Modular Multitree Genetic Programming for Evolutionary Feature Construction for Regression

Hengzhe Zhang, *Member, IEEE*, Qi Chen, *Member, IEEE*, Bing Xue, *Senior Member, IEEE*, Wolfgang Banzhaf, *Member, IEEE*, and Mengjie Zhang, *Fellow, IEEE*

***Abstract*—Evolutionary feature construction is a key technique in evolutionary machine learning, with the aim of constructing high-level features that enhance performance of a learning algorithm. In real-world applications, engineers typically construct complex features based on a combination of basic features, reusing those features as modules. However, modularity in evolutionary feature construction is still an open research topic. This article tries to fill that gap by proposing a modular and hierarchical multitree genetic programming (GP) algorithm that allows trees to use the output values of other trees, thereby representing expressive features in a compact form. Based on this new representation, we propose a macro parent-repair strategy to reduce redundant and irrelevant features, a macro crossover operator to preserve interactive features, and an adaptive control strategy for crossover and mutation rates to dynamically balance the tradeoff between exploration and exploitation. A comparison with seven bloat control methods on 98 regression datasets shows that the proposed modular representation achieves significantly better results in terms of test performance and smaller model size. Experimental results on the state-of-the-art symbolic regression benchmark demonstrate that the proposed symbolic regression method outperforms 22 existing symbolic regression and machine learning algorithms, providing empirical evidence for the superiority of the modularized evolutionary feature construction method.**

***Index Terms*—Evolutionary feature construction, evolutionary forest, genetic programming (GP), modularity, random forest.**

Manuscript received 14 May 2023; revised 1 August 2023; accepted 11 September 2023. Date of publication 25 September 2023; date of current version 3 October 2024. This work was supported in part by the Marsden Fund of New Zealand Government under Contract VUW1913, Contract VUW1914, and Contract VUW2016; in part by the Science for Technological Innovation Challenge (SfTI) Fund under Grant E3603/2903; in part by the MBIE Data Science SSIF Fund under Contract RTVU1914; in part by the Huayin Medical under Grant E3791/4165; and in part by the MBIE Endeavor Research Programme under Contract C11X2001 and Contract UOCX2104. *(Corresponding author: Qi Chen.)*

Hengzhe Zhang, Qi Chen, Bing Xue, and Mengjie Zhang are with the Centre for Data Science and Artificial Intelligence and the School of Engineering and Computer Science, Victoria University of Wellington, Wellington 6140, New Zealand (e-mail: hengzhe.zhang@ecs.vuw.ac.nz; qi.chen@ecs.vuw.ac.nz; bing.xue@ecs.vuw.ac.nz; mengjie.zhang@ecs.vuw.ac.nz).

Wolfgang Banzhaf is with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824 USA (e-mail: banzhafw@msu.edu).

This article has supplementary material provided by the authors and color versions of one or more figures available at https://doi.org/10.1109/TEVC.2023.3318638.

Digital Object Identifier 10.1109/TEVC.2023.3318638

I. Introduction

AUTOMATED feature construction is an important task in machine learning. The goal of feature construction is to construct expressive features Φ based on the original features X to enhance the performance of existing machine learning algorithms [1]. A good feature representation can facilitate the learning process for an algorithm, compared to the original feature space.

In machine learning, various representation learning methods like non-linear dimensionality reduction, kernel methods [2], and deep neural networks [3] have been developed to construct expressive features. Among these, neural-network-based deep learning algorithms achieve outstanding performance in computer vision and natural language processing. However, in tabular data learning tasks, they often overfit the training data and cannot generalize well on unseen data [4]. Additionally, the large number of parameters makes neural-network-based methods challenging to interpret [5]. In recent years, evolutionary feature construction methods have shown promising results in constructing expressive features for tabular data [1], [6]. The general idea behind evolutionary feature construction methods is to iteratively optimize a set of features use evolutionary algorithms, aiming to improve the generalization performance of the learned model.

In the evolutionary feature construction domain, genetic programming (GP)-based methods have demonstrated outstanding performance in various scenarios. The flexible representation and gradient-free search mechanism make GP attractive for feature construction, particularly for non-differentiable models and high-order features. Based on the evaluation method, GP-based evolutionary feature construction methods can be classified as filterbased [7], wrapper-based [1], [8], and embedded [9] methods. Filter methods evaluate features based on information gain [10], correlation [11], and other measures. Wrapper methods evaluate features based on a specific machine learning algorithm and use the model's performance as fitness value. Embedded methods construct features during the learning process, such as solely using GP for symbolic regression [12].

Recently, multitree genetic programming (GP) methods have become increasingly prevalent in evolutionary feature construction tasks, demonstrating impressive results [1], [8]. The key idea behind multitree GP is similar to ensemble

learning, in which several weak GP trees are combined to obtain good predictive capabilities. Although multitree GP achieves good predictive performance, it also increases model size, which often reduces interpretability [13]. Moreover, increasing model size may not always lead to improved performance, which has spurred the development of numerous model size control methods [14], [15], [16], [17]. One key reason for the large model size in multitree GP is that many identical building blocks appear in different trees in a GP individual. When delving deeper into this problem, it becomes evident that GP researchers typically assume that the evolved features can only take original features as inputs. However, in real-world feature construction scenarios, machine learning engineers often develop basic features at first, such as $\phi_1 = x + y$ and $\phi_2 = x - y$, and then construct higher-order features using these basic features, such as $\phi_1 * \phi_2$. Similarly, in deep learning, deep neural networks construct higher-order features hierarchically [3]. Therefore, it would be sensible to develop a modular and hierarchical GP system that can automatically maintain shared building blocks to reduce model size and improve search effectiveness. In this article, modularity denotes the presence of reusable building blocks within the system, and hierarchy refers to arranging these building blocks in a tiered structure based on their relationships. In other words, a modular and hierarchical GP system should explicitly maintain some basic low-order GP-constructed features as building blocks to be jointly used by high-order GP-constructed features, allowing for more effective feature construction.

Based on this analysis, one idea is that trees ϕ_i in a multitree GP system can take not only the original features as inputs but also the output of other trees. Based on this idea, we propose a modular multitree GP (MMTGP) system in this article that evolves a compact GP model with high accuracy.[1] Considering that MMTGP is a special multitree GP with a strong restriction in tree sizes, to leverage the representation of MMTGP, we further propose a macro parent repair strategy to reduce the number of redundant and irrelevant features in MMTGP. Additionally, we introduce an adaptive parameter control strategy for controlling crossover and mutation rates and a macro crossover operator to increase search effectiveness. In summary, the main objectives of this article are as follows.

1) To develop a compact MMTGP representation that allows later GP trees in an individual to use the outputs from previous GP trees, forming a modular GP system with minimal changes to existing multitree GP algorithms. The proposed modular GP is equipped with a layer constraint, a scope constraint, and a sliding window connection style to restrict the search space and explicitly encourage modularization to improve search effectiveness.
2) To design a macro parent repairing mechanism that can repair irrelevant and redundant feature/trees before crossover and mutation. This mechanism facilitates the generation of relevant and nonredundant features in offspring and guides MMTGP in searching for useful features.
3) To develop a macro crossover operator that improves search effectiveness by performing crossover on the individual level rather than the tree level, thus not disrupting the interdependency relationship between different GP trees.
4) Proposing an adaptive crossover and mutation rates control strategy that can dynamically tune the variation rate according to the diversity of GP trees for each index of the multitree GP to balance exploration and exploitation.
5) To investigate whether the proposed compact representation can outperform existing tree GP with bloat control methods in terms of both model size and accuracy, as well as whether it can surpass state-of-the-art symbolic regression and machine learning algorithms.

The remainder of this article is organized as follows. Section II introduces bloat control and modularization techniques in GP as well as related work for evolutionary feature construction. Section III describes the proposed representation and related genetic operators. Section IV presents the experimental settings. Section V reports the experimental results to demonstrate the effectiveness of the proposed representation. Section VI further analyzes the effectiveness of all components in the proposed method. Section VII concludes the article and proposes some future work.

II. Related Work

A. Modularization in Genetic Programming

Modularization is a well-established research topic in GP [18]. Researchers have proposed various GP algorithms for achieving modularization, including Automatically Defined Function (ADF) [19], Tangled Program Graphs (TPG) [20], Cartesian genetic programming (CGP) [21], linear genetic programming (LGP) [22], and stack-based genetic programming (SGP) [23]. ADF is a modularization technique in tree GP that evolves multiple GP trees as functions for use in the primary GP tree, showing impressive performance in discovering complex symbolic models [24]. However, unlike multitree GP, it lacks the ability to combine the outputs of multiple weaker GP trees to enhance predictive performance. CGP, on the other hand, represents computer programs as directed acyclic graphs (DAG) and has been successful in searching for neural network architectures [25]. LGP is another modular GP representation that uses a linear sequence of instructions to represent evolved programs, excelling in solving even parity problems [26]. SGP is a GP technique that uses a stack data structure for program representation and has achieved great success in program synthesis [27]. Recently, a modular and hierarchical GP method named TPG has been proposed, demonstrating impressive performance in multi-task timeseries prediction tasks and visual reinforcement learning tasks [20]. Although these methods have achieved success across domains, an intriguing direction to explore is whether a specialized representation is necessary to address the evolutionary feature construction problem. Recent research on a symbolic regression benchmark [28] suggests that multitree GP [8], [13] outperforms stack-based GP [29]. It would be desirable to slightly modify multi-tree GP to make it modular, rather than reinventing the wheel.

[1] Source Code: https://tinyurl.com/Modular-MTGP.

