Knowledge into Prompts

Algorithm evolution requires deep domain expertise, which general-purpose LLMs often lack. To compensate, domain knowledge is commonly incorporated into prompts to enhance LLM-based algorithm evolution (Romera-Paredes et al. 2024; Liu et al. 2024a). For the design of selection operators, we embed the following principles into the prompt to guide the LLM toward generating more effective solutions:

Diversity-aware: Purely objective-driven selection can cause premature convergence. To counter this, it is desirable to select models that perform well on different training instances to maintain population diversity.

Interpretability-aware: Interpretability, often measured by the number of nodes in a symbolic expression, is a critical criterion. Let n_i denote the number of nodes in solution p_i. The selection operator should favor solutions with smaller n_i

to promote simpler, more interpretable models.

Stage-aware: Selection pressure should adapt over time. In early generations ($t \ll T_{\max}$), it should favor exploration. In later stages ($t \approx T_{\max}$), it should prioritize exploitation of high-fitness solutions to encourage convergence.

Complementarity-aware: To effectively recombine useful traits, crossover should favor solutions with complementary strengths. Given performance vectors $\mathbf{s}_i$ and $\mathbf{s}_j$ over d instances, complementarity means selecting pairs with low correlation. Such combinations integrate distinct capabilities and can yield offspring that perform well across a broader range of instances.

Vectorization-aware: Vectorized operations are preferred, as they can be efficiently accelerated using NumPy or GPU computation. This improves the runtime efficiency of the evolved operator and reduces evaluation overhead during meta-evolution.

The prompt incorporating these domain knowledge principles is provided in Figure 34 of the supplementary material.

4 Experimental Settings

Meta-Evolution: The experiments are conducted using GPT-4.1 Mini. For the outer loop, the population size N is set to 20, the number of solutions generated by mutation M is set to 1, and the number of solutions generated by crossover is 19. The total number of generations is set to 20. A length limit of 30 lines is imposed to control operator complexity. For the inner loop, four datasets are selected to evaluate the quality of the generated selection operators. These datasets have OpenML IDs 505, 4544, 588, and 650, corresponding to the four highest-dimensional datasets in the contemporary symbolic regression benchmark (SRBench) (La Cava et al. 2021). High-dimensional datasets pose greater challenges for symbolic regression algorithms, and optimizing on these is likely to generalize well to both easy and hard problems. The number of generations in the inner loop is set to 30, and the population size is 100. More detailed experimental settings for the inner symbolic regression loop are provided in Appendix A of the supplementary material. All experiments are repeated three times (Yao et al. 2025) with random seeds set to 0, 1, and 2, to report the median and confidence intervals.

Symbolic Regression: For symbolic regression, experiments are conducted on 116 out of the 120 datasets in SRBench, excluding the 4 datasets used during the meta-evolution phase to prevent potential data leakage. Each algorithm is evaluated using 10 independent runs per dataset to ensure statistically robust comparisons. The symbolic regression algorithm used is standard genetic programming with linear scaling, which is a widely adopted framework in symbolic regression research (Virgolin et al. 2021; Kronberger et al. 2022). The parameter settings are the same as those used in the inner symbolic regression loop, except that the number of generations is increased to 100.

5 Experimental Results

5.1 Meta-Evolution Results

In this section, we evaluate our meta-evolution framework (LLM-Meta-SR) against several ablated variants, including: without semantic evolution (W/O Semantics), without semantic evolution and bloat control (W/O SE+BC), without domain knowledge (W/O Knowledge), without domain knowledge and semantic evolution (W/O DK+SE), and without domain knowledge, semantic evolution, and bloat control (W/O DK+SE+BC). Since domain knowledge is not always available in practice, we include ablation studies under the setting without domain knowledge to evaluate how the proposed strategy performs in entirely new domains. In the W/O Semantics setting, random selection with identical objective prevention from ReEvo (Ye et al. 2024) is used as the baseline. In the W/O SE+BC setting, survival selection is replaced with direct population replacement and elitism, following the strategy used in ReEvo (Ye et al. 2024). In the W/O Knowledge setting, semantic evolution and bloat control are retained, but the domain knowledge prompt is removed.

Objective Score: The objective score is measured by the average test R^2 score of each selection operator across the four training datasets. For the W/O DK+SE+BC setting, only one run completed within 24 hours and is thus reported. The results in Figure 4, along with the numerical values presented in Table 1, show that the absence of domain knowledge causes the largest performance drop, highlighting the importance of domain knowledge introduced in Section 3.5 in guiding algorithm evolution. This is consistent with findings from FunSearch (Romera-Paredes et al. 2024) and EoH (Liu et al. 2024a) that LLMs often require external knowledge for guidance.

However, domain knowledge alone is insufficient to discover high-performing operators. As shown in Figure 4, semantic evolution plays a critical role in achieving higher objective scores, both with and without domain knowledge. To investigate the role of semantics, we visualize the semantic distribution of solutions over the first five generations in Figure 6. Each point represents a solution, colored by its average score across the four tasks. Positions are computed via t-SNE based on the four-dimensional score vector. The figure reveals that the number of solutions achieving top-3 performance on at least one dataset consistently exceeds three, indicating that top individuals by average score do not dominate all tasks. Additionally, the final subplot in Figure 6 shows that some solutions, marked with stars, achieve top-3 performance on at least one dataset despite having modest average scores. This illustrates that relying solely on average performance may overlook individuals with strong task-specific capabilities. To further understand how semantic-aware selection works, a real example of parent crossover is provided in Appendix D of the supplementary material, which further confirms the importance of semantic-aware selection for generating better offspring. These findings highlight the importance of semantics-based algorithm evolution in capturing diverse and task-specific behaviors that aggregate metrics may obscure.

Code Length: Figure 5 shows the evolution of code length for the best-performing solutions. Without bloat control, code length grows excessively throughout the optimization process when domain knowledge is absent. Incorporating domain knowledge helps shape the basic structure and mitigates excessive growth, but the code length still remains substantial.

Table 1: Median historical best scores and the corresponding code lengths for each algorithm over three runs.

	LLM-Meta-SR	W/O Semantics	W/O SE+BC	W/O Knowledge	W/O DK+SE	W/O DK+SE+BC
Score	0.86	0.84	0.84	0.79	0.75	0.75
Lines of Code	48	43	80	55	41	222

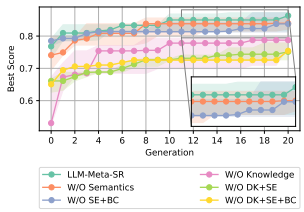

Figure 4: Validation R^2 of the best solution across generations for different LLM-driven search strategies.

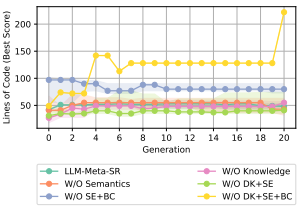

Figure 5: Code length of the best solution across generations for different LLM-driven search strategies.

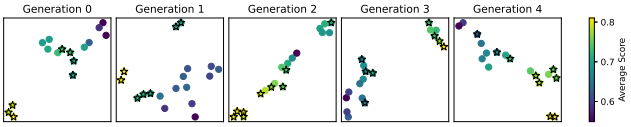

Figure 6: t-SNE visualization of evolved operator semantics. The shape of each point indicates whether it achieved top-3 performance on any dataset: stars denote top-3 performance on at least one dataset, while circles indicate otherwise.

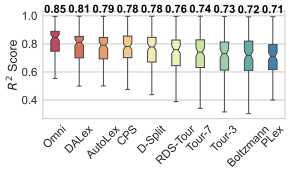

Figure 7: Test R^2 scores of different selection operators on symbolic regression benchmarks. Median values are annotated.

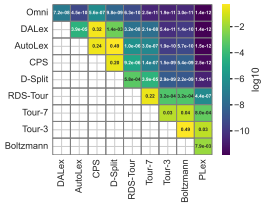

Figure 8: Pairwise statistical comparison of selection operators using the Wilcoxon signed-rank test with Benjamini–Hochberg correction.

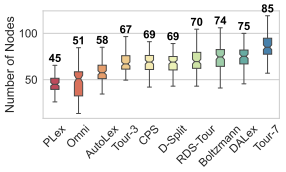

Figure 9: Tree sizes of selection operators on symbolic regression benchmarks.

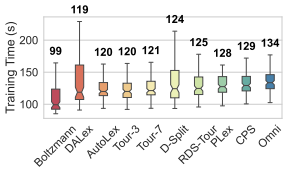

Figure 10: Training times of selection operators on symbolic regression benchmarks.

In contrast, when bloat control is applied, it directly curbs the complexity bias of LLMs, resulting in smaller generated programs that stabilize at a length of around 50, while simultaneously achieving faster improvements in objective scores.

5.2 Discovered Operators

In this section, we present experiments on datasets from contemporary symbolic regression benchmarks to evaluate the performance of the discovered operators. Since the LLM-designed operator incorporates several desirable characteristics of selection operators—including diversity-awareness and stage-awareness—it is referred to as Omni selection (Omni). The code for the Omni selection operator is provided in Appendix C of the supplementary material. We compare the performance of the LLM-generated selection operator against several expert-designed baselines, including automatic ϵ-lexicase selection (AutoLex) (La Cava et al. 2019), probabilistic lexicase selection (PLex) (Ding, Pantridge, and Spector 2023), lexicase-like selection via diverse aggregation (DALex) (Ni, Ding, and Spector 2024), ϵ-lexicase selection with dynamic split (D-Split) (Imai Aldeia, De França, and La Cava 2024), random sampling with tournament selection (RDS-Tour) (Geiger, Sobania, and Rothlauf 2025), complementary phenotype selection (CPS) (Dolin, Arenas, and Merelo 2002), tournament selection with sizes 3 and 7 (Banzhaf et al. 1998), and Boltzmann selection with temperature scheduling (Boltzmann) (Shojaee et al. 2025). All of these operators were manually designed by domain experts. This section only presents results for a single LLM-discovered operator. Additional results for other LLM-discovered operators are presented in Appendix E of the supplementary material.

Test Accuracy: The results in Figure 7 show that the LLM-generated selection operator outperforms the expert-designed baselines in terms of R^2 scores and achieves the best overall performance. A Wilcoxon signed-rank test with Benjamini–Hochberg correction is presented in Figure 8. These results demonstrate that the evolved Omni selection operator performs significantly better than existing expert-designed operators, many of which cannot be statistically distinguished from one another. This confirms that LLMs can effectively discover selection operators that surpass those created by domain experts.

Tree Size: The distribution of model sizes is presented in Figure 9. The results show that the evolved operator produces smaller models compared to top-performing selection operators like AutoLex and CPS. This is primarily because the evolved operator incorporates model size into the selection process, biasing the search toward regions where symbolic expressions are more compact. As shown in Figure 9 and Fig-

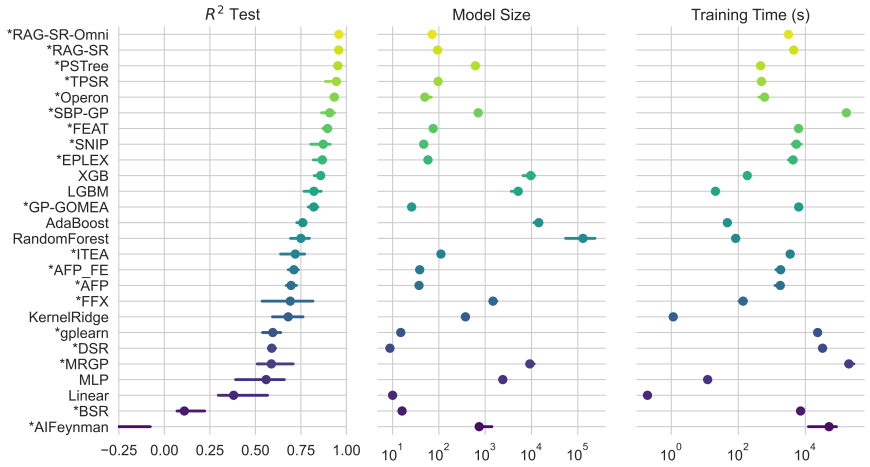

Figure 11: Median R^2 scores, model sizes, and training time of 26 algorithms on the symbolic regression benchmark.

ure 7, model size and performance are not always in conflict. By carefully designing the selection operator, it is possible to evolve symbolic models that achieve both high accuracy and small model size, indicating high interpretability.