B. Bloat Control in Genetic Programming

Code bloat is a long-standing issue in GP for over three decades. It refers to solutions becoming increasingly complex without improving the fitness value. Hypotheses for explaining the reasons for bloat include hitchhiking [30], defense against crossover [31], removal bias [32], and the nature of the program search space [33]. Although the reason for bloat is still an ongoing research topic, the benefits of controlling bloat have been widely acknowledged [15], [34], [35]. Generally speaking, bloat control techniques can be categorized as selection-based, variationbased, and evaluation-based methods based on when they are applied in the evolutionary process.

For the evaluation-based methods, parsimony pressure is a traditional method to control bloat by adding tree size to the fitness function with a user-specified weight [36]. However, determining the optimal weight to balance model accuracy and size can be challenging. To circumvent this dilemma, researchers in GP utilize dynamic depth limit tuning strategies to adjust the depth limit based on the size distribution of high-quality individuals, thus preventing the excessive generation of large individuals [34], [37], [38], [39]. In addition to dynamic depth limit, multiobjective evolutionary algorithms (MOEAs) offer an alternative approach to balance accuracy and complexity by generating a set of candidate models with varying degrees of accuracy and complexity in a single run. Then, users can determine the optimal trade-off among the set of Pareto-optimal models [6], [40], [41]. However, MOEA-based methods may favor trivial solutions, i.e., very small solutions with poor performance [42]. For example, in symbolic regression tasks, a study shows that traditional MOEA-based GP (MOGP) methods have more than 30% individuals in the final population containing only one node [42]. To address this, α-dominance relationship [17] and evolvability estimation [42] algorithms are proposed to guide GP search more effectively.

As for selection-based bloat control methods, one such method is lexicographic parsimony pressure [43], which selects parents based on fitness values and uses tree size as a secondary criterion to break ties. Proportional tournament selection and double tournament selection [15] are also developed to control bloat in GP. Among these methods, double tournament selection has been found to be superior for symbolic regression and multiplexer problems [15].

Unlike evaluation-based and selection-based methods, variation-based bloat control methods actively prune GP trees to control bloat. For example, size fair crossover [44] explicitly requires the size of the second subtree s_b in a crossover operator to be less than $s_a * 2 + 1$, where s_a is the size of the first subtree. Moreover, pruning operators, such as hoist mutation [19] and prune and plant (PAP) [45], are also useful for bloat control. These operators replace a subtree with either a subtree from itself or a randomly sampled node.

While variation-based methods can prune GP trees to any size, they may remove useful building blocks without considering program semantics. To address this, program simplification methods have been developed. These methods can be categorized as exact simplification and approximate simplification [46], depending on whether they strictly require semantic equivalence. Exact simplification methods use mathematical rules to simplify GP trees [47] or remove inactive code from GP trees [48]. However, requiring strict semantic equivalence can be too restrictive. In contrast, approximate simplification methods only demand similar semantics after pruning. For example, a subtree can be replaced with a semantically similar subtree within itself [49], or a randomly generated tree with similar semantics [50].

Although bloat control methods based on genetic operators have shown impressive results, they can only reduce model sizes when unnecessary parts are present. To further reduce the size of GP trees that do not contain redundant components, modularized representations is worth investigating.

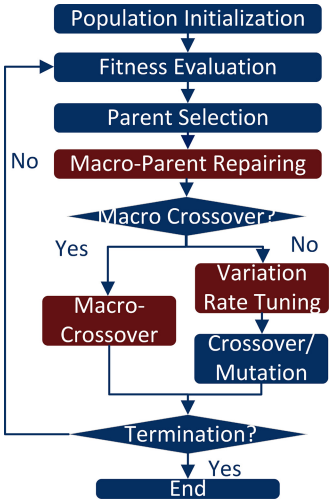


Fig. 1. Overall workflow of MMTGP$_{\text{ALL}}$.

III. New Algorithm

In this section, a novel algorithm with modular representation, named MMTGP, is proposed for GP-based evolutionary feature construction. This section first introduces the overall process of the algorithm. Then, we describe the new representation. Finally, a macro parent repairing mechanism, a macro crossover operator, and an adaptive variation rate tuning strategy are introduced. It is important to note that MMTGP refers only to multitree GP with modular representation, whereas MMTGP with the three proposed strategies (macro parent repairing, macro crossover, and adaptive variation rates) plus an index-aware guided mutation operator is named MMTGP$_{\text{ALL}}$. The index-aware guided mutation operator is introduced in Section I of the supplementary materials, as it is a simple variant of the guided mutation operator [51] adapted to MMTGP.

A. Overall Algorithm

The MMTGP algorithm follows a structure similar to standard GP, as depicted in Fig. 1. The algorithm consists of the following five steps.

1) *Population Initialization:* This stage involves randomly initializing n individuals. Each individual contains m GP trees, initialized using the ramped half-and-half method.
2) *Fitness Evaluation:* In this stage, MMTGP transforms original features X into constructed features

$\{\phi_1(X), \ldots, \phi_m(X)\}$. A linear model is then trained on these constructed features to make a prediction, with the coefficients of the linear model fitted using ridge regression. To encourage the constructed features to generalize well on unseen data, efficient leave-one-out ridge regression is employed to obtain a vector of squared errors $\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$ for each training instance, which is used for parent selection. The fitness metric of each individual is the mean squared error, which is the criterion used for selecting the final model for making predictions on unseen data.

3) *Parent Selection:* After obtaining fitness values for the individuals, parent individuals are selected using the automatic ϵ-lexicase selection operator [52] to preserve good population diversity. The lexicase selection operator randomly selects a case index $k \in [1, n]$ and filters out individuals in an individual pool P with training error larger than $\min_{p \in P} \mathcal{L}_k(p) + \epsilon_k$, where ϵ_k is the mean absolute deviation of fitness values on case k. The individual pool is initialized by filling it with all individuals in the population, and the filtering process is repeated until only one individual remains, which is then selected as the parent.
4) *Offspring Generation:* After selecting parent individuals, random subtree crossover and random subtree mutation operators are applied to vary the parent individuals. In MMTGP, each individual contains m GP trees, so the crossover and mutation operators are performed in m rounds. In each round, a tree $\phi_t \in [\phi_1, \ldots, \phi_m]$ is randomly selected, and the variation operators are applied to the kth GP tree of both parents under the control of variation probability to generate offspring.

B. Modular Multitree GP

In MMTGP, each individual contains m GP trees $\{\phi_1, \ldots, \phi_m\}$. These m trees can construct m features $\{\phi_1(X), \ldots, \phi_m(X)\}$, and a linear model can combine these features to give a final prediction. The primary difference between MMTGP and multitree GP is that MMTGP allows the latter tree ϕ_i to use the outputs of the previous trees $\{\phi_1(X), \ldots, \phi_{i-1}(X)\}$. Fig. 2(a) shows an example of MMTGP, where tree ϕ_4 takes the outputs of $\{\phi_1, \phi_2, \phi_3\}$ and X_2 as inputs. In this way, previous features $\{\phi_1, \phi_2, \phi_3\}$ can be viewed as building blocks.

The naive version of MMTGP, as depicted in Fig. 2(a), does not impose any constraints on the connections, allowing each GP tree ϕ_i to use any outputs of the previous GP trees $\{\phi_1(X), \ldots, \phi_{i-1}(X)\}$ and input variables $\{X_1, \ldots, X_k\}$. However, this may result in an overly large search space when i is large, which can hinder MMTGP from finding good solutions within a limited evaluation budget. To increase search effectiveness, we introduce three constraints in MMTGP to restrict the variables that can be used as inputs.

1) *Scope Constraint:* In MMTGP, we limit each GP tree ϕ_i to use only the outputs of the previous v GP trees $\{\phi_{i-v}(X), \ldots, \phi_{i-1}(X)\}$, thereby alleviating the issue of an overly large search space. For the first v GP trees that do not have enough previous GP trees, their scopes are set to $\{\phi_1(X), \ldots, \phi_{i-1}(X)\}$.
2) *Layer Constraint:* In the naive MMTGP representation, it is possible that all GP trees only use original variables instead of forming a modular representation. To reduce the search space and encourage modularization, we add a layer constraint that restricts GP trees $\{\phi_i | i > v\}$ to only use the output of previous GP trees $\{\phi_{i-v}(X), \ldots, \phi_{i-1}(X)\}$ as inputs and forbids them from using original variables. As for other GP trees $\{\phi_i | i \leq v\}$, they are allowed to use all original features $\{X_1, \ldots, X_n\}$ as well as all outputs from previous GP trees $\{\phi_1(X), \ldots, \phi_{i-1}(X)\}$.
3) *Sliding Window Connection:* In MMTGP, the scope of nodes is determined based on a sliding window rather than a layer boundary. This means that each tree ϕ has a unique scope, which significantly differs from the layer-to-layer architecture in neural networks.

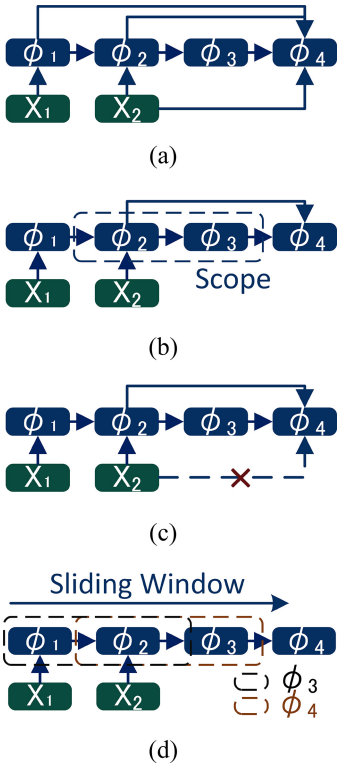


Fig. 2. Illustrative examples of MMTGP with and without constraints. ($\phi_1, \phi_2, \phi_3, \phi_4$ represent four GP trees and solid lines represent active connections between trees). (a) MMTGP without constraints. (b) MMTGP with scope constraint. (c) MMTGP with layer constraint. (d) MMTGP with sliding window.