Training Time: As shown in Figure 10, the proposed Omni selection operator is efficient, even though it integrates multiple criteria into the selection process, which makes it slightly more time-consuming than some other operators in terms of overall symbolic regression time. However, it is important to consider not only the training phase but also the cost of evaluating and deploying candidate models. Since Omni selection tends to favor smaller models with fewer nodes, it reduces computational cost during evaluation and deployment, especially on resource-constrained devices. Thus, despite its multi-faceted design leading to a slight increase in selection time, Omni remains an efficient and practical selection operator.

5.3 Analysis on State-of-the-Art Symbolic Regression Algorithms

Experimental Settings In this section, we apply the evolved operator to a state-of-the-art Transformer-assisted symbolic regression algorithm, namely retrieval-augmentation-generation-based symbolic regression (RAG-SR) (Zhang et al. 2025), to evaluate the effectiveness of the proposed operator in the context of modern symbolic regression. The only modification is the replacement of the automatic epsilon lexicase selection (La Cava et al. 2019) with the Omni selection operator described in Code 3 in the supplementary material. All other components remain unchanged. The resulting algorithm is referred to as RAG-SR-Omni. For the experimental datasets, following the setup in Section 4, we use 116 out of 120 regression problems, excluding the four used during meta-evolution.

Experimental Results The results are presented in Figure 11. The figure show that RAG-SR-Omni outperforms RAG-SR in terms of median R^2 scores, model sizes, and training time, achieving the best performance among 26 symbolic regression and machine learning algorithms. These results indicate that the LLM-evolved selection operator can be seamlessly integrated into state-of-the-art symbolic regression methods to further enhance their performance.

Recall that RAG-SR uses automatic-lexicase selection, which only considers diversity-awareness. These results suggest that even for a state-of-the-art symbolic regression algorithm, it is still desirable to consider multiple aspects, such as diversity, stage-awareness, complementarity, and interpretability, during selection to achieve stronger performance. Nonetheless, manually designing such an operator is challenging, which highlights the value of leveraging LLMs in this context.

6 Conclusions and Limitations

In this paper, we propose an LLM-driven framework for automatically designing selection operators in symbolic regression, and design several strategies to enhance its effectiveness. First, we highlight the challenges of limited semantic awareness and bloat in LLM-driven evolution. To enhance semantic awareness, we propose a semantic-based selection operator and a semantic feedback mechanism. To mitigate bloat, we introduce a prompt-based length control and a multi-objective survival selection strategy. Additionally, we define a set of design principles derived from domain knowledge to guide LLM for operator generation. Ablation studies confirm that addressing these three factors significantly enhances the effectiveness of the evolved algorithms. Next, we evaluate an LLM-generated selection operator against expert-designed baselines across a wide range of symbolic regression datasets, and also apply the evolved operator to state-of-the-art symbolic regression algorithms. The results show that the evolved operator outperforms expert-designed selection operators and can boost the performance of SR algorithms, achieving the best performance among 26 SR and ML algorithms, demonstrating that LLMs can discover selection operators that outperform those crafted by domain experts. For future work, it is worth exploring the automatic design of crossover and mutation operators in symbolic regression to further reduce the effort required for developing novel SR algorithms.

References

Allen, D. M. 1974. The relationship between variable selection and data agumentation and a method for prediction. *Technometrics*, 16(1): 125–127.

Banzhaf, W.; Nordin, P.; Keller, R. E.; and Francone, F. D. 1998. *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications*. Morgan Kaufmann Publishers Inc.

Biggio, L.; Bendinelli, T.; Neitz, A.; Lucchi, A.; and Parascandolo, G. 2021. Neural symbolic regression that scales. In *International Conference on Machine Learning*, 936–945. Pmlr.

Boldi, R.; Briesch, M.; Sobania, D.; Lalejini, A.; Helmuth, T.; Rothlauf, F.; Ofria, C.; and Spector, L. 2024. Informed Down-Sampled Lexicase Selection: Identifying productive training cases for efficient problem solving. *Evolutionary computation*, 32(4): 307–337.

Cava, W. L.; Singh, T. R.; Taggart, J.; Suri, S.; and Moore, J. 2019. Learning concise representations for regression by evolving networks of trees. In *International Conference on Learning Representations*.

Chan, J. S.; Chowdhury, N.; Jaffe, O.; Aung, J.; Sherburn, D.; Mays, E.; Starace, G.; Liu, K.; Maksin, L.; Patwardhan, T.; et al. 2025. MLE-bench: Evaluating Machine Learning Agents on Machine Learning Engineering. In *ICLR*. OpenReview.net.

Chen, A.; Dohan, D.; and So, D. 2023. Evoprompting: Language models for code-level neural architecture search. *Advances in Neural Information Processing Systems*, 36: 7787–7817.

Chen, J.; Ma, Z.; Guo, H.; Ma, Y.; Zhang, J.; and Gong, Y.-J. 2024a. SYMBOL: Generating Flexible Black-Box Optimizers through Symbolic Equation Learning. In *The Twelfth International Conference on Learning Representations*.

Chen, J.; Tian, J.; Wu, L.; ChenXinWei; Yang, X.; Jin, Y.; and Xu, Y. 2025. KinFormer: Generalizable Dynamical Symbolic Regression for Catalytic Organic Reaction Kinetics. In *The Thirteenth International Conference on Learning Representations*.

Chen, X.; Liang, C.; Huang, D.; Real, E.; Wang, K.; Pham, H.; Dong, X.; Luong, T.; Hsieh, C.-J.; Lu, Y.; et al. 2023. Symbolic discovery of optimization algorithms. *Advances in Neural Information Processing Systems*, 36: 49205–49233.

Chen, Z.; Zhou, Z.; Lu, Y.; Xu, R.; Pan, L.; and Lan, Z. 2024b. QUBE: Enhancing Automatic Heuristic Design via Quality-Uncertainty Balanced Evolution. *arXiv preprint arXiv:2412.20694*.

Dat, P. V. T.; Doan, L.; and Binh, H. T. T. 2025. Hsevo: Elevating automatic heuristic design with diversity-driven harmony search and genetic algorithm using llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 26931–26938.

Ding, L.; Pantridge, E.; and Spector, L. 2023. Probabilistic lexicase selection. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 1073–1081.

Dolin, B.; Arenas, M. G.; and Merelo, J. J. 2002. Opposites attract: Complementary phenotype selection for crossover in genetic programming. In *Parallel Problem Solving from Nature—PPSN VII: 7th International Conference Granada, Spain, September 7–11, 2002 Proceedings 7*, 142–152. Springer.

Feng, M.; Huang, Y.; Liu, Y.; Jiang, B.; and Yan, J. 2025. PhysPDE: Rethinking PDE Discovery and a Physical HYpothesis Selection Benchmark. In *The Thirteenth International Conference on Learning Representations*.

Fong, K. S.; and Motani, M. 2024a. MetaSR: A Meta-Learning Approach to Fitness Formulation for Frequency-Aware Symbolic Regression. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 878–886.

Fong, K. S.; and Motani, M. 2024b. Symbolic regression enhanced decision trees for classification tasks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 12033–12042.

Fortin, F.-A.; De Rainville, F.-M.; Gardner, M.-A. G.; Parizeau, M.; and Gagné, C. 2012. DEAP: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, 13(1): 2171–2175.

Gao, X.; and Das, K. 2024. Customizing language model responses with contrastive in-context learning. In *Proceedings of the aaai conference on artificial intelligence*, volume 38, 18039–18046.

Geiger, A.; Briesch, M.; Sobania, D.; and Rothlauf, F. 2025. Was Tournament Selection All We Ever Needed? A Critical Reflection on Lexicase Selection. In *European Conference on Genetic Programming (Part of EvoStar)*, 207–223. Springer.

Geiger, A.; Sobania, D.; and Rothlauf, F. 2025. A Performance Analysis of Lexicase-Based and Traditional Selection Methods in GP for Symbolic Regression. *ACM Trans. Evol. Learn. Optim.*

Gong, N.; Reddy, C. K.; Ying, W.; Chen, H.; and Fu, Y. 2025. Evolutionary large language model for automated feature transformation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 16844–16852.

Grayeli, A.; Sehgal, A.; Costilla-Reyes, O.; Cranmer, M.; and Chaudhuri, S. 2024. Symbolic regression with a learned concept library. *arXiv preprint arXiv:2409.09359*.

Huang, Y.; Wu, S.; Zhang, W.; Wu, J.; Feng, L.; and Tan, K. C. 2025a. Autonomous multi-objective optimization using large language model. *IEEE Transactions on Evolutionary Computation*.

Huang, Z.; Mei, Y.; Zhang, F.; and Zhang, M. 2024. Toward evolving dispatching rules with flow control operations by grammar-guided linear genetic programming. *IEEE Transactions on Evolutionary Computation*.

Huang, Z.; Wu, W.; Wu, K.; Wang, J.; and Lee, W.-B. 2025b. CALM: Co-evolution of Algorithms and Language Model for Automatic Heuristic Design. *arXiv preprint arXiv:2505.12285*.

Imai Aldeia, G. S.; De França, F. O.; and La Cava, W. G. 2024. Minimum variance threshold for epsilon-lexicase selection. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 905–913.

Jiang, C.; Shu, X.; Qian, H.; Lu, X.; Zhou, J.; Zhou, A.; and Yu, Y. 2025. LLMOPT: Learning to Define and Solve General Optimization Problems from Scratch. In *Proceedings of the Thirteenth International Conference on Learning Representations (ICLR)*. Singapore, Singapore.

Jiang, N.; and Xue, Y. 2024. Racing Control Variable Genetic Programming for Symbolic Regression. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 12901–12909.

Kahlmeyer, P.; Fischer, M.; and Giesen, J. 2025. Dimension Reduction for Symbolic Regression. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 17707–17714.

Kamienny, P.-A.; d'Ascoli, S.; Lample, G.; and Charton, F. 2022. End-to-end symbolic regression with transformers. *Advances in Neural Information Processing Systems*, 35: 10269–10281.

Kamienny, P.-A.; Lample, G.; Lamprier, S.; and Virgolin, M. 2023. Deep generative symbolic regression with Monte-Carlo-tree-search. In *International Conference on Machine Learning*, 15655–15668. PMLR.

Kronberger, G.; de França, F. O.; Burlacu, B.; Haider, C.; and Kommenda, M. 2022. Shape-constrained symbolic regression—improving extrapolation with prior knowledge. *Evolutionary Computation*, 30(1): 75–98.

La Cava, W.; Burlacu, B.; Virgolin, M.; Kommenda, M.; Orzechowski, P.; de França, F. O.; Jin, Y.; and Moore, J. H. 2021. Contemporary symbolic regression methods and their relative performance. *Advances in neural information processing systems*, 2021(DB1): 1.