The differences between the constrained version of MMTGP and the naive MMTGP are illustrated in Fig. 2. In Fig. 2(b), a scope constraint is added, which means ϕ_4 is not allowed to take inputs from ϕ_1. Similarly, a layer constraint was added to MMTGP in Fig. 2(c), and thus ϕ_4 cannot take X_2 as inputs. Fig. 2(d) further shows the sliding window connection style in MMTGP, and thus ϕ_3 and ϕ_4 have their unique connection scopes.

Algorithm 1 Feature Construction in MMTGP

Input: GP Individuals $\Phi = \{\phi_1, \ldots \phi_m\}$, Original Features X
Output: Constructed Features $\phi_1(X), \ldots, \phi_m(X)$

```
FS ← {}
for φ ∈ Φ do
    φ(X) ← Feature Construction(X ∪ FS)
    FS ← FS ∪ {φ(X)}
return φ_1(X), ..., φ_m(X)
```

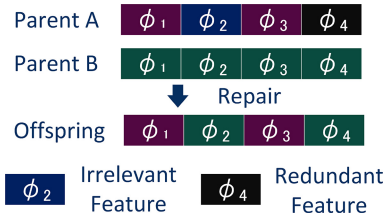


Fig. 3. Macro parent repair strategy in MMTGP.

C. Macro Parent Repairing

In MMTGP, each individual consists of m GP trees representing m constructed features. To ensure a compact representation, we limit the maximum tree depth to 2, i.e., three layers of nodes. Thus, it often constructs irrelevant and redundant features.

1) *Irrelevant Features:* In this article, irrelevant features are defined as those with importance values I_ϕ lower than 0.01 in a fitted machine learning model. For a linear regression model, feature importance I_ϕ is determined by the coefficient assigned to each feature in the trained model, with the precondition of normalizing all features before training the linear model. Lower-importance values indicate that the corresponding features do not help predict the output variable in regression. Thus, to enhance the effectiveness of GP search, it is desirable to repair individuals containing irrelevant features.
2) *Redundant Features:* In evolutionary feature construction, if two GP trees have equivalent semantics, i.e., $\phi_a(X) = \phi_b(X)$, the features constructed by the two trees are considered redundant to each other. Similar to repairing irrelevant features, repairing redundant features can also improve search effectiveness, enabling GP to find better solutions in a limited time.

For repairing irrelevant and redundant features, one feasible idea is to borrow useful features from other individuals. Therefore, before performing crossover and mutation, for each parent Φ_A, we apply the lexicase selection operator to select another parent Φ_B as the donor parent. Then, as shown in Fig. 3, the repair strategy enumerates all features $\{\phi_i^A \in \Phi_A | i \in [1, m]\}$ in individual A. For each irrelevant feature ϕ_i^A, the repair operator checks whether the importance value of the corresponding feature ϕ_i^B in individual B is larger than that of ϕ_i^A. If so, the repair operator replaces ϕ_i^A with ϕ_i^B. Similar to the repair operator for irrelevant features, the repair operator for redundant features replaces every redundant feature ϕ_i^A, but this replacement is performed directly without checking feature importance. By combining these two repairing strategies, the number of irrelevant and redundant features could be reduced, thereby improving search effectiveness.

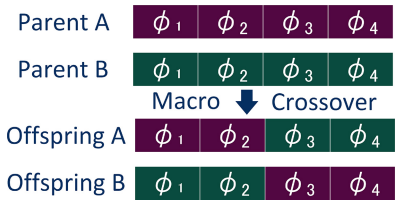


Fig. 4. Illustrative example of macro crossover.

D. Macro Tree Crossover and Adaptive Variation Rate

In MMTGP, random subtree crossover and random subtree mutation [19] operators are used to generate offspring. In this section, we propose a macro crossover operator and an adaptive variation probability control strategy to improve search effectiveness.

1) Macro Tree Crossover: In MMTGP, the crossover operator is performed at the GP tree level, where the random crossover operator swaps a randomly chosen subtree ψ_a from ϕ_k^A in the first parent Φ_A and a randomly chosen subtree ψ_b from ϕ_k^B in the second parent Φ_B. However, in MMTGP, some features $\{\phi_{k1}^A, \phi_{k2}^A\}$ may have dependencies that perform well in Φ_A. For example, ϕ_{k2}^A may take the outputs of ϕ_{k1}^A as inputs. Independently performing crossover may break the dependency between different GP trees. To address this issue, we propose a macro tree crossover (MTC) operator in this article to exchange genetic materials on the individual level rather than the tree level. Specifically, considering m GP trees in each individual, the MTC operator randomly selects two indices $i \in [1, m], j \in [i, m]$. Then, the MTC operator exchanges two sets of GP trees $\{\phi_i^A, \ldots, \phi_j^A\}$ and $\{\phi_i^B, \ldots, \phi_j^B\}$ in two individuals Φ_A and Φ_B, as shown in Fig. 4. By exchanging multiple entire trees in a single crossover operator, the MTC operator preserves the coexistence relationship of features and may be able to generate high-quality feature sets more effectively. It is worth noting that the probability of using a macro crossover operator is controlled by a parameter CR_{MTC}. Once the macro crossover operator has been invoked, the random subtree crossover operator and the guided mutation operator will not be applied as the macro crossover has already made a large change to the parent individuals.

2) Adaptive Variation Rate: In GP, crossover and mutation rates for random crossover and random mutation operators are usually set in advance. However, in MMTGP, GP trees at different positions $i \in [1, m]$ exhibit distinct evolutionary dynamics. For simplicity, we refer to GP trees with lower indices within an individual as low-level GP trees, as they serve as fundamental building blocks. In contrast, GP trees with higher indices within an individual are referred to as high-level GP trees, as they use low-level GP trees to construct more expressive features. The low-level GP trees are easier to converge upon than high-level GP trees, as the input features of low-level GP trees in MMTGP are stable, whereas the input features of high-level GP trees are always changing. Based

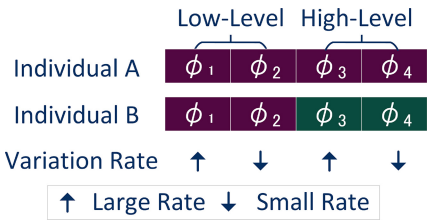


Fig. 5. Illustrative example of adaptive crossover and mutation rate control in early generations.

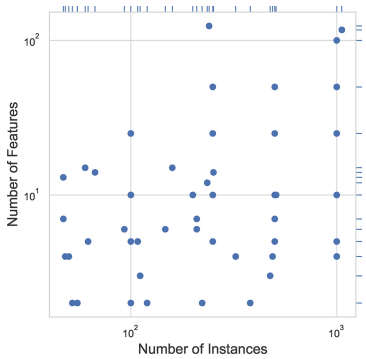


Fig. 6. Properties of the regression benchmark.

on this, it is desirable to have an adaptive crossover/mutation rate control strategy that assigns a higher-variation probability to low-level GP trees to prevent premature convergence in the early stage, while assigning a lower-variation probability to high-level GP trees to avoid excessive fluctuation. Fig. 5 presents an example of such an adaptive rate control strategy. In this example, GP trees at the first two indices should have a higher-crossover/mutation rate, as the phenotypic diversity at these two indices is low, and we need to have a higher-crossover/mutation rate to encourage exploration in the early stage of evolution. To dynamically adjust the crossover/mutation rate, we first need to measure the phenotypic diversity of each index $i \in [1, m]$. In MMTGP, we count the number of GP trees with unique semantics for each index as $c_1, \ldots, c_m$, and use them as an indicator of diversity. The greater the number of GP trees with unique semantics, the higher the diversity. Assuming the maximum crossover rate is cr, and $c_{\min} = \min_{i\in[1,m]} c_i$ represents the minimum number of unique GP trees for all indices, the crossover rate for each index i is defined as shown in

$$cr_i = cr * \frac{c_{\min}}{c_i^2}. \tag{1}$$

Based on (1), it is clear that the index with the smallest diversity corresponds to the largest crossover rate cr, whereas the index with the largest diversity corresponds to the smallest crossover rate. The mutation rate is controlled in the same way as the crossover rate, with the difference being that the maximum mutation rate is limited to mr. However, in the later stage of evolution, low-level GP trees should be encouraged to converge, and computational resources ought to be allocated to explore high-level GP trees. Consequently, the crossover rate is inverted according to (2), where $c_{\max} = \max_{i\in[1,m]} c_i$ represents the maximum number of unique GP trees for all indices

$$cr_i = cr * \frac{c_i^2}{c_{\max}}. \tag{2}$$

Based on (2), the index exhibiting the lowest diversity is encouraged to converge, while the index with the highest diversity continues to explore the search space. Consequently, a portion of the features become fixed, while the remaining features maintain the ability to change until the end of the evolutionary process. The strategy of fixing several GP trees to optimize others is widely used in gradient boosting GP [53] to achieve good performance. However, in MMTGP, the adaptive variation rate strategy allows the remaining tree to optimize at a small rate rather than being completely fixed, providing more flexibility.