La Cava, W.; Helmuth, T.; Spector, L.; and Moore, J. H. 2019. A probabilistic and multi-objective analysis of lexicase selection and ε-lexicase selection. *Evolutionary Computation*, 27(3): 377–402.

Landajuela, M.; Lee, C. S.; Yang, J.; Glatt, R.; Santiago, C. P.; Aravena, I.; Mundhenk, T.; Mulcahy, G.; and Petersen, B. K. 2022. A unified framework for deep symbolic regression. *Advances in Neural Information Processing Systems*, 35: 33985–33998.

Lehman, J.; and Stanley, K. O. 2010. Efficiently evolving programs through the search for novelty. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, 837–844.

Lin, X.; Yang, Z.; Zhang, X.; and Zhang, Q. 2022. Pareto set learning for expensive multi-objective optimization. *Advances in Neural Information Processing Systems*, 35: 19231–19247.

Liu, F.; Xialiang, T.; Yuan, M.; Lin, X.; Luo, F.; Wang, Z.; Lu, Z.; and Zhang, Q. 2024a. Evolution of Heuristics: Towards Efficient Automatic Algorithm Design Using Large Language Model. In *International Conference on Machine Learning*, 32201–32223. PMLR.

Liu, S.; Chen, C.; Qu, X.; Tang, K.; and Ong, Y.-S. 2024b. Large language models as evolutionary optimizers. In *2024 IEEE Congress on Evolutionary Computation (CEC)*, 1–8. IEEE.

Liu, T.; Huynh, N.; and van der Schaar, M. 2025. Decision Tree Induction Through LLMs via Semantically-Aware Evolution. In *The Thirteenth International Conference on Learning Representations*.

Lones, M. A. 2021. Evolving continuous optimisers from scratch. *Genetic Programming and Evolvable Machines*, 22(4): 395–428.

Ma, Y. J.; Liang, W.; Wang, G.; Huang, D.-A.; Bastani, O.; Jayaraman, D.; Zhu, Y.; Fan, L.; and Anandkumar, A. 2024. Eureka: Human-Level Reward Design via Coding Large Language Models. In *The Twelfth International Conference on Learning Representations*.

Martins, J. F. B.; Oliveira, L. O. V.; Miranda, L. F.; Casadei, F.; and Pappa, G. L. 2018. Solving the exponential growth of symbolic regression trees in geometric semantic genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 1151–1158.

Maudet, G.; and Danoy, G. 2025. Search Strategy Generation for Branch and Bound Using Genetic Programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 11299–11308.

Meyerson, E.; Nelson, M. J.; Bradley, H.; Gaier, A.; Moradi, A.; Hoover, A. K.; and Lehman, J. 2024. Language model crossover: Variation through few-shot prompting. *ACM Transactions on Evolutionary Learning*, 4(4): 1–40.

Mo, S.; Wu, K.; Gao, Q.; Teng, X.; and Liu, J. 2025. AutoSGNN: automatic propagation mechanism discovery for spectral graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 19493–19502.

Moraglio, A.; Krawiec, K.; and Johnson, C. G. 2012. Geometric semantic genetic programming. In *Parallel Problem Solving from Nature-PPSN XII: 12th International Conference, Taormina, Italy, September 1-5, 2012, Proceedings, Part I 12*, 21–31. Springer.

Nadizar, G.; Sakallioglu, B.; Garrow, F.; Silva, S.; and Vanneschi, L. 2024. Geometric semantic GP with linear scaling: Darwinian versus Lamarckian evolution. *Genetic Programming and Evolvable Machines*, 25(2): 17.

Ni, A.; Ding, L.; and Spector, L. 2024. Dalex: Lexicase-like selection via diverse aggregation. In *European Conference on Genetic Programming (Part of EvoStar)*, 90–107. Springer.

Ni, J.; Drieberg, R. H.; and Rockett, P. I. 2012. The use of an analytic quotient operator in genetic programming. *IEEE Transactions on Evolutionary Computation*, 17(1): 146–152.

Owen, C. A.; Dick, G.; and Whigham, P. A. 2022. Standardization and data augmentation in genetic programming. *IEEE Transactions on Evolutionary Computation*, 26(6): 1596–1608.

Peabody, C.; and Seitzer, J. 2015. GEF: A self-programming robot using grammatical evolution. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29.

Pluhacek, M.; Kovac, J.; Viktorin, A.; Janku, P.; Kadavy, T.; and Senkerik, R. 2024. Using LLM for automatic evolvement of metaheuristics from swarm algorithm SOMA. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2018–2022.

Real, E.; Liang, C.; So, D.; and Le, Q. 2020. Automl-zero: Evolving machine learning algorithms from scratch. In *International Conference on Machine Learning*, 8007–8019. PMLR.

Ren, S.; Guo, D.; Lu, S.; Zhou, L.; Liu, S.; Tang, D.; Sundaresan, N.; Zhou, M.; Blanco, A.; and Ma, S. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Romera-Paredes, B.; Barekatain, M.; Novikov, A.; Balog, M.; Kumar, M. P.; Dupont, E.; Ruiz, F. J.; Ellenberg, J. S.; Wang, P.; Fawzi, O.; et al. 2024. Mathematical discoveries from program search with large language models. *Nature*, 625(7995): 468–475.

Shen, J.; Qian, H.; Zhang, W.; and Zhou, A. 2024. Symbolic cognitive diagnosis via hybrid optimization for intelligent education systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 14928–14936.

Shi, H.; Song, W.; Zhang, X.; Shi, J.; Luo, C.; Ao, X.; Arian, H.; and Seco, L. A. 2025. AlphaForge: A Framework to Mine and Dynamically Combine Formulaic Alpha Factors. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 12524–12532.

Shojaee, P.; Meidani, K.; Barati Farimani, A.; and Reddy, C. 2024. Transformer-based planning for symbolic regression. *Advances in Neural Information Processing Systems*, 36.

Shojaee, P.; Meidani, K.; Gupta, S.; Farimani, A. B.; and Reddy, C. K. 2025. LLM-SR: Scientific Equation Discovery via Programming with Large Language Models. In *The Thirteenth International Conference on Learning Representations*.

Song, X.; Tian, Y.; Lange, R. T.; Lee, C.; Tang, Y.; and Chen, Y. 2024. Position: leverage foundational models for black-box optimization. In *Proceedings of the 41st International Conference on Machine Learning*, 46168–46180.

Stanovov, V.; Akhmedova, S.; and Semenkin, E. 2022. The automatic design of parameter adaptation techniques for differential evolution with genetic programming. *Knowledge-Based Systems*, 239: 108070.

Stein, N. v.; and Bäck, T. 2024. Llamea: A large language model evolutionary algorithm for automatically generating metaheuristics. *IEEE Transactions on Evolutionary Computation*.

Šurina, A.; Mansouri, A.; Seddas, A.; Viazovska, M.; Abbe, E.; and Gulcehre, C. 2025. Algorithm Discovery With LLMs: Evolutionary Search Meets Reinforcement Learning. In *Scaling Self-Improving Foundation Models without Human Supervision*.

Virgolin, M.; Alderliesten, T.; Witteveen, C.; and Bosman, P. A. 2021. Improving model-based genetic programming for symbolic regression of small expressions. *Evolutionary computation*, 29(2): 211–237.

Wu, X.; Wang, D.; Wu, C.; Wen, L.; Miao, C.; Xiao, Y.; and Zhou, Y. 2025. Efficient Heuristics Generation for Solving Combinatorial Optimization Problems Using Large Language Models. *arXiv preprint arXiv:2505.12627*.

Xie, H.; and Zhang, M. 2012. Parent selection pressure auto-tuning for tournament selection in genetic programming. *IEEE Transactions on Evolutionary Computation*, 17(1): 1–19.

Xu, M.; Mei, Y.; Zhang, F.; and Zhang, M. 2022. Genetic programming with diverse partner selection for dynamic flexible job shop scheduling. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 615–618.

Xue, K.; Xu, J.; Yuan, L.; Li, M.; Qian, C.; Zhang, Z.; and Yu, Y. 2022. Multi-agent dynamic algorithm configuration. *Advances in Neural Information Processing Systems*, 35: 20147–20161.

Yang, C.; Wang, X.; Lu, Y.; Liu, H.; Le, Q. V.; Zhou, D.; and Chen, X. 2024. Large Language Models as Optimizers. In *The Twelfth International Conference on Learning Representations*.

Yao, S.; Liu, F.; Lin, X.; Lu, Z.; Wang, Z.; and Zhang, Q. 2025. Multi-objective evolution of heuristic using large language model. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 27144–27152.

Ye, F.; Doerr, C.; Wang, H.; and Bäck, T. 2022. Automated configuration of genetic algorithms by tuning for anytime performance. *IEEE Transactions on Evolutionary Computation*, 26(6): 1526–1538.

Ye, H.; Wang, J.; Cao, Z.; Berto, F.; Hua, C.; Kim, H.; Park, J.; and Song, G. 2024. ReEvo: Large Language Models as Hyper-Heuristics with Reflective Evolution. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Ye, H.; Xu, H.; Yan, A.; and Cheng, Y. 2025. Large Language Model-driven Large Neighborhood Search for Large-Scale MILP Problems. In *Forty-second International Conference on Machine Learning*.

Yu, Z.; Ding, J.; Li, Y.; and Jin, D. 2025. Symbolic regression via MDLformer-guided search: from minimizing prediction error to minimizing description length. In *The Thirteenth International Conference on Learning Representations*.

Zhang, H.; Chen, Q.; XUE, B.; Banzhaf, W.; and Zhang, M. 2025. RAG-SR: Retrieval-Augmented Generation for Neural Symbolic Regression. In *The Thirteenth International Conference on Learning Representations*.

Zhang, R.; Liu, F.; Lin, X.; Wang, Z.; Lu, Z.; and Zhang, Q. 2024. Understanding the importance of evolutionary search in automated heuristic design with large language models. In *International Conference on Parallel Problem Solving from Nature*, 185–202. Springer.

Zhao, Z.; Wen, H.; Wang, P.; Wei, Y.; Zhang, Z.; Lin, X.; Liu, F.; An, B.; Xiong, H.; Wang, Y.; et al. 2025. From Understanding to Excelling: Template-Free Algorithm Design through Structural-Functional Co-Evolution. *arXiv preprint arXiv:2503.10721*.

A Evolutionary Symbolic Regression

A.1 Solution Representation

The symbolic regression model is represented as a symbolic expression tree with linear scaling (Nadizar et al. 2024). For a given expression Φ, the prediction is computed as $\hat{Y} = \alpha \cdot \Phi(X) + \beta$, where the coefficients $\alpha \in \mathbb{R}$ and $\beta \in \mathbb{R}$ are fitted using ridge regression by minimizing the regularized squared error between $\hat{Y}$ and the targets Y.

A.2 Algorithm Workflow

The symbolic regression algorithm is implemented using the genetic programming framework (Banzhaf et al. 1998). Let $\mathcal{F}^{(t)} = \{\Phi_1, \Phi_2, \ldots, \Phi_\lambda\}$ denote the population of λ symbolic expressions at generation t. The evolutionary process proceeds as follows:

- **Population Initialization:** The initial population $\mathcal{F}^{(0)}$ is generated using the ramped half-and-half method (Banzhaf et al. 1998).
- **Fitness Evaluation:** Each expression $\Phi_i \in \mathcal{F}^{(t)}$ is evaluated on the dataset (X, Y). Let $\mathbf{z}_i = \Phi_i(X) \in \mathbb{R}^n$ denote the vector of symbolic outputs. The linear coefficients α_i and β_i are computed by fitting a ridge regression model of the form $\hat{Y}_i = \alpha_i \cdot \mathbf{z}_i + \beta_i$. The leave-one-out cross-validation (LOOCV) (Allen 1974) squared error is computed efficiently as $\mathcal{E}_i = \sum_{j=1}^{n} \left(\frac{r_{ij}}{1 - H_{ijj}} \right)^2$, where $\mathbf{r}_i = Y - \hat{Y}_i$ is the residual vector, and $H_i = Z_i (Z_i^\top Z_i + \lambda I)^{-1} Z_i^\top$ is the hat matrix for ridge regression with input $Z_i = [\mathbf{z}_i \ \mathbf{1}] \in \mathbb{R}^{n \times 2}$. The full error vector is retained for use in selection.
- **Elitism:** The best-performing expression $\Phi^* = \arg\min_{\Phi_i \in \mathcal{F}^{(t)}} \mathcal{E}_i$ is preserved in an external archive and used for solution selection to maintain historical performance.
- **Solution Selection:** The LLM-evolved selection operator is invoked to select a set of promising parent expressions from the current population and the elite archive.
- **Solution Generation:** A new population $\mathcal{F}^{(t+1)}$ is generated from the selected parents using standard genetic programming operators:
  - **Subtree Crossover:** Given two parent expressions Φ_a and Φ_b, offspring are generated by exchanging randomly selected subtrees.
  - **Subtree Mutation:** A randomly selected subtree of an expression Φ_i is replaced with a newly generated subtree.

  During solution generation, all solutions generated in the evolutionary process are stored in a hash set. If an offspring is identical to any historical solution, it is discarded and a new solution is generated. The hash set is used because it maintains $O(1)$ query complexity. This mechanism prevents symbolic regression from wasting resources by evaluating the same solution multiple times.

A.3 Parameter Settings

The parameters for symbolic regression follow conventional settings. The population size and number of generations are set to match those used in D-Split (Imai Aldeia, De França, and La Cava 2024). To prevent division-by-zero errors, we use the analytical quotient ($\mathrm{AQ}(x, y) = \frac{x}{\sqrt{1+y^2}}$) (Ni, Drieberg, and Rockett 2012) in place of the standard division operator.

Table 2: Parameter settings

Parameter	Value
Population size	100
Maximum generations	100
Maximum tree depth	10
Initialization method	Ramped half-and-half (depth 0-6)
Function set	$+$, $-$, $\times$, AQ, $\sqrt{\lvert\cdot\rvert}$, $\log(1+\lvert\cdot\rvert)$, $\lvert\cdot\rvert$, $(\cdot)^2$, $\sin_\pi(\cdot)$, $\cos_\pi(\cdot)$, Max, Min, Neg
Crossover rate	0.9
Mutation rate	0.1

A.4 Evaluation Protocol in Meta-Evolution

In the evaluation of selection operators generated by the LLM, all other algorithmic components—including solution initialization, evaluation, generation, and elitism—follow standard practices commonly used in evolutionary symbolic regression. For each dataset $\mathcal{D}_j$, the data is split into training and validation subsets with an 80:20 ratio (La Cava et al. 2021). The regression model is optimized to maximize the training R^2 score, while the validation R^2 score is used as the final score of the selection operator. This encourages the selection of operators with implicit regularization capabilities.

A.5 Evaluation Protocol for Discovered Operators

For the experiment evaluating the evolved selection operator on SRBench, the dataset is randomly split into a training set and a test set with an 80:20 ratio (La Cava et al. 2021). Subsampling is applied to limit the maximum number of training instances to 10,000, consistent with the SRBench protocol (La Cava et al. 2021). Symbolic regression is run on the training set and evaluated on the test set. All features are standardized, as this has been shown to be beneficial for symbolic regression (Owen, Dick, and Whigham 2022). Standardization parameters are learned from the training data and applied to both the training and test sets. All symbolic regression experiments are conducted on an AMD Milan 7713 CPU.

B Analysis of Token Count of Generated Code

The token count directly impacts the cost of algorithm evolution, as language service providers typically charge based on the number of tokens. In this section, we analyze how bloat influences the token count of generated code. Token

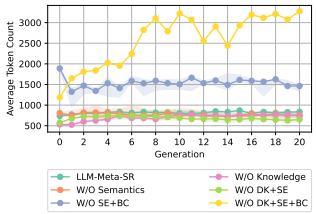

Figure 12: Average token count of generated code across generations for different LLM-driven search strategies.

counting is performed using the `cl100k_base` tokenizer, which is employed by OpenAI models. The experimental results are shown in Figure 12. It is evident that when no bloat control strategy is applied, the token count of the generated code is significantly higher. In contrast, the application of a bloat control strategy substantially reduces the token count in both the LLM-Meta-SR and LLM-Meta-SR without domain knowledge settings. This indicates that the bloat control strategy is not only helpful for improving the interpretability of generated code, but also beneficial for reducing the cost of algorithm evolution.

C Analysis of an Evolved Selection Operator

C.1 Code Evolved by LLM-Meta-SR

Static Analysis Code 1 shows the best evolved selection operator among all runs. The code has been simplified by removing logically redundant parts. In selecting the first parent (parent_a), the operator evaluates two key criteria: specificity—quantified as the lowest error achieved on any subset of the dataset, with lower values indicating superior performance in specialized regions—and a complexity measure that jointly considers the number of nodes in the symbolic expression tree and the height of the tree, thereby reflecting the interpretability of the solution. Candidate solutions are ranked such that subset error is prioritized, with model complexity serving as a secondary criterion when the subset error alone does not sufficiently differentiate candidates. This approach encourages the identification of solutions that balance high accuracy with interpretability.

For the selection of the second parent (parent_b), the operator computes the absolute cosine similarity between the residuals of each candidate in the population and the residual of the previously selected first parent. The second parent is then chosen based on a combination of semantic complementarity, as measured by cosine similarity, and model complexity. The trade-off between complementarity and complexity is controlled by a linear function related to the evolutionary stage. In the early stages, the pressure toward parsimony is intentionally relaxed to promote broader exploration of the search space. In contrast, in later stages, the pressure increases to favor models that are both accurate and concise, thus facilitating interpretability.

Overall, the selection operator satisfies all the desired properties, including interpretability, diversity, complementarity, stage-awareness, and vectorization, as defined in Section 3.5.

```python
def omni_selection(population, k=100, status={}):
    stage = np.clip(status.get("evolutionary_stage", 0), 0, 1)
    n = len(population)
    half_k = k // 2
    y = population[0].y
    n_cases = y.size

    ssize = max(7, n_cases // max(1, 2 * half_k))
    half_struct = half_k // 2
    structured = [
        np.arange(i * ssize, min((i + 1) * ssize, n_cases)) for i in range(half_struct)
    ]
    random_ = [
        np.random.choice(n_cases, ssize, replace=False)
        for _ in range(half_k - half_struct)
    ]
    subsets = structured + random_

    residuals = np.vstack([ind.y - ind.predicted_values for ind in population])
    complexity = np.array([len(ind) + ind.height for ind in population], float)
    complexity /= max(1, complexity.max())
    subset_mse = np.array(
        [
            [((residuals[i, s]) ** 2).mean() if s.size else np.inf for s in subsets]
            for i in range(n)
        ]
    )
    comp_factor = 0.25 + 0.25 * stage

    parent_a = [
        population[np.lexsort((complexity, subset_mse[:, i]))[0]]
        for i in range(len(subsets))
    ]

    idx_map = {ind: i for i, ind in enumerate(population)}
    norms = np.linalg.norm(residuals, axis=1) + 1e-12

    parent_b = []
    for a in parent_a:
        i_a = idx_map[a]
        res_a = residuals[i_a]
        cors = (residuals @ res_a) / (norms * norms[i_a])
        cors[i_a] = 1
        comp_score = np.abs(cors) + comp_factor * complexity
        b_idx = np.argmin(comp_score)
```

```
        parent_b.append(population[b_idx])

    return [ind for pair in zip(parent_a, parent_b) for ind in pair][:k]

```

Code 1: Omni Selection

Benchmark Analysis To further analyze the behavior of the Omni selection operator, we plot the test R^2 scores on five representative datasets in Figure 13. The diversity of the population is shown in Figure 14, and the tree size is shown in Figure 15.

Test R^2 Scores: The test R^2 scores demonstrate that the Omni selection operator achieves an advantage at the beginning of the search, indicating that it is an effective selection operator for anytime performance (Ye et al. 2022).

Population Diversity: To understand why the Omni selection operator achieves this performance, we plot the population diversity trajectory in Figure 14. Diversity is measured by the cosine distance between individuals, which is preferred over Euclidean distance because the latter can be trivially large if some solutions have very large errors. The results show that population diversity is well maintained across generations and is better than the baseline. In other words, the Omni selection operator maintains a more diverse population, with individuals making errors on different subsets of instances. This helps explain the superior performance of the Omni selection operator compared to the baseline.

Tree Size: Finally, we plot the tree size trajectory in Figure 15. The results show that Omni selection not only achieves better performance but also suffers less from rapid growth in tree size compared to other selection operators throughout the entire evolution process. This suggests that the Omni selection operator explores regions of the search space containing parsimonious solutions more exhaustively than other operators, which is an ideal behavior for finding effective and parsimonious solutions in symbolic regression.

C.2 Code Evolved by LLM-Meta-SR without Domain Knowledge

In this section, we analyze the code evolved by LLM-Meta-SR without domain knowledge to understand how an LLM can generate algorithms based solely on its internal knowledge. The evolved code is shown in Code 2. We refer to the resulting algorithm as Omni-Zero, as it evolves the model from scratch, similar to AutoML-Zero (Real et al. 2020).

Static Analysis Based on the code in Code 2, the LLM can evolve selection operators with desirable properties even without domain-specific guidance. These properties include diversity-awareness, stage-awareness, and interpretability-awareness. For diversity-awareness, the selection operator uses cosine similarity to measure pairwise distances between individuals. Formally, it defines the novelty score as:

$$\text{nov}_i = \frac{1 - \frac{1}{n}\sum_{k=1}^{n}\left\langle \frac{\mathbf{x}_i - \bar{x}_i}{\|\mathbf{x}_i - \bar{x}_i\|_2 + \epsilon}, \frac{\mathbf{x}_k - \bar{x}_k}{\|\mathbf{x}_k - \bar{x}_k\|_2 + \epsilon} \right\rangle}{\max_j \left[1 - \frac{1}{n}\sum_{k=1}^{n}\left\langle \frac{\mathbf{x}_j - \bar{x}_j}{\|\mathbf{x}_j - \bar{x}_j\|_2 + \epsilon}, \frac{\mathbf{x}_k - \bar{x}_k}{\|\mathbf{x}_k - \bar{x}_k\|_2 + \epsilon} \right\rangle\right]} \tag{3}$$

This cosine-distance-based novelty score is better suited for symbolic regression than Euclidean-distance-based novelty scores, as it is more robust to outliers and cannot be easily manipulated by increasing the distance with some outlier, as discussed in Appendix C.1.

For stage-awareness, the selection operator applies a non-linear weighting function to control the trade-off between error and novelty. The function is defined as $0.35 + 0.3\left(1 - \frac{t}{T_{\max}}\right)^{0.8}$, where t is the current generation and $T_{\max}$ is the maximum number of generations, as described in Code 2. This function gradually decreases over time, promoting exploration in early generations and exploitation in later ones. This dynamic trade-off satisfies the stage-awareness criterion defined in Section 3.5.