IV. Experimental Settings

In this article, a series of experiments have been conducted to evaluate the effectiveness of the proposed MMTGP method for feature construction. This section describes the experimental settings, including the experimental datasets, baseline methods, parameter settings, and evaluation protocol.

A. Datasets

The experiments are conducted on datasets from the Penn machine learning benchmark (PMLB) [54]. To save computational resources, the experiments are conducted on all the 98 datasets in PMLB with less than 2000 instances. The details of the 98 datasets are presented in Fig. 6. The details show that the number of instances ranges from 47 to 1059, whereas the number of dimensions (original features) is between 2 and 124. It is worth noting that in comparative experiments with other SR and machine learning methods on the state-of-the-art symbolic regression benchmark (SRBench) [28], we use all 120 datasets in PMLB to ensure consistency with SRBench.

B. Baseline Methods

MMTGP with a new compact representation aims to reduce model sizes and improve model accuracies. In the GP domain, there are many bloat control techniques for reducing model sizes. Consequently, we compare MMTGP with seven bloat control methods to study the advantages of the proposed compact representation. These seven bloat control methods are selected because they are widely used for comparisons in existing literature and have demonstrated good performance in controlling bloat [17], [50]. For a fair comparison, all bloat control methods are combined with multitree GP methods for feature construction, and a linear model is used to combine features for making predictions. The seven baseline bloat control methods include the following.

1) *Depth Limiting:* Depth limiting is the standard bloat control method in GP [19]. This method adds a strict limit to the depth of each tree for bloat control.
2) *double tournament selection (DTS) [15]:* DTS is a selection-based bloat control method that uses the tournament selection operator to select two parents, and

then selects the smaller one with a higher probability to control bloat.

3) *Tarpeian [14]:* The Tarpeian bloat control method randomly kills some individuals that are larger than the average tree size to avoid generating excessively large trees.
4) *Prune-and-Plant (PAP) [16]:* PAP prunes a GP tree by replacing a subtree with a random node and plants the pruned subtree in the population as a new GP tree to preserve genetic materials.
5) *α-MOEA-based GP (MOGP) [17]:* α-MOGP is an multiobjective evolutionary algorithm (MOEA)-based bloat control method that balances the fitness and complexity using the α-dominance relationship.
6) *TS-S [55]:* The statistics tournament selection method (TS-S) controls bloat by explicitly keeping the smaller solution if the semantics of two individuals are not significantly different.
7) *Dynamic Semantic Approximation (DSA) [50]:* DSA is a variation-based bloat control method that randomly replaces a subtree with randomly generated smaller trees. DSA is named dynamic because it dynamically chooses trees larger than the average tree size to prune.

C. Evaluation Protocol

In this article, we follow the traditional experimental setup in the GP field. All experiments are independently run 30 times to ensure reliable results. In each run, each dataset is randomly split into training data and test data with a ratio of 80:20 [1]. In order to avoid the magnitude difference between different dimensions in each dataset, all data are standardized at the beginning of the training process. After the training process, to eliminate the magnitude difference between different datasets, test R^2 scores are reported, which is defined as $1-([\sum_i(y_i-\hat{y}_i)^2]/[\sum_i(y_i-\bar{y})^2])$, where y_i represents the prediction on test instance i, and $\bar{y}$ stands for the average of target values. The optimal value for the R^2 score is 1, which represents a perfect fit of the model to the data. For $0 < R^2 < 1$, the model can identify and capture some of the underlying patterns in the data, with a higher R^2 indicating a better fit. However, if the predictions provided by the model are worse than those made by using the average output value, the R^2 score will be less than zero, indicating poor prediction results. After 30 runs of experiments, a Wilcoxon signed rank test with a significance level of 0.05 is used to compare the performance of MMTGP with benchmark methods. In addition to the significance test, Friedman's rank is also presented to show the relative rank of different algorithms.

D. Parameter Settings

Most parameters in the GP methods use common settings in the existing literature, as shown in Table I. In order to show the high-search effectiveness of the proposed representation, we use a smaller evolution budget than existing common settings in the current GP literature. Specifically, the number of generations is set to 100 and the population size is set to 30 times the number of original features D, with an upper bound of 300 [56]. A relatively large initial crossover rate is used, which is 0.9. In comparison, the initial mutation rate is set to a relatively low value, which is 0.1. In order to avoid the zero division error, we use the analytical quotient (AQ) [57] instead of the division operator. For the parameters of other bloat control methods, we use the suggested values in their original papers as shown in Table II. It is worth noting that the height limit of standard GP in baseline bloat control methods is set as 10 to be able to construct expressive features without modularization. For the parameters of machine learning and SR methods in SRBench, they are tuned using the successive-halving grid search method according to the parameter search space defined in [28].

TABLE I
PARAMETER SETTINGS FOR MMTGP

Parameter	Value
Maximal Population Size	30D (300)
Number of Generations	100
Crossover and Mutation Rates	0.9 and 0.1
Maximum Tree Depth	2
Initial Tree Depth	1-2
Number of Trees in An Individual	20
Scope (Number of Trees)	10
Macro Crossover Rate	0.25
Elitism (Number of Individuals)	1
Functions	+, -, *, AQ, Sin, Cos, Abs, Max, Min, Negative

TABLE II
PARAMETER SETTINGS FOR BLOAT CONTROL IN DIFFERENT ALGORITHMS

Algorithm	Parameter Settings
DTS [15]	Tournament size = 7, Parsimony size = 1.4
Tarpeian [15]	Reduced fraction = 0.3
PAP [16]	Pruning Probability = 0.5
αMOGP [17]	Initial alpha value = 0, Step size = 90
TS-S [55]	Tournament size = 7

As mentioned in Section III, MMTGP does not use the macro parent repair strategy, the macro crossover operator, the adaptive parameter control strategy and the guided mutation operator when compared to other bloat control methods. This ensures a direct comparison between standard GP with modular GP and tree GP on predictive performance and tree sizes. In experiments on the state-of-the-art SRBench, we conduct experiments with MMTGP, as well as MMTGP with the four proposed strategies, which is named $\text{MMTGP}_{\text{ALL}}$, for achieving state-of-the-art performance in the benchmark.

V. EXPERIMENTAL RESULTS

A. Experimental Results on R^2 Scores

The experimental results are presented in Table III. The experimental results show that MMTGP is significantly better than multitree GP with bloat control techniques. Specifically, compared to the second-ranked bloat control method, αMOGP, MMTGP outperforms it on 46 datasets and performs similarly on 48 datasets. As for other bloat control methods, MMTGP outperforms them on 52 datasets and is inferior to them on up to 14 datasets. It is worth noting that MMTGP is significantly

TABLE III
STATISTICAL COMPARISON ON *Test* R^2 *Scores* FOR DIFFERENT BLOAT CONTROL METHODS. ("+," "∼," AND "-" INDICATE THAT USING THE METHOD IN A ROW PERFORMS BETTER THAN, SIMILAR TO, OR WORSE THAN USING THE METHOD IN A COLUMN)

	αMOGP	Tarpeian	DTS	PAP	TS-S	DSA	DepthLimiting
MMTGP	46(+)/48(∼)/4(-)	52(+)/43(∼)/3(-)	54(+)/42(∼)/2(-)	62(+)/30(∼)/6(-)	53(+)/31(∼)/14(-)	55(+)/41(∼)/2(-)	54(+)/41(∼)/3(-)
αMOGP	—	12(+)/80(∼)/6(-)	24(+)/73(∼)/1(-)	58(+)/28(∼)/12(-)	27(+)/57(∼)/14(-)	10(+)/83(∼)/5(-)	11(+)/81(∼)/6(-)
Tarpeian	—	—	16(+)/79(∼)/3(-)	54(+)/39(∼)/5(-)	16(+)/70(∼)/12(-)	3(+)/83(∼)/12(-)	4(+)/83(∼)/11(-)
DTS	—	—	—	54(+)/33(∼)/11(-)	10(+)/70(∼)/18(-)	8(+)/67(∼)/23(-)	5(+)/64(∼)/29(-)
PAP	—	—	—	—	3(+)/45(∼)/50(-)	8(+)/36(∼)/54(-)	11(+)/34(∼)/53(-)
TS-S	—	—	—	—	—	14(+)/58(∼)/26(-)	15(+)/55(∼)/28(-)
DSA	—	—	—	—	—	—	4(+)/93(∼)/1(-)

TABLE IV
STATISTICAL COMPARISON ON *Tree Sizes* FOR DIFFERENT BLOAT CONTROL METHODS. ("+," "∼," AND "-" INDICATE THAT USING THE METHOD IN A ROW PERFORMS BETTER THAN, SIMILAR TO, OR WORSE THAN USING THE METHOD IN A COLUMN)

	αMOGP	Tarpeian	DTS	PAP	TS-S	DSA	DepthLimiting
MMTGP	98(+)/0(∼)/0(-)	98(+)/0(∼)/0(-)	97(+)/1(∼)/0(-)	43(+)/31(∼)/24(-)	65(+)/10(∼)/23(-)	98(+)/0(∼)/0(-)	98(+)/0(∼)/0(-)
αMOGP	—	1(+)/63(∼)/34(-)	2(+)/36(∼)/60(-)	0(+)/1(∼)/97(-)	0(+)/24(∼)/74(-)	75(+)/23(∼)/0(-)	93(+)/5(∼)/0(-)
Tarpeian	—	—	8(+)/65(∼)/25(-)	0(+)/2(∼)/96(-)	0(+)/37(∼)/61(-)	90(+)/8(∼)/0(-)	98(+)/0(∼)/0(-)
DTS	—	—	—	0(+)/4(∼)/94(-)	13(+)/26(∼)/59(-)	87(+)/10(∼)/1(-)	95(+)/3(∼)/0(-)
PAP	—	—	—	—	55(+)/18(∼)/25(-)	98(+)/0(∼)/0(-)	98(+)/0(∼)/0(-)
TS-S	—	—	—	—	—	96(+)/2(∼)/0(-)	98(+)/0(∼)/0(-)
DSA	—	—	—	—	—	—	44(+)/54(∼)/0(-)