For interpretability-awareness, the selection operator uses both the number of nodes and the height of the tree to measure solution complexity. A linear weighting function is used to balance interpretability and accuracy during selection.

A limitation of this operator is that it does not consider diversity from the perspective of specificity, the selection of each individual is largely dominated by the mean squared error of each individual, nor does it account for the complementarity between two parents.

```
def omni_selection(population, k=100, status={}):
    import numpy as np
    evo_stage = status.get("evolutionary_stage", 0.)
    n = len(population)
    if n == 0:
        return []

    # Extract metrics
    errs = np.array([np.mean(ind.case_values) for ind in population], dtype=float)
    sizes = np.array([len(ind) for ind in population], dtype=float)
    heights = np.array([ind.height for ind in population], dtype=float)
    residuals = np.array([ind.y - ind.predicted_values for ind in population])
    preds = np.array([ind.predicted_values for ind in population])

    def safe_norm(arr):
        m = max(arr.max(), 1e-10)
        return arr / m

    norm_size, norm_height = safe_norm(sizes), safe_norm(heights)
    res_vars = np.var(residuals, axis=1)
    res_norms = np.linalg.norm(residuals, axis=1)
    norm_var, norm_resnorm = safe_norm(res_vars), safe_norm(res_norms)
    norm_err = safe_norm(errs)

```

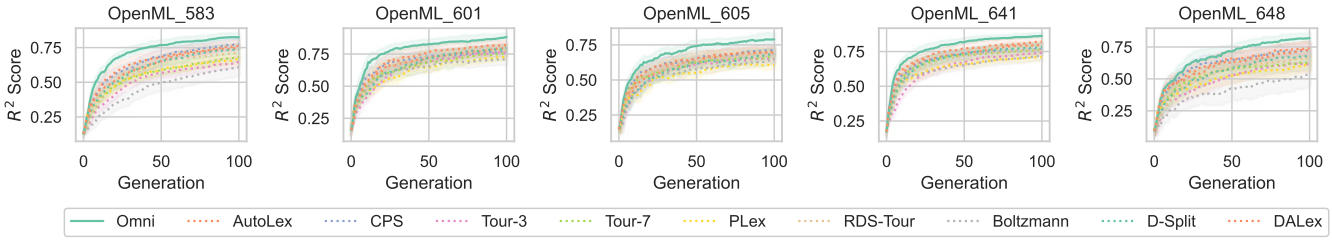

Figure 13: Test R^2 scores across generations.

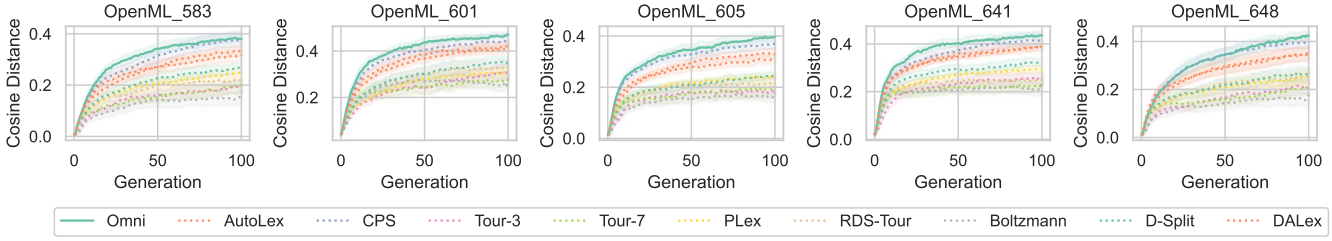

Figure 14: Population cosine semantic diversity across generations.

```
    # Complexity metric combines more residual stats while balancing structural features
    complexity = (norm_size + norm_height + norm_var + norm_resnorm) / 4
    alpha, beta = 1 - evo_stage, evo_stage
    base_score = beta * (1 - norm_err) + alpha * (1 - complexity)
    base_score -= base_score.min()
    if base_score.sum() == 0:
        base_score[:] = 1
    base_probs = base_score / base_score.sum()

    # Novelty score based on residuals and predicted values diversity
    def novelty_score(mat):
        centered = mat - mat.mean(axis=1, keepdims=True)
        norms = np.linalg.norm(centered, axis=1, keepdims=True) + 1e-10
        normed = centered / norms
        sim = normed @ normed.T
        nov = 1 - sim.mean(axis=1)
        max_n = nov.max()
        return nov / (max_n if max_n > 0 else 1)

    novelty_res = novelty_score(residuals)
    novelty_pred = novelty_score(preds)
    novelty = 0.5 * (novelty_res + novelty_pred)

    # Dynamic novelty weighting: stronger early novelty, moderate late novelty
    novelty_weight = 0.35 + 0.3 * (1 - evo_stage)**0.8

    mixed_probs = (1 - novelty_weight) * base_probs + novelty_weight * novelty
    mixed_probs -= mixed_probs.min()
    if mixed_probs.sum() == 0:
        mixed_probs[:] = 1
    mixed_probs /= mixed_probs.sum()

    # Adaptive tournament size encourages pressure on error, preserves diversity
    tour_size = min(3 + int(evo_stage * 2), n)
    selected = []
    while len(selected) < k:
        chosen = np.random.choice(n, size=tour_size, replace=False, p=mixed_probs)
        # Tournament winner: lowest error considering base scores and novelty tie-break
        chosen_errs = errs[chosen]
        chosen_scores = base_probs[chosen]
        min_err_idx = chosen_errs.argmin()
        # Tie-break with highest combined score
        best_candidates = np.flatnonzero(chosen_errs == chosen_errs[min_err_idx])
        if len(best_candidates) > 1:
            best_scores = chosen_scores[best_candidates] + novelty[chosen][
```

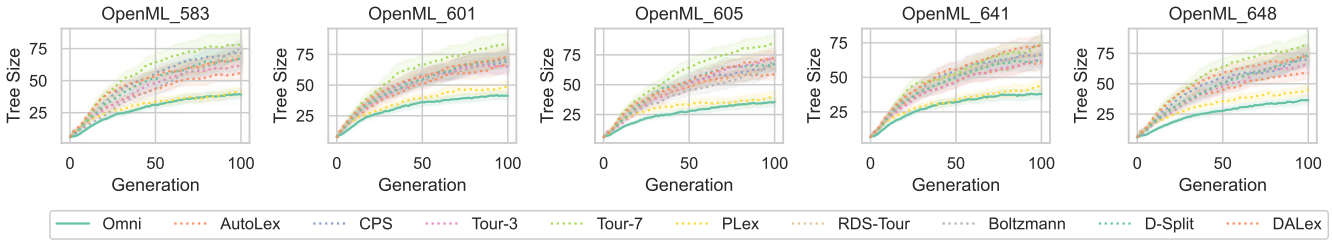

Figure 15: Tree sizes across generations.

```
    best_candidates]
70              winner_idx = best_candidates
    [np.argmax(best_scores)]
71          else:
72              winner_idx = min_err_idx
73          selected.append(population[
    chosen[winner_idx]])
74      return selected
75
```

Code 2: Omni Selection Zero

Benchmark Analysis The Omni-Zero operator is evaluated on the SRBench datasets (La Cava et al. 2021), and the results are shown in Figure 16. As shown by the results, the proposed method achieves medium performance among the benchmarked algorithms. It performs significantly better than tournament selection, as indicated by the p-values in Figure 17. However, compared to selection operators that consider specificity, such as the lexicase selection operator, the evolved selection operator is inferior.

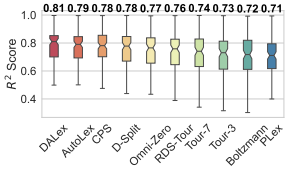

Figure 16: Test R^2 scores of different selection operators on symbolic regression benchmarks, including Omni-Zero.

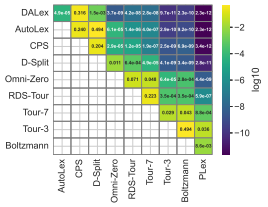

Figure 17: Pairwise statistical comparison of selection operators using the Wilcoxon signed-rank test with Benjamini–Hochberg correction.

C.3 A Repaired Version of Omni Selection for Small Datasets

Problem Analysis Upon closely analyzing the code generated by the LLM in Code 1, we identified a logical bug that can lead to poor performance of the selection operator on small datasets. When the dataset is small, the structured division in line 10 of Code 1 does not yield enough samples to form sufficient subsets, which can result in some

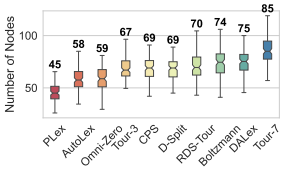

Figure 18: Tree sizes of different selection operators on symbolic regression benchmarks, including Omni-Zero.

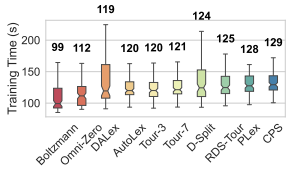

Figure 19: Training times of different selection operators on symbolic regression benchmarks, including Omni-Zero.

Table 3: Summary of datasets used in meta-evolution.

Dataset	n_observations	n_features
OPENML_505	240	124
OPENML_4544	1059	117
OPENML_588	1000	100
OPENML_650	500	50

subsets being empty. These empty subsets cause the subsequent selection process to rely solely on complexity when selecting individuals, without considering their predictive performance. This issue arises because the datasets used during meta-evolution generally contain a relatively large number of training instances, as shown in Table 3. Consequently, the bug has limited impact during training but becomes problematic when applied to smaller datasets.

Solution To address this issue, we propose a repaired version of the Omni selection operator, as shown in Code 3. In this version, random subsets are used to replace any empty subsets generated during structured division. All other parts of the code remain unchanged.

Benchmark Analysis We compare the performance of the repaired and original versions on datasets with no more than 100 training instances. The results, presented in Figure 20, show that the repaired version outperforms the original, indicating that the logical bug in the original version is indeed problematic. Regarding model size, as shown in Figure 21, the repaired version results in an increased tree size, which is reasonable because it corrects the previous behavior of selecting individuals solely based on complexity by also con-

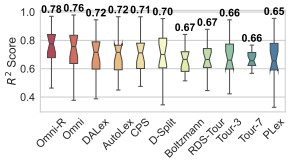

Figure 20: Test R^2 scores of different selection operators on small-scale symbolic regression benchmarks, including the repaired omni selection.

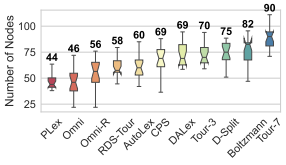

Figure 21: Tree sizes of different selection operators on small-scale symbolic regression benchmarks, including the repaired omni selection.

sidering the error on subsets. Nonetheless, the increase in tree size is not substantial and still leads to a competitive model size compared to other selection operators.