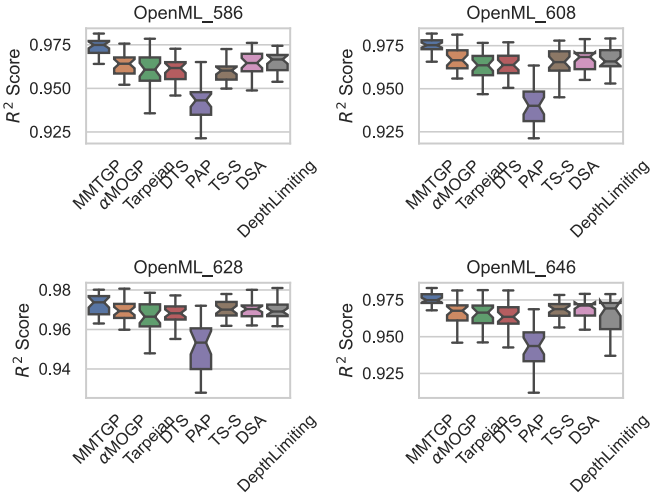


Fig. 7. Distribution of test R^2 scores for different bloat control methods on four representative datasets.

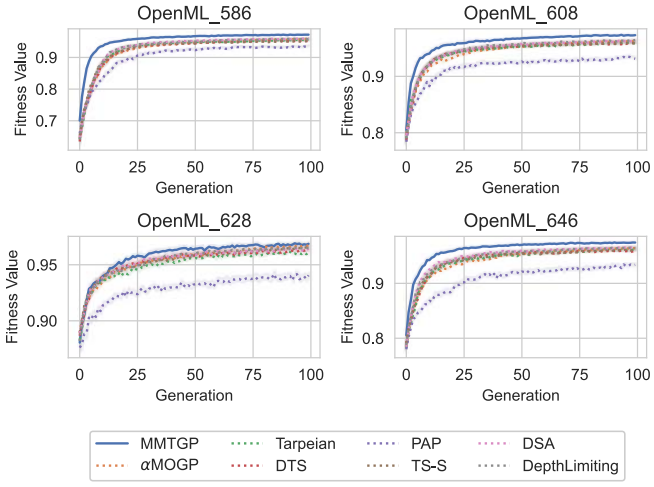


Fig. 8. Evolutionary plots of the best-fitness values for different bloat control methods and 95% confidence interval.

better than the standard GP on 54 datasets and only worse on three datasets. In comparison, other bloat control methods can only have slightly better-test performance than standard GP with the depth limiting method.

Fig. 7 further presents the distribution of R^2 scores over the 30 independent runs on four representative datasets randomly chosen from all experimental datasets. Fig. 7 shows that MMTGP not only has better-average performance than standard GP but is also more stable in achieving good performance, which is particularly noticeable in the "OpenML_646" dataset, reflected by a much smaller standard deviation. In order to gain a better understanding of how different bloat control methods affect search effectiveness, we present the convergence curves for the best-fitness values in Fig. 8. From these figures, we can see that although all methods start from the same initial point, MMTGP has a significantly faster convergence rate than other GP with bloat control methods in the early stages. These results indicate that simply removing noisy and less informative subtrees with existing bloat control methods is not enough to improve search effectiveness, and that using the modularization technique to remove duplicate building blocks is a better approach. This can explain why MMTGP outperforms other baseline methods on many datasets.

B. Experimental Results on Tree Sizes

The goal of modular GP is not only to obtain better-R^2 scores but also to obtain smaller GP trees. Specifically, we aim to reduce the average number of nodes of all trees in each individual, which is defined as the average tree size. To validate the decrease in average tree sizes, we present a statistical comparison on tree sizes for different bloat control methods in Table IV. The results show that the size of GP trees found by MMTGP is also smaller than that of other

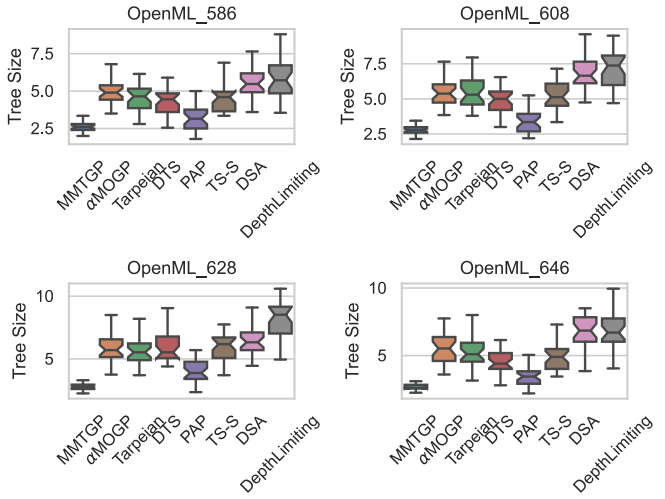


Fig. 9. Distribution of tree sizes for different bloat control methods on four representative datasets.

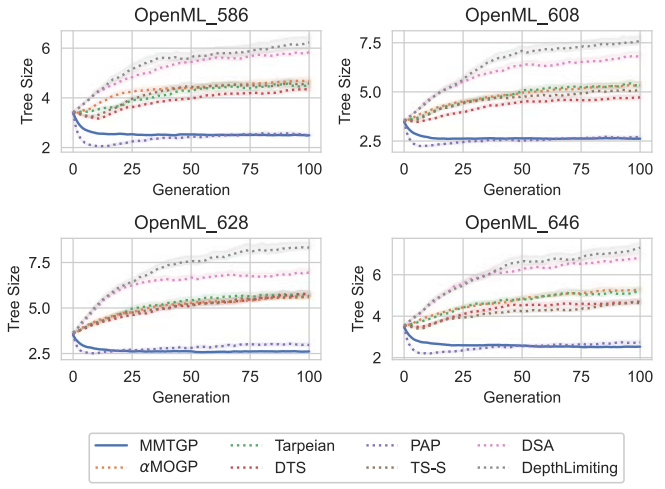


Fig. 10. Evolutionary plots of average tree sizes for different bloat control methods and 95% confidence interval.

bloat control methods. For example, when comparing MMTGP with the second-best method, PAP, it significantly outperforms DSA on 43 out of the 98 datasets and has worse results on 24 datasets. Compared with the third-ranked method, TS-S, MMTGP outperforms it on 65 datasets and is inferior to it on 23 datasets. As for the other methods, MMTGP has a clear advantage, where MMTGP is significantly better than the other methods on at least 97 datasets and no worse on any dataset. These results validate the effectiveness of MMTGP in controlling code bloat.

A more concrete comparison of four representative datasets for the distribution of tree sizes over the 30 runs is presented in Fig. 9. These results further confirm that MMTGP not only has smaller average tree sizes than standard GP with bloat control methods but is also stable across different runs. This is exactly as expected since MMTGP sets the maximal tree depth to 2, and thus the tree size will not be very large, which makes the tree size stable across different runs. To have a deeper understanding of why MMTGP can have a small tree size, we plot the evolutionary plots of tree sizes in Fig. 10. It

TABLE V
FRIEDMAN'S RANK OF R^2 TEST SCORES AND AVERAGE TREE SIZES OF ALL BLOAT CONTROL METHODS ON ALL DATASETS. (THE RELATIVE RANKS ARE PRESENTED IN PARENTHESES.)

Algorithm	R^2 Rank	Size Rank
MMTGP	2.69 (1)	1.77 (1)
αMOGP	3.98 (2)	5.72 (6)
DSA	4.05 (3)	7.02 (7)
TS-S	4.47 (4)	2.94 (3)
DepthLimiting	4.48 (5)	7.84 (8)
Tarpeian	4.83 (6)	4.77 (5)
DTS	5.35 (7)	4.09 (4)
PAP	6.15 (8)	1.86 (2)

is evident that, except for PAP and MMTGP, all bloat control methods show an increase in tree size as the evolutionary progress continues. This is not necessarily a problem, as a modest increase in tree size to search for more complex trees is reasonable. However, Fig. 10 shows that the average tree size of MMTGP decreases as the evolution progresses. This could be attributed to the modular representation employed by MMTGP, which allows small GP trees to effectively represent complex programs. When resources are limited, smaller GP trees, corresponding to a smaller search space, tend to be more effective in finding good solutions compared to larger GP trees. Thus, smaller trees are more likely to survive, leading to a decrease in average tree size. Given that MMTGP is significantly better than other bloat control methods in predictive performance, we can conclude that increasing tree sizes to improve predictive performance is unnecessary as long as the modular technique is used.

Table V summarizes Friedman's rank of R^2 scores and the average tree sizes. The results show that MMTGP has both the best result in test R^2 scores and model sizes. In comparison, the second-ranked and third-ranked methods in terms of R^2 scores, αMOGP and DSA, have significantly larger model size than MMTGP. As for the second-ranked and third-ranked methods with respect to model size, PAP, and TS-S, they only rank eighth and fourth in R^2 scores. These results show that MMTGP achieves the best tradeoff between predictive performance and tree sizes.

C. Comparison With Symbolic Regression Methods

In this section, we present experimental results on SRBench, which includes the proposed $\text{MMTGP}_{\text{ALL}}$ and MMTGP methods, and the 22 benchmark algorithms with 120 datasets. Fig. 11 presents the normalized test R^2 scores, model size, and training time distribution of all algorithms. Test R^2 scores are normalized by the maximum and minimum score on each dataset across all algorithms. This normalization eliminates the magnitude bias where it is easy to get an R^2 of 0.99 on simple regression datasets, whereas it is hard to get an R^2 of 0.8 on difficult regression datasets. The result shows that the proposed MMTGP variant $\text{MMTGP}_{\text{ALL}}$ is the best method among all comparison methods in terms of normalized R^2 scores. Further statistical analysis in Fig. 12 shows that $\text{MMTGP}_{\text{ALL}}$ is significantly better than other SR and machine learning methods except for PS-Tree, which has similar R^2

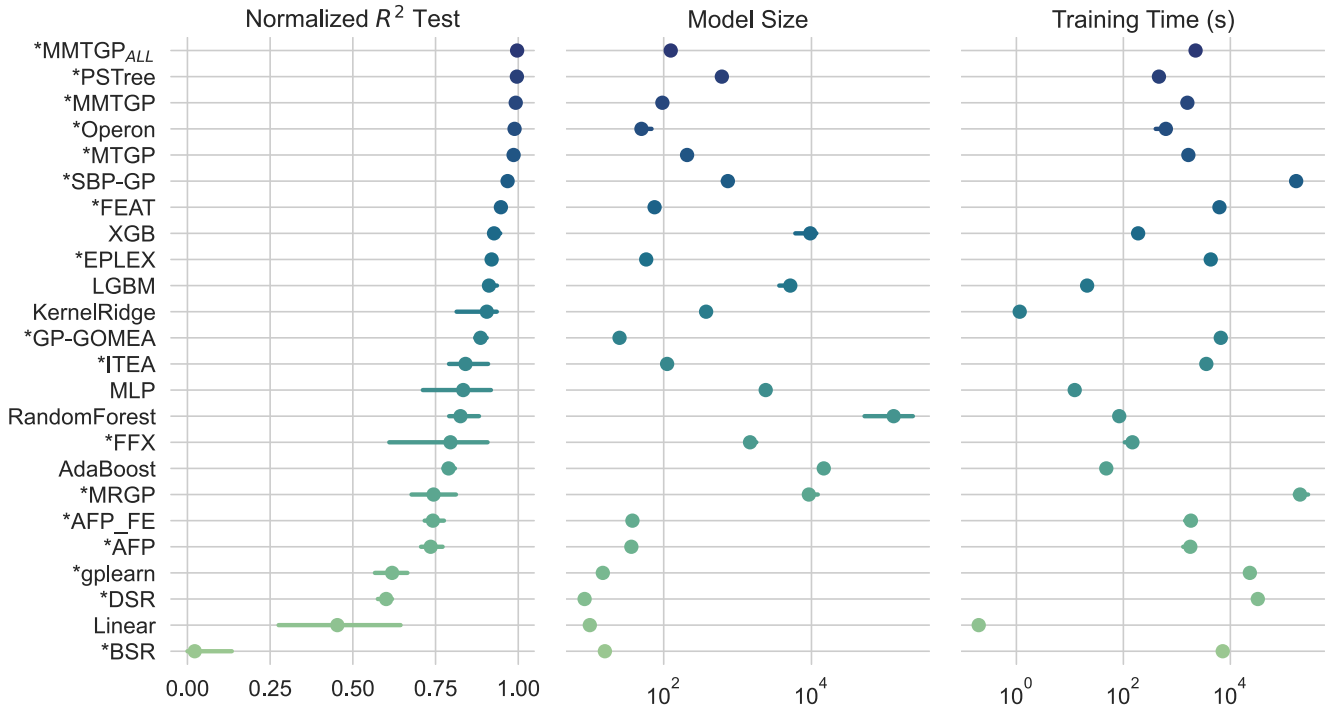


Fig. 11. R^2 scores, model sizes, and training time of 24 algorithms on 120 regression problems. (Asterisk means that the method is a symbolic regression method.)

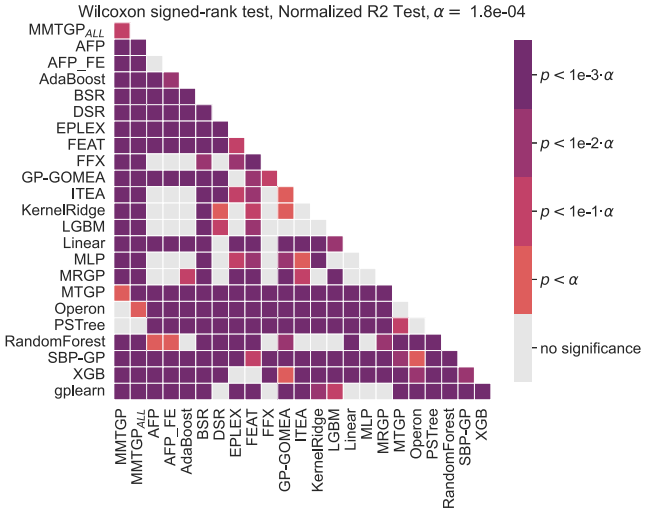


Fig. 12. Pairwise statistical comparison on R^2 scores with Wilcoxon signed-rank tests and a Bonferonni correction.

scores to MMTGP. As for the model size, MMTGP$_{\text{ALL}}$ has a significantly smaller size than PS-Tree and has a comparable size to FEAT. It is worth noting that we follow the guideline of SR-Bench to count the number of nodes in a SymPy-compatible format. For example, the AQ operator $(a/\sqrt{1+b^2})$ is counted as 10 nodes because SymPy does not support the division operator, and it needs to be transformed with the power operation as $a(\sqrt{1+b^2})^{-1}$ before counting the number of nodes. As for baseline methods, some methods, e.g., GP-GOMEA, in SRBench only count the number of nodes in GP trees rather than transforming GP trees into a SymPy-compatible format, which may cause MMTGP$_{\text{ALL}}$ to have a slightly larger value in model sizes. Finally, with respect to the training time, MMTGP$_{\text{ALL}}$ spends more time than PS-Tree and Operon but has a shorter training time than SBP-GP and FEAT. Considering that MMTGP$_{\text{ALL}}$ has better-test performance than these benchmark methods, the increase in training time is acceptable. It is worth noting that although MMTGP is worse than MMTGP$_{\text{ALL}}$, it still has a good rank and significantly outperforms MTGP, indicating that modular representation is the key component for achieving state-of-the-art performance.

VI. Further Analysis

In this section, we first conduct further analysis of the layer constraint and scope constraint in MMTGP. Then, we discuss the effectiveness of the macro parent repair strategy, the macro crossover operator, and the adaptive parameter control strategy.

A. Structural Constraints

In MMTGP, we add a layer constraint and a scope constraint to encourage component reuse and avoid an overly large search space. Additionally, we employ a sliding-window-style connection instead of a layer-to-layer-style connection, as it allows for a more flexible connection. In this section, we investigate the impact of these three constraints. Specifically, we test three variants of MMTGP.

1) *MMTGP Without the Layer Constraint (W/O Layer):* This variant removes the strict component reuse constraint and allows high-level GP trees to directly access original features.
2) *MMTGP Without the Scope Constraint (W/O Scope):* This variant allows high-level GP trees to access all

TABLE VI
STATISTICAL COMPARISON ON *Test* R^2 *Scores* WITH DIFFERENT CONSTRAINTS

	W/O Layer	W/O Scope	W/O Sliding
MMTGP	33(+)/63(∼)/2(-)	29(+)/65(∼)/4(-)	60(+)/32(∼)/6(-)
W/O Layer	—	2(+)/91(∼)/5(-)	45(+)/50(∼)/3(-)
W/O Scope	—	—	45(+)/51(∼)/2(-)

TABLE VII
STATISTICAL COMPARISON ON *Test* R^2 *Scores* WITH MACRO PARENT REPAIR

	Redundant Repair	Irrelevant Repair
No Repair	1(+)/68(∼)/29(-)	1(+)/70(∼)/27(-)
Redundant Repair	—	5(+)/89(∼)/4(-)

outputs of previous GP trees instead of having a scope of 10 GP trees.

3) *MMTGP Without Sliding Window Style Connection (W/O Sliding):* This variant divides 20 GP trees into four layers: a) $\{\Phi_1, \ldots, \Phi_5\}$; b) $\{\Phi_6, \ldots, \Phi_{10}\}$; c) $\{\Phi_{11}, \ldots, \Phi_{15}\}$; and d) $\{\Phi_{16}, \ldots, \Phi_{20}\}$, with each layer consisting of five GP trees. Without the sliding window connection, all GP individuals in the same layer share and use the same set of outputs from GP trees in the previous layer. MMTGP without sliding windows style connection is similar to a deep neural network (DNN), with the difference being that the activation function at each node is a GP tree rather than a sigmoid or ReLU function.

Experimental results are presented in Table VI. The results show that the layer constraint improves the test performance of MMTGP on 33 out of the 98 datasets, whereas it only worsens the test R^2 score on two datasets. As for the scope constraint, it significantly improves the performance on 29 datasets and only results in worse performance on four datasets. In terms of the sliding-window-style connection, it enhances the test performance of MMTGP on 60 datasets and only degrades the test performance on six datasets. Based on these experimental results, we can conclude that the scope constraint and layer constraint are necessary for MMTGP. Moreover, for the layer constraint, using a sliding-window-style connection is more advantageous than the layer-style connection.