This failure highlights that, when using LLMs for meta-evolution, it is important that the LLM is exposed to a wide range of instances. Otherwise, the generated code may contain subtle logical bugs that are not easily detected.

```python
def omni_selection(population, k=100, status={}):
    stage = np.clip(status.get("evolutionary_stage", 0), 0, 1)
    n = len(population)
    half_k = k // 2
    y = population[0].y
    n_cases = y.size

    ssize = max(7, n_cases // max(1, 2 * half_k))
    half_struct = half_k // 2
    structured = [
        np.arange(i * ssize, min((i + 1) * ssize, n_cases)) for i in range(half_struct)
    ]
    # Filter out empty subsets
    structured = [s for s in structured if s.size > 0]
    random_ = [
        np.random.choice(n_cases, ssize, replace=False)
        for _ in range(half_k - len(structured))
    ]
    subsets = structured + random_

    residuals = np.vstack([ind.y - ind.predicted_values for ind in population])
    complexity = np.array([len(ind) + ind.height for ind in population], float)
    complexity /= max(1, complexity.max())
    subset_mse = np.array(
        [
            [((residuals[i, s]) ** 2).mean() if s.size else np.inf for s in subsets]
            for i in range(n)
        ]
    )
    comp_factor = 0.25 + 0.25 * stage

    parent_a = [
        population[np.lexsort((complexity, subset_mse[:, i]))[0]]
        for i in range(len(subsets))
    ]

    idx_map = {ind: i for i, ind in enumerate(population)}
    norms = np.linalg.norm(residuals, axis=1) + 1e-12

    parent_b = []
    for a in parent_a:
        i_a = idx_map[a]
        res_a = residuals[i_a]
        cors = (residuals @ res_a) / (norms * norms[i_a])
        cors[i_a] = 1
        comp_score = np.abs(cors) + comp_factor * complexity
        b_idx = np.argmin(comp_score)
        parent_b.append(population[b_idx])

    return [ind for pair in zip(parent_a, parent_b) for ind in pair][:k]
```

Code 3: The Repaired Version of Omni Selection (Omni-R)

D Further Analysis of the Semantic-Aware Evolution

In this paper, we propose a semantic-aware evolutionary approach that explicitly selects complementary pairs of individuals for crossover. To illustrate how two complementary codes work together to generate improved solutions, we provide a concrete example in this section. In the existing literature, the most common approach to combining the capabilities of two selection operators is to embed them within a dynamic algorithm selection framework (Xue et al. 2022), allowing each operator to be applied under different conditions. While this strategy permits both operators to contribute, it does so sequentially and without integrating their underlying semantics. Moreover, repeatedly combining two code fragments over multiple generations can lead to exponential code growth (Martins et al. 2018), impairing interpretability and maintainability. In contrast, our approach enables a more meaningful integration by combining semantically complementary code fragments through crossover. As shown in Code 4 and its parent codes in Code 5 and Code 6, the resulting offspring code fragments are not simply concatenations but structured combinations of meaningful components from both parents, as detailed in Table 4. Compared to the mutation operator, which performs a random search around

a single solution, crossover provides a directional search by explicitly fusing different capabilities. Thus, the generated code is expected to perform well across multiple datasets.

As shown in Figure 22, parent 1 and parent 2 excel on dataset 4 and dataset 3, respectively. By combining their complementary strengths through the mixing strategy performed by the LLM in Table 4, the offspring operator achieves strong performance on both datasets 3 and 4, thereby yielding better average performance. Unlike random subtree crossover in tree-based genetic programming (Banzhaf et al. 1998) or one-point crossover in linear genetic programming (Huang et al. 2024), LLM-based crossover can synthesize code that semantically resembles an intermediate between the two parent programs by leveraging its code understanding capability. This highlights the importance of semantic diversity: if both parents are highly similar, even if performant, the resulting offspring would be nearly identical to the parents, leading to redundant evaluations and wasted computation. Therefore, semantic-aware evolution is vital for automated algorithm design.

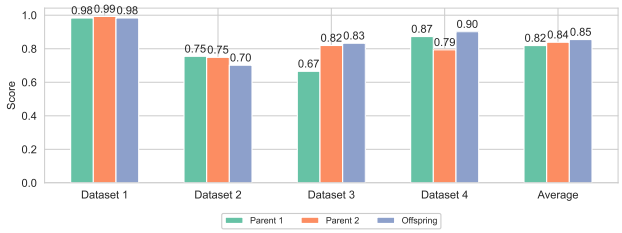

Figure 22: Validation R^2 scores of parent and offspring selection operators on different datasets.

```python
def omni_selection(population, k=100, status={}):
    stage = status.get("evolutionary_stage", 0)
    n = len(population)
    half_k = k // 2
    rng = np.random.default_rng(12345)

    errs = np.array([ind.case_values for ind in population])           # (n, n_cases)
    residuals = np.array([ind.y - ind.predicted_values for ind in population])
    sizes = np.array([len(ind) for ind in population])
    heights = np.array([ind.height for ind in population])

    n_cases = errs.shape[1]
    comp_pen = (sizes + heights) * (0.4 + 0.6 * stage) / (40 + 60 * stage)  # adaptive comp penalty

    subset_size = max(8, n_cases // (half_k + 2))
    max_tries = 15 * half_k
    tried_subsets = set()
    full_mse = errs.mean(axis=1)

    parent_a = []
    tries = 0
    while len(parent_a) < half_k and tries < max_tries:
        tries += 1
        subset = tuple(sorted(rng.choice(n_cases, subset_size, replace=False)))
        if subset in tried_subsets:
            continue
        tried_subsets.add(subset)

        mse_sub = errs[:, subset].mean(axis=1)
        specialization = full_mse - mse_sub  # + means better on subset specialization
        scores = (
            (1 - stage) * specialization +
            stage * (1 / (full_mse + 1e-10))
        ) - comp_pen
        best_idx = np.argmax(scores)
        parent_a.append(population[best_idx])

    if not parent_a:
        parent_a = rng.choice(population, half_k, replace=True).tolist()

    norm_resid = residuals / (np.linalg.norm(residuals, axis=1, keepdims=True) + 1e-10)
    parent_b = []
    comp_factor = 5 if stage < 0.5 else 10  # stronger complexity pressure late
    for a in parent_a:
        a_idx = population.index(a)
        corr = norm_resid @ norm_resid[a_idx]
        corr[a_idx] = 1  # exclude self
        scores = corr + comp_pen / comp_factor
        comp_idx = np.argmin(scores)
        parent_b.append(population[comp_idx])

    selected = [ind for pair in zip(parent_a, parent_b) for ind in pair]
    return selected[:k]
```

Code 4: Offspring Code

```python
def omni_selection(population, k=100, status={}):
    stage = status.get("evolutionary_stage", 0)
    n = len(population)
```

Component	Parent 1	Parent 2	Generated Code
Complexity penalty	$(s+h) \times \frac{0.3 + 0.7\,\text{stage}}{100}$	$\frac{s+h}{30 + 70\,\text{stage}}$	$(s+h) \times \frac{0.4 + 0.6\,\text{stage}}{40 + 60\,\text{stage}}$
Subset size	$\max\left(7, \lfloor \frac{n_c}{\frac{k}{2}+3} \rfloor\right)$	$\max\left(10, \lfloor \frac{n_c}{\frac{k}{2}+1} \rfloor\right)$	$\max\left(8, \lfloor \frac{n_c}{\frac{k}{2}+2} \rfloor\right)$
Parent A scoring	$(1-\text{stage}) \times (\text{mse}_{\text{full}} - \text{mse}_{\text{sub}}) + \text{stage} \times \frac{1}{\text{mse}_{\text{full}}} - \text{comp\_pen}$	$(1-\text{stage}) \times \frac{1}{\text{mse}_{\text{sub}}} + \text{stage} \times \frac{1}{\text{mse}_{\text{full}}} - \text{comp\_pen}$	Same as Parent 1
Parent B scoring	$corr + \frac{\text{comp\_pen}}{1 + 5\,\text{stage}}$	$corr + \frac{\text{comp\_pen}}{10}$	$corr + \frac{\text{comp\_pen}}{\begin{cases} 5, & \text{if stage} < 0.5 \\ 10, & \text{otherwise} \end{cases}}$

Table 4: Fusion of key selection components. Here s = sizes, h = heights, n_c is the number of cases, comp_pen is the complexity penalty, and *corr* denotes the normalized residual correlation.

```
    half_k = k // 2

    errs = np.array([ind.case_values for ind in population])  # (n, n_cases)
    residuals = np.array([ind.y - ind.predicted_values for ind in population])
    sizes = np.array([len(ind) for ind in population])
    heights = np.array([ind.height for ind in population])
    n_cases = errs.shape[1]
    rng = np.random.default_rng(777)

    # Complexity penalty grows with stage, stronger late to favor interpretability
    comp_pen = (sizes + heights) * (0.3 + 0.7 * stage) / 100

    # Parameters for subset selection
    subset_size = max(7, n_cases // (half_k + 3))
    max_tries = 10 * half_k
    tried_subsets = set()
    full_mse = errs.mean(axis=1)

    parent_a = []
    while len(parent_a) < half_k and max_tries > 0:
        max_tries -= 1
        subset = tuple(sorted(rng.choice(n_cases, subset_size, replace=False)))
        if subset in tried_subsets:
            continue
        tried_subsets.add(subset)

        mse_sub = errs[:, subset].mean(axis=1)
        specialization = full_mse - mse_sub  # positive means better on subset
        # Blend specialization and overall fitness with stage-dependent weights
        scores = (1 - stage) * specialization + stage * (1 / (full_mse + 1e-10))
        scores -= comp_pen
        best_idx = np.argmax(scores)
        parent_a.append(population[best_idx])

    if not parent_a:
        parent_a = rng.choice(population, half_k, replace=True).tolist()

    norm_resid = residuals / (np.linalg.norm(residuals, axis=1, keepdims=True) + 1e-10)
    parent_b = []
    for a in parent_a:
        a_idx = population.index(a)
        corr = norm_resid @ norm_resid[a_idx]
        corr[a_idx] = 1
        # Score complementary: minimize correlation and complexity; complexity effect softer early on
        scores = corr + comp_pen / (1 + stage * 5)
        b_idx = np.argmin(scores)
        parent_b.append(population[b_idx])

    selected = [ind for pair in zip(parent_a, parent_b) for ind in pair]
    return selected[:k]
```

Code 5: Code of Parent A

```
def omni_selection(population, k=100, status={}):
    n = len(population)
    half_k = k // 2
    stage = status.get("evolutionary_stage", 0)
    max_tries = 10 * k

    preds = np.array([ind.
```

```
    predicted_values for ind in
    population])
    cases = np.array([ind.case_values
    for ind in population])
    residuals = np.array([ind.y - ind.
    predicted_values for ind in
    population])
    sizes = np.array([len(ind) for ind
    in population])
    heights = np.array([ind.height for
    ind in population])

    n_cases = preds.shape[1]
    subset_size = max(10, n_cases // (
    half_k + 1))
    idx_pool = np.arange(n_cases)
    np.random.seed(42)

    # Complexity penalty scaled by stage
    comp_pen = (sizes + heights) / (30 +
     70 * stage)

    # Score function balances subset/
    full MSE and complexity
    def score(i, subset_idx):
        mse_subset = np.mean((cases[i,
    subset_idx] - preds[i, subset_idx])
    ** 2)
        mse_full = np.mean((cases[i] -
    preds[i]) ** 2)
        return ((1 - stage) * (1 / (
    mse_subset + 1e-10)) + stage * (1 / (
    mse_full + 1e-10))) - comp_pen[i]

    parent_a = []
    tried_subsets = set()
    tries = 0
    while len(parent_a) < half_k and
    tries < max_tries:
        tries += 1
        subset_idx = tuple(sorted(np.
    random.choice(idx_pool, subset_size,
    replace=False)))
        if subset_idx in tried_subsets:
            continue
        tried_subsets.add(subset_idx)
        scores = np.array([score(i,
    subset_idx) for i in range(n)])
        best_idx = np.argmax(scores)
        parent_a.append(population[
    best_idx])

    if not parent_a:
        parent_a = np.random.choice(
    population, half_k, replace=True).
    tolist()

    parent_b = []
    norm_residuals = residuals / (np.
    linalg.norm(residuals, axis=1,
    keepdims=True) + 1e-10)
    for a in parent_a:
        a_idx = population.index(a)
        a_res = norm_residuals[a_idx]
        corr = norm_residuals @ a_res
        corr[a_idx] = 1.0  # exclude
    self
        complement_scores = corr +
    comp_pen / 10  # prefer low corr &
    low complexity
        comp_idx = np.argmin(
    complement_scores)
        parent_b.append(population[
    comp_idx])

    selected_individuals = [ind for pair
     in zip(parent_a, parent_b) for ind
    in pair]
    return selected_individuals[:k]
```

Code 6: Code of Parent B

E Results on More Discovered Operators

The proposed LLM-Meta-SR method can discover various selection operators. For example, Code 4 presents a selection operator discovered by the LLM-Meta-SR algorithm, which differs from the operator shown in Code 1. To avoid confusion, we refer to the discovered operator in Code 4 as "Holo". The results of the test R^2 scores are shown in Figure 23 and Figure 24, which demonstrate that the Holo selection operator outperforms expert-designed selection operators. Additionally, the complexity size, shown in Figure 25, indicates that the discovered operator is competitive in terms of model complexity. A direct comparison with the tree sizes evolved using the Omni selection, reported in Figure 9, shows that the discovered operator is even slightly smaller. Therefore, the "Holo" selection operator is preferred when interpretability is a key concern.

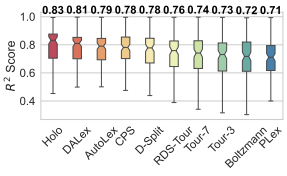

Figure 23: Test R^2 scores of different selection operators on symbolic regression benchmarks, including the Holo selection operator.

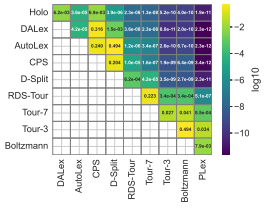

Figure 24: Pairwise statistical comparison of selection operators using the Wilcoxon signed-rank test with Benjamini–Hochberg correction.

F Test Cases for Synthetic Evaluation

For the synthetic evaluation in Section 3.2, two synthetic test cases are designed to reveal implementation flaws and inefficiencies early in the evaluation process, prior to applying each evolved selection operator to real symbolic regression tasks. The first case consists of 100 solutions whose predicted values and training errors are independently sampled from 100 random integers in the range $[1, 10]$. The second case also

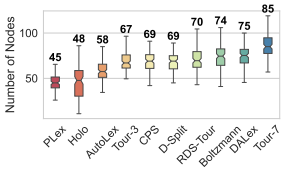

Figure 25: Tree sizes of different selection operators on symbolic regression benchmarks, including the Holo selection operator.

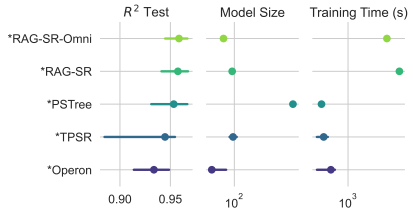

Figure 26: Median R^2 scores, model sizes, and training time of top-5 algorithms on the symbolic regression benchmark.

consists of 100 solutions, but with both predicted values and training errors set to a constant integer, repeated across all individuals, based on a value sampled from $[1, 10]$. The first case represents a scenario with diverse inputs and varying fitness values, while the second serves as an edge case where all individuals have identical fitness.

G More Analysis on Symbolic Regression Benchmark

The results of the top-5 performing algorithms are shown in Figure 26 for better clarity, and the Pareto front of the ranks of test R^2 scores and model sizes across different algorithms is shown in Figure 27. These results demonstrate that RAG-SR-Omni is a Pareto-optimal algorithm on SRBench. Specifically, RAG-SR-Omni dominates retrieval-augmentation-generation-based symbolic regression (RAG-SR) (Zhang et al. 2025) as well as transformer-based planning for symbolic regression (Shojaee et al. 2024). Overall, these results show the effectiveness of the LLM-evolved selection operator.

H Baseline Selection Operators

The hyperparameters for the baseline selection operators follow the default settings specified in their respective original papers. For RDS-Tour, the sampling ratio is set to 10% and the tournament size to 7 (Geiger et al. 2025). For PLex selection, the temperature parameter is set to 1.0 (Ding, Pantridge, and Spector 2023). For DALex, a particularity pressure of 3 is used (Ni, Ding, and Spector 2024). For CPS, the first parent is selected via tournament, with the tournament size set to 7 (Xu et al. 2022).

For Boltzmann sampling with temperature scheduling (Shojaee et al. 2025; Romera-Paredes et al. 2024), a temperature decay rule is applied to balance exploration

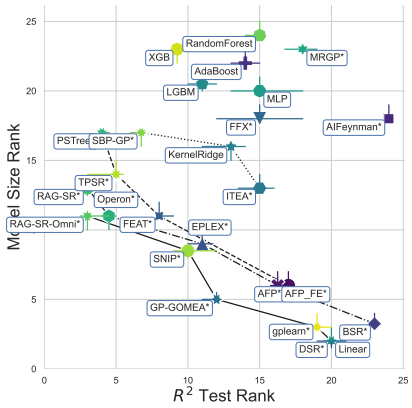

Figure 27: Pareto front of the ranks of test R^2 scores and model sizes for different algorithms.

and exploitation over time, similar to the approaches used in LLM-SR (Shojaee et al. 2025) and FunSearch (Romera-Paredes et al. 2024). The sampling probability for individual i is defined as:

$$P_i = \frac{\exp\left(\frac{s_i}{\tau_c}\right)}{\sum_{i'} \exp\left(\frac{s_{i'}}{\tau_c}\right)}, \quad \tau_c = \tau_0 \left(1 - \frac{t \bmod T_{\max}}{T_{\max}}\right), \tag{4}$$

where s_i denotes the score of individual i, $\tau_0 = 0.1$ is the initial temperature (Shojaee et al. 2025), t is the current generation, and $T_{\max}$ is the total number of generations. This scheduling rule linearly decays the temperature over generations and gradually shifts the selection pressure from exploration (high temperature) to exploitation (low temperature) as evolution progresses in evolutionary symbolic regression.

I Further Investigation on Crossover and Mutation Ratio

In the main paper, the number of individuals in the population generated by crossover is 19, and the number generated by mutation is 1. In this section, we investigate the proposed method under different settings of crossover and mutation ratios to examine how these ratios affect its performance. Specifically, in this section, the number of individuals generated by crossover is 15 and by mutation is 5. The results related to validation score and code length are shown in Figure 28 and Figure 29, respectively.

First, the experimental results in Figure 28 and Figure 29 show that the conclusions from the ablation studies in Section 5.2 generalize to different settings of crossover and mutation ratios. This includes the effectiveness of semantics-based evolution, bloat control, and domain-knowledge-based operators.

Second, the results in Figure 28 as well as the numerical values in Table 5 indicate that the optimal crossover and mutation ratios vary depending on the prompts. When domain knowledge is available, a smaller mutation rate, as used in Section 5.2, achieves slightly better performance than the larger mutation rate used in this section. In contrast, when

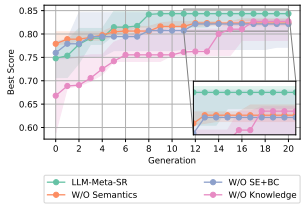

Figure 28: Validation R^2 of the best solution across generations for different LLM-driven search strategies in the case of large mutation rate.

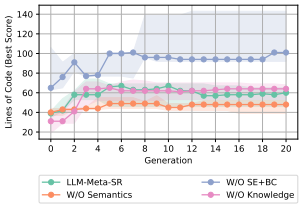

Figure 29: Code length of the best solution across generations for different LLM-driven search strategies in the case of large mutation rate.

domain knowledge is not available, a larger mutation rate appears to be more beneficial. One possible reason is that when domain knowledge is available, the search space is more confined and a large mutation rate does not provide additional benefit. Conversely, when domain knowledge is not available, the search space is larger and a higher mutation rate is needed to effectively explore the space.

J Related Work

J.1 Recent Advancements in Selection Operators

In recent years, lexicase selection has shown strong empirical performance and is widely adopted in modern symbolic regression systems (Cava et al. 2019; Zhang et al. 2025). Inspired by lexicase selection, several variants have been developed, including probabilistic lexicase selection (Ding, Pantridge, and Spector 2023), lexicase-like selection via diverse aggregation (DALex) (Ni, Ding, and Spector 2024), ϵ-lexicase selection with dynamic split (D-Split) (Imai Aldeia, De França, and La Cava 2024), down-sampled lexicase selection (Boldi et al. 2024), and random down-sampled tournament selection (Geiger, Sobania, and Rothlauf 2025). These methods share the core idea of emphasizing performance on subsets of training instances to encourage specialization, and they demonstrate advantages in various scenarios (Geiger, Sobania, and Rothlauf 2025). However, these operators are manually designed and do not fully satisfy all the desired properties outlined in Section 3.5, highlighting the need for further exploration of selection operators.

J.2 Automated Evolutionary Algorithm Design

Before the LLM era, genetic programming had been widely used for automated evolutionary algorithm design, including the design of fitness functions (Fong and Motani 2024a), parameter adaptation techniques (Stanovov, Akhmedova, and Semenkin 2022), and update strategies (Lones 2021). More recently, reinforcement learning has also been applied to designing update rules in differential evolution (Chen et al. 2024a). However, these approaches are typically limited to generating only small code segments, as exploring large program spaces remains challenging. The emergence of LLMs has made automated algorithm design more feasible. For example, LLMs have been used to design a differential evolution algorithm competitive with state-of-the-art continuous optimization methods (Stein and Bäck 2024). Similarly, LLMs have been employed to improve the self-organizing migrating algorithm (Pluhacek et al. 2024). This growing interest in LLMs motivates the development of efficient LLM-based approaches for automated algorithm design, where semantic awareness and code bloat are two key issues that remain underexplored and warrant further investigation.

J.3 Large Language Models for Optimization

LLMs have been employed as optimization tools across various domains (Meyerson et al. 2024; Song et al. 2024), including neural architecture search (Chen, Dohan, and So 2023), graph neural network design (Mo et al. 2025), large neighborhood search algorithm design (Ye et al. 2025), reward function design (Ma et al. 2024), the traveling salesman problem (Liu et al. 2024b), prompt optimization (Yang et al. 2024), automated machine learning (Chan et al. 2025), solver code generation (Jiang et al. 2025), symbolic regression (Shojaee et al. 2025), and feature engineering (Gong et al. 2025). Evolutionary search has gained considerable attention in the LLM era due to its effectiveness over random sampling (Zhang et al. 2024). Various efforts have aimed to enhance LLM-based evolutionary search, including group relative policy optimization (Huang et al. 2025b), direct preference optimization (Šurina et al. 2025), core abstraction prompting (Wu et al. 2025), quality-uncertainty optimization (Chen et al. 2024b), and bi-dimensional co-evolution (Zhao et al. 2025).

Exploring the potential of LLMs in automating the design of improved evolutionary search strategies is a worthwhile direction. However, evaluating LLM-based evolutionary search in automated algorithm design is computationally expensive. In comparison, genetic programming is a relatively inexpensive and historically dominant method for code generation prior to the LLM era (Peabody and Seitzer 2015; Chen et al. 2023; Maudet and Danoy 2025). Therefore, leveraging LLMs for the automated design of genetic programming algorithms presents a relevant and cost-effective avenue for investigation.