B. Effectiveness of Macro Parent Repair

In $\text{MMTGP}_{\text{ALL}}$, we propose a macro parent repair strategy to repair redundant and irrelevant GP trees in each individual. This section investigates whether these repair strategies can enhance the test performance of MMTGP. The experimental results are presented in Table VII. The results show that repairing redundant features can improve the test performance on 29 out of the 98 datasets, whereas repairing irrelevant features can improve the test performance on 27 out of the 98 datasets. Compared to the performance improvements, the performance degradation is negligible, as the two repair strategies only degrade the test performance on 1 dataset. To gain a deeper understanding of the repair strategies, we plot the change in the number of redundant features and irrelevant features during the evolution process in Figs. 13 and 14, respectively. Fig. 13 shows that the redundant feature repair strategy can significantly reduce the number of redundant features, and Fig. 14 validates that the irrelevant feature repair strategy can significantly reduce the number of irrelevant features. Based on these results, we can conclude that MMTGP may generate many redundant and irrelevant features during the evolution process, and the macro parent repair strategy improves the search effectiveness of MMTGP.

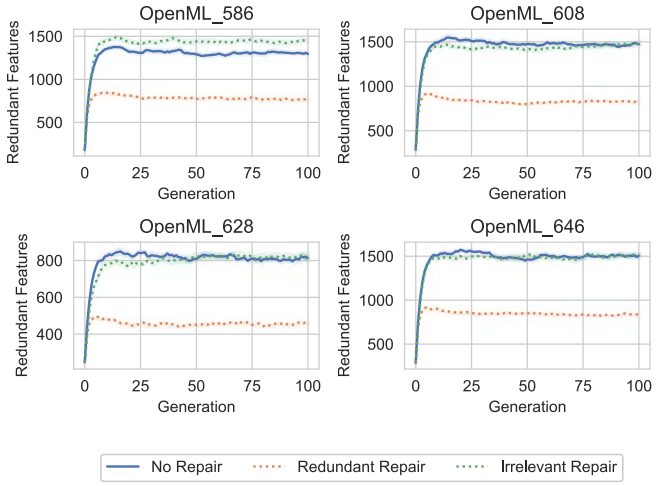


Fig. 13. Change for *the number of redundant features* with the use of repair strategies during the evolutionary process.

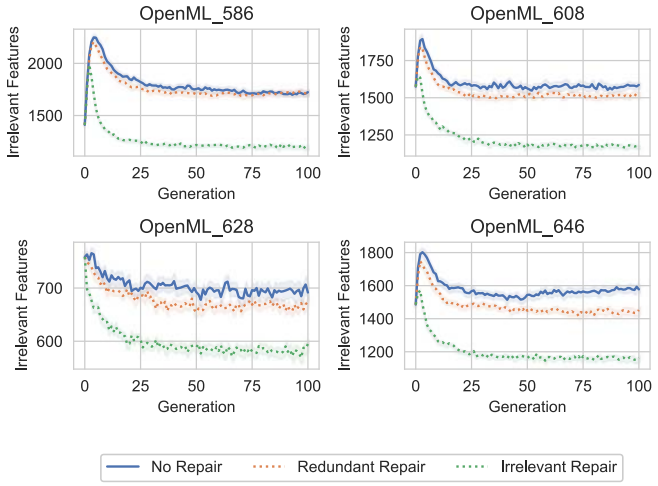


Fig. 14. Change for *the number of irrelevant features* with the use of repair strategies during the evolutionary process.

C. Effectiveness of Macro Crossover

In this article, we propose a macro crossover operator to allow $\text{MMTGP}_{\text{ALL}}$ to exchange interactive GP features. To validate the effectiveness of the proposed macro crossover operator, as well as to determine the optimal value of macro crossover probability, we conduct experiments in this section to investigate four macro crossover probabilities: $\{0, 0.1, 0.25, 0.5\}$. Table VIII presents the experimental results. It is clear that using macro crossover is beneficial, and using the macro crossover with a probability of 0.25 is the

TABLE VIII
STATISTICAL COMPARISON ON *Test* R^2 *Scores* WITH DIFFERENT MACRO CROSSOVER PROBABILITIES

	0.1	**0.25**	**0.5**
0	0(+)/80(∼)/18(-)	3(+)/57(∼)/38(-)	7(+)/51(∼)/40(-)
0.1	—	2(+)/87(∼)/9(-)	10(+)/68(∼)/20(-)
0.25	—	—	5(+)/89(∼)/4(-)

TABLE IX
STATISTICAL COMPARISON ON *Fitness Values* WITH DIFFERENT MACRO CROSSOVER PROBABILITIES

	0.1	**0.25**	**0.5**
0	1(+)/48(∼)/49(-)	0(+)/20(∼)/78(-)	0(+)/8(∼)/90(-)
0.1	—	2(+)/44(∼)/52(-)	1(+)/19(∼)/78(-)
0.25	—	—	1(+)/61(∼)/36(-)

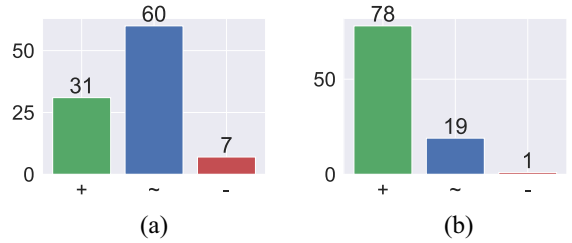


Fig. 15. Statistical comparison results for using adaptive parameter control strategies or not. (a) R2 scores. (b) Fitness values.

optimal choice, as it can improve the test performance on 38 datasets compared to not using the macro crossover operator, while only degrading the test performance on three datasets.

More significant results can be seen from the best-fitness values. As shown in Table IX, using the macro crossover probability of 0.25 can lead to significantly better-fitness values on 78 datasets, not degrade performance on any datasets. If further increasing the crossover probability to 0.5, it can further improve fitness value on 12 datasets. These results indicate that considering feature interaction in the crossover operator is beneficial to improve the performance of MMTGP. In practice, it is recommended to set the macro crossover probability to 0.25, since Table VIII shows that further increasing the probability will not significantly improve the predictive performance.

D. Effectiveness of Adaptive Crossover and Mutation Rates

In this article, we propose an adaptive crossover rate and mutation rate control strategy to dynamically determine the crossover rate and mutation rate during the evolution process. Here, we first present the statistical significance comparison results on the test R^2 scores in Fig. 15(a). The results show that the adaptive parameter control strategy can significantly improve test performance on 31 datasets and degrade performance on seven datasets, showing the effectiveness of the adaptive parameter control strategy. The reason why the control strategy can improve predictive performance is that models with better-generalization capability can be found by balancing the tradeoff between exploration and exploitation during the evolutionary process. Fig. 15(b) shows the gain in fitness values. The experimental results show that the

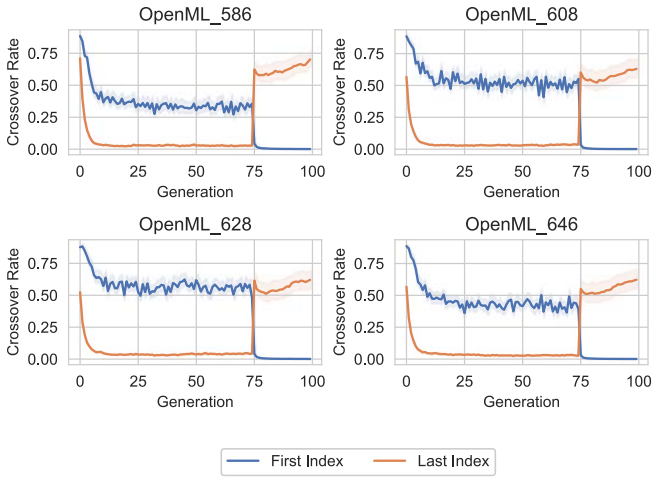


Fig. 16. Crossover rate for the first index and the last index when using the adaptive crossover rate control strategy.

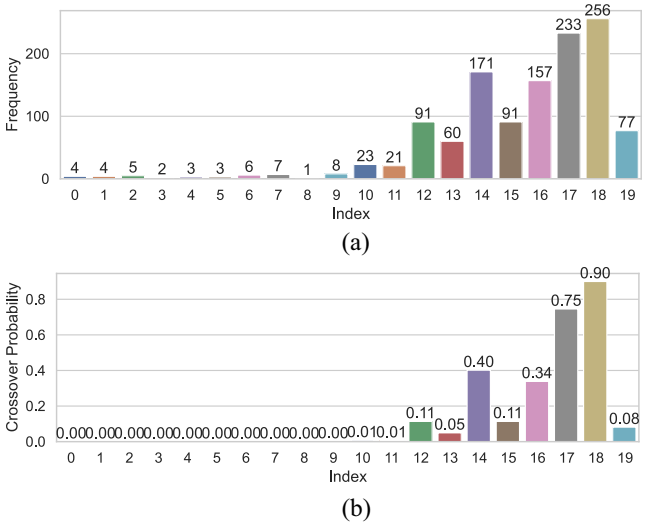


Fig. 17. Example for the number of unique trees out of 300 trees and the corresponding adaptive probabilities on the "OpenML_605" dataset at the end of evolution. (a) Number of unique trees. (b) Adaptive crossover probabilities.

adaptive rate control strategy can significantly improve the best-fitness value on 78 datasets and only degrade performance on one dataset. This verifies that adaptive parameter control is an effective strategy to improve predictive performance by enhancing search effectiveness. Moreover, to gain a deeper understanding of the change in variation rate during the evolution process, we present the change in the crossover rate in the first index and last index in Fig. 16. The results verify that the adaptive parameter control strategy makes the first index have a large variation rate to encourage exploration in early generations. In comparison, the adaptive parameter control strategy makes the last index have a relatively smaller variation rate in early generations as it has large diversity. The crossover rate significantly changes after 75% generations, as low-level GP trees need to converge in later generations, and high-level GP trees can fully be explored in this stage.