K Prompt for Algorithm Evolution

System Prompt: Figure 30 shows the system prompt used for algorithm evolution, which is added before the specific prompt in all three phases of the automated algorithm design process, including initialization, crossover, and mutation. The system prompt is designed to guide the LLM in generating efficient selection operators by preferring NumPy vectorized operations and avoiding explicit Python for-loops unless absolutely necessary. Python is chosen for its simplicity and strong support in LLMs, but standard for-loops are time-consuming. In contrast, NumPy vectorized operations are implemented in C++ and offer better computational speed. The system prompt also specifies the maximum number of lines of code that the LLM can generate. This constraint helps control bloat, as discussed in Section 3.4.

Initialization/Mutation Prompt: Figure 31 presents the prompts used for initialization and mutation. The main goal of these prompts is to encourage the LLM to generate a novel selection operator to explore the search space. The key

Table 5: Historical best scores and corresponding code lengths for each algorithm in the case of large mutation rate.

	LLM-Meta-SR	W/O Semantics	W/O SE+BC	W/O Knowledge
Score	0.84	0.82	0.82	0.83
Lines of Code	64	48	101	64

difference between the prompts used for the two phases is that in the initialization phase, no baseline code is provided to the LLM. In the mutation phase, the elite solution is provided as the baseline code. The prompt format for wrapping the baseline code is shown in Figure 32, which aims to encourage the LLM to generate a novel selection operator based on the provided code.

Crossover Prompt: Figure 33 shows the crossover prompt used for algorithm evolution. The goal is specified as "selection operator," and its plural form, "selection operators". The crossover prompt takes the code of two existing selection operators as input and aims to generate a better operator by combining effective building blocks from both. To support semantic awareness and concise code generation, the prompt includes the score vector and the corresponding line of code. The *properties* describe the desired characteristics of the selection operator based on domain knowledge from Section 3.5. The specific prompt is shown in Figure 34. The *template* provides a function signature to ensure that the generated operator can be integrated into the automatic evaluation pipeline as shown in Code 7. When domain knowledge is not provided, the *properties* are left empty and a simplified template without knowledge guidance for the selection operator, as shown in Code 8, is used.

Fallback Selection Operator: In rare cases where the LLM fails to generate valid Python code, a simple tournament selection operator, as shown in Code 9, is used to fill the population.

> When writing code, prefer NumPy vectorized operations and avoid explicit Python for-loops unless absolutely necessary. Please implement code within {max_code_lines} lines.

Figure 30: System prompt for evolving selection operators.

> Your task is to develop an innovative and novel {goal} for symbolic regression using genetic programming in Python.
>
> {Baseline}
> {Properties}
>
> Ensure that your newly designed function adheres to the following signature:
> {Template}
>
> You do not need to provide a usage example.
>
> Embrace creativity, novelty, and bold experimentation to push the boundaries of the state of the art in {goals} for genetic programming.

Figure 31: Initialization and mutation prompt for designing an LLM-based selection operator. For initialization, the baseline is empty.

> Inspirational Example:
>
> {Code}
>
> Use this as inspiration to create a distinctly original and inventive {goal}.

Figure 32: Prompt embedded in the baseline.

You are tasked with designing a novel {goal} for symbolic regression using genetic programming.

Your goal is to synthesize a new operator that combines the strengths and mitigates the weaknesses of the two operators shown below:

— Operator A —
Score (Higher is Better): {RoundToDecimalPlaces(Score_A, 3)}, Lines of Code: {EffectiveLineCount(OperatorA)}
Code:
{OperatorA_Code}

— Operator B —
Score (Higher is Better): {RoundToDecimalPlaces(Score_B, 3)}, Lines of Code: {EffectiveLineCount(OperatorB)}
Code:
{OperatorB_Code}

{Properties}

Ensure that your newly designed function adheres to the following signature:
{Template}

You do not need to provide a usage example.

Figure 33: Prompt used for crossover to design a novel selection operator with an LLM.

1. Diverse & Specialized Selection
– Choose a varied set of high-performing individuals.
– Encourage specialization to maintain a diverse population.

2. Crossover-Aware Pairing
– Promote complementarity between parents.

3. Stage-Specific Pressure
– Vary selection pressure based on the current stage of evolution.

4. Interpretability
– Prefer individuals with fewer nodes and lower tree height to improve model interpretability.

5. Efficiency & Scalability
– Include clear stopping conditions to avoid infinite loops.

6. Code Simplicity
– Favor clear, concise logic with minimal complexity.

Figure 34: Desired properties of good selection operators based on domain knowledge.

```python
def selection(population, k=100, status={}):
    # Useful information about individuals:
    # squared_error_vector = individual.case_values
    # predicted_values = individual.predicted_values
    # residual = individual.y-individual.predicted_values
    # number_of_nodes = len(individual)
    # height = individual.height

    # Useful information about evolution:
    # status["evolutionary_stage"]: [0,1], where 0 is the first generation and 1 is the final generation

    parent_a = Select k//2 individuals based on performance over subsets of instances:
        - evaluate individuals on k//2 different random or structured subsets of the data
        - reward those that specialize or perform well on those subsets
        - may also consider overall fitness (e.g., full-dataset MSE)
        - optionally include structural complexity
        - an individual can be selected multiple times

    parent_b = For each parent_a, select a complementary individual:
        - low residual correlation with parent_a
        - may also consider complexity

    # Interleave parent_a and parent_b to form crossover pairs
    selected_individuals = [individual for pair in zip(parent_a, parent_b) for individual in pair]
    return selected_individuals
```

Code 7: Template for selection operator evolution.

```python
def selection(population, k=100, status={}):
    # Useful information about individuals:
    # squared_error_vector = individual.case_values
    # predicted_values = individual.predicted_values
    # residual = individual.y-individual.predicted_values
    # number_of_nodes = len(individual)
    # height = individual.height

    # Useful information about evolution:
    # status["evolutionary_stage"]: [0,1], where 0 is the first generation and 1 is the final generation

    # Implement selection logic here
    return selected_individuals
```

Code 8: Template for selection operator evolution without domain knowledge.

```python
import numpy as np
import random

def selection(individuals, k=100, tour_size=3):
    # Tournament selection
    # Compute mean error for each individual
    mean_errors = [np.mean(ind.case_values) for ind in individuals]
    selected_individuals = []

    # Select `k` individuals
    for _ in range(k):
        # Randomly select competitors
        competitors = random.sample(range(len(individuals)), tour_size)
        # Select the one with the lowest error
        best_idx = min(competitors, key=lambda idx: mean_errors[idx])
        selected_individuals.append(individuals[best_idx])

    return selected_individuals
```

Code 9: Default code used when no code can be extracted from the LLM response.

L Computing Infrastructure

All experiments are run on AMD Milan CPUs. The software packages used in our experiments are listed in Table 6.

Table 6: Software used for experiments.

Type	Name
Code	DEAP (Fortin et al. 2012)
Code	CodeBLEU (Ren et al. 2020)
Benchmark	SRBench (La Cava et al. 2021)

Reproducibility Checklist

Instructions for Authors:
This document outlines key aspects for assessing reproducibility. Please provide your input by editing this `.tex` file directly.

For each question (that applies), replace the "Type your response here" text with your answer.

Example: If a question appears as

```
\question{Proofs of all novel claims
are included} {(yes/partial/no)}
Type your response here
```

you would change it to:

```
\question{Proofs of all novel claims
are included} {(yes/partial/no)}
yes
```

Please make sure to:

- Replace ONLY the "Type your response here" text and nothing else.
- Use one of the options listed for that question (e.g., **yes**, **no**, **partial**, or **NA**).
- **Not** modify any other part of the `\question` command or any other lines in this document.

You can `\input` this .tex file right before `\end{document}` of your main file or compile it as a stand-alone document. Check the instructions on your conference's website to see if you will be asked to provide this checklist with your paper or separately.

1. General Paper Structure

1.1. Includes a conceptual outline and/or pseudocode description of AI methods introduced (yes/partial/no/NA) yes

1.2. Clearly delineates statements that are opinions, hypothesis, and speculation from objective facts and results (yes/no) yes

1.3. Provides well-marked pedagogical references for less-familiar readers to gain background necessary to replicate the paper (yes/no) yes

2. Theoretical Contributions

2.1. Does this paper make theoretical contributions? (yes/no) no

If yes, please address the following points:

2.2. All assumptions and restrictions are stated clearly and formally (yes/partial/no) NA

2.3. All novel claims are stated formally (e.g., in theorem statements) (yes/partial/no) NA

2.4. Proofs of all novel claims are included (yes/partial/no) NA

2.5. Proof sketches or intuitions are given for complex and/or novel results (yes/partial/no) NA

2.6. Appropriate citations to theoretical tools used are given (yes/partial/no) NA

2.7. All theoretical claims are demonstrated empirically to hold (yes/partial/no/NA) NA

2.8. All experimental code used to eliminate or disprove claims is included (yes/no/NA) NA

3. Dataset Usage

3.1. Does this paper rely on one or more datasets? (yes/no) yes

If yes, please address the following points:

3.2. A motivation is given for why the experiments are conducted on the selected datasets (yes/partial/no/NA) yes

3.3. All novel datasets introduced in this paper are included in a data appendix (yes/partial/no/NA) NA

3.4. All novel datasets introduced in this paper will be made publicly available upon publication of the paper with a license that allows free usage for research purposes (yes/partial/no/NA) NA

3.5. All datasets drawn from the existing literature (potentially including authors' own previously published work) are accompanied by appropriate citations (yes/no/NA) yes

3.6. All datasets drawn from the existing literature (potentially including authors' own previously published work) are publicly available (yes/partial/no/NA) yes

3.7. All datasets that are not publicly available are described in detail, with explanation why publicly available alternatives are not scientifically satisficing (yes/partial/no/NA) NA

4. Computational Experiments

4.1. Does this paper include computational experiments? (yes/no) yes

If yes, please address the following points:

4.2. This paper states the number and range of values tried per (hyper-) parameter during development of the paper, along with the criterion used for selecting the final parameter setting (yes/partial/no/NA) yes

4.3. Any code required for pre-processing data is included in the appendix (yes/partial/no) yes

4.4. All source code required for conducting and analyz-

ing the experiments is included in a code appendix (yes/partial/no) partial

4.5. All source code required for conducting and analyzing the experiments will be made publicly available upon publication of the paper with a license that allows free usage for research purposes (yes/partial/no) yes

4.6. All source code implementing new methods have comments detailing the implementation, with references to the paper where each step comes from (yes/partial/no) yes

4.7. If an algorithm depends on randomness, then the method used for setting seeds is described in a way sufficient to allow replication of results (yes/partial/no/NA) yes

4.8. This paper specifies the computing infrastructure used for running experiments (hardware and software), including GPU/CPU models; amount of memory; operating system; names and versions of relevant software libraries and frameworks (yes/partial/no) yes

4.9. This paper formally describes evaluation metrics used and explains the motivation for choosing these metrics (yes/partial/no) yes

4.10. This paper states the number of algorithm runs used to compute each reported result (yes/no) yes

4.11. Analysis of experiments goes beyond single-dimensional summaries of performance (e.g., average; median) to include measures of variation, confidence, or other distributional information (yes/no) yes

4.12. The significance of any improvement or decrease in performance is judged using appropriate statistical tests (e.g., Wilcoxon signed-rank) (yes/partial/no) yes

4.13. This paper lists all final (hyper-)parameters used for each model/algorithm in the paper's experiments (yes/partial/no/NA) yes