A more concrete example is presented in Fig. 17, where Fig. 17(a) represents the number of unique trees at each index

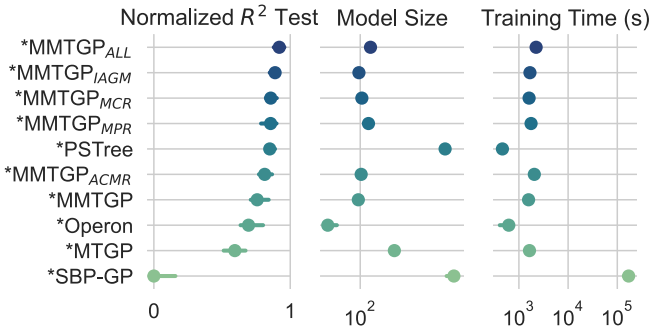


Fig. 18. R^2 scores, model sizes, and training time of integrating different strategies on 120 regression problems.

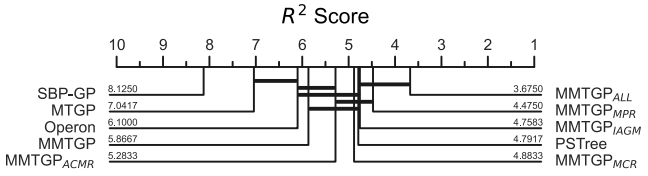


Fig. 19. Critical diagram of integrating different strategies in MMTGP with Wilcoxon signed-rank tests and Holm method.

at the end of evolution, and Fig. 17(b) shows the corresponding adaptive crossover probabilities based on the number of unique trees. The first ten indices have fewer unique trees as these trees can use original variables and exhibit stable behaviors. The adaptive variation rate control strategy encourages their convergence in the later stages of evolution, further resulting in only a few unique trees surviving in the population. In contrast, the last ten indices have a larger number of unique trees because these trees can only use low-level modules, and their behavior is more unstable. Coupled with the adaptive variation rate control strategy assigns them relatively large variation rates in the later stages, leading to greater diversity at the end stage of evolution. Based on this design, the evolutionary algorithm can focus on exploiting useful features while still having enough probability to explore new features in the later stages. This explains why the adaptive parameter control strategy is effective for MMTGP.

E. Overall Analysis

To have an intuitive view of the effectiveness of the proposed strategies, we develop four variants of MMTGP.

1) *MMTGP$_{MPR}$:* MMTGP with macro parent repairing, which is proposed in Section III-C to replace irrelevant and redundant features with more promising features.
2) *MMTGP$_{MCR}$:* MMTGP with macro crossover operator, which is proposed in Section III-D1 to preserve interdependency relationships between features.
3) *MMTGP$_{ACMR}$:* MMTGP with adaptive crossover and mutation rate strategies, which is proposed in Section III-D2 to adaptively balance exploration and exploitation for each tree.
4) *MMTGP$_{IAGM}$:* MMTGP with index-aware guided mutation, the details of which are presented proposed in Section I of the supplementary materials to increase the usage of effective original features.

Fig. 18 shows the normalized test R^2 scores, model sizes, and training time of MMTGP with different strategies. Generally, experimental results show that all the proposed strategies are helpful in improving the overall R^2 test scores, and combining all these strategies can achieve the best performance. Furthermore, Fig. 19 shows that MMTGP$_{\text{ALL}}$ is significantly better than MMTGP and has a better-mean rank than other MMTGP variants, although it does not significantly outperform MMTGP$_{\text{MPR}}$ and MMTGP$_{\text{IAGM}}$. Overall, these results indicate that simply using MMTGP has good performance, and including some additional strategies, provides further improvements for MMTGP.

F. Example Models

In order to further understand the constructed features, we present an example of generated features in Fig. 20 based on a galaxy visualization dataset [58]. This model achieves an R^2 test score of 0.973, whereas XGBoost only achieves an R^2 test score of 0.886, showing that the modular GP system can achieve high-predictive accuracy without impairing interpretability. Based on the coefficients of the constructed features, it is clear that the high-level GP trees have large coefficients in the final model, indicating that MMTGP can effectively combine low-level GP trees to construct expressive features. Moreover, Fig. 20 shows that $\{\phi_0, \phi_1, \phi_3\}$ are important features in MMTGP. This is consistent with the important features found by FEAT [8], and thus a domain-knowledge-based analysis for the features found by MMTGP is worthwhile to investigate in the future.

VII. Conclusion

This article aimed to develop a novel GP method to automatically construct compact features that enhance regression performance. The goal has been successfully achieved by proposing a MMTGP representation with three connection constraints. Additionally, we proposed a macro parent repairing strategy, a macro crossover operator, and an adaptive crossover and mutation rate control strategy to achieve better-search effectiveness.

The performance of MMTGP was examined on 98 regression datasets and compared with seven bloat control methods. The experimental results indicate that MMTGP can evolve a compact and accurate GP model and significantly outperforms traditional GP with bloat control methods. Based on this, MMTGP achieved competitive test R^2 scores with existing state-of-the-art symbolic regression methods while greatly reducing the model size. Furthermore, incorporating the four proposed strategies makes MMTGP outperform all baseline methods on test R^2 scores.

This article explored the effectiveness of MMTGP on regression problems. In the future, it would be interesting to study whether MMTGP is useful for classification and operational research problems. It is worth noting that this article limits the depth of tree to 2 for obtaining an interpretable model. Investigating modularization with large GP

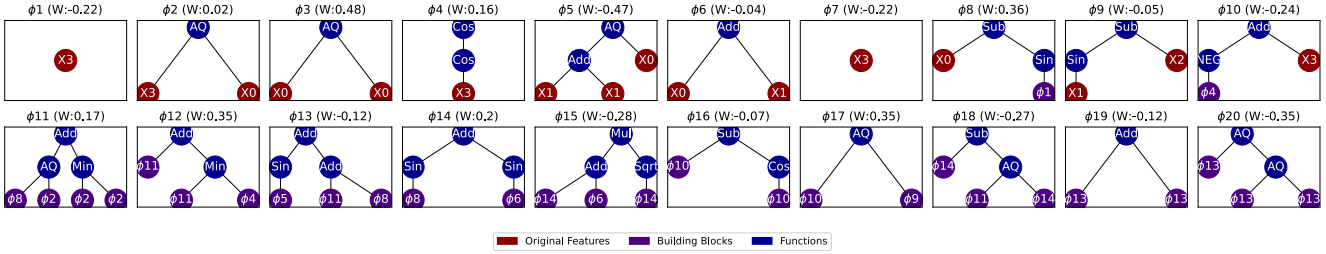


Fig. 20. Example of constructed features based on MMTGP. Red nodes represent original features. Purple nodes represent building blocks, which are the outputs of other low-level GP trees in practice. Blue nodes represent functions.

trees to tackle complex problems like image classification is also an interesting future direction. Also, the current method still needs to determine the number of trees in advance, which could be avoided by developing a genetic operator that can automatically insert or remove GP trees. Further investigations are worthwhile in the future.

REFERENCES

[1] H. Zhang, A. Zhou, and H. Zhang, "An evolutionary forest for regression," *IEEE Trans. Evol. Comput.*, vol. 26, no. 4, pp. 735–749, Aug. 2022.

[2] A. Rahimi and B. Recht, "Random features for large-scale kernel machines," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS 2007)*, vol. 20, 2007, pp. 1–8.

[3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[4] A. Kadra, M. Lindauer, F. Hutter, and J. Grabocka, "Well-tuned simple nets excel on tabular datasets," in *Proc. NeurIPS 2021 Conf.*, vol. 34, 2021, pp. 23928–23941.

[5] T. Hu, "Genetic programming for interpretable and explainable machine learning," in *Genetic Programming Theory and Practice XIX*. Singapore: Springer, 2023, pp. 81–90.

[6] K. Nag and N. R. Pal, "Feature extraction and selection for parsimonious classifiers with multiobjective genetic programming," *IEEE Trans. Evol. Comput.*, vol. 24, no. 3, pp. 454–466, Jun. 2020.

[7] K. Neshatian, M. Zhang, and P. Andreae, "A filter approach to multiple feature construction for symbolic learning classifiers using genetic programming," *IEEE Trans. Evol. Comput.*, vol. 16, no. 5, pp. 645–661, Oct. 2012.

[8] W. La Cava, T. R. Singh, J. Taggart, S. Suri, and J. H. Moore, "Learning concise representations for regression by evolving networks of trees," in *Proc. ICLR*, 2018, pp. 1–16.

[9] W. La Cava and J. Moore, "A general feature engineering wrapper for machine learning using ϵ-lexicase survival," in *Proc. EuroGP*, 2017, pp. 80–95.

[10] M. Muharram and G. D. Smith, "Evolutionary constructive induction," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 11, pp. 1518–1528, Nov. 2005.

[11] J. Ma and X. Gao, "A filter-based feature construction and feature selection approach for classification using genetic programming," *Knowl. Based Syst.*, vol. 196, May 2020, Art. no. 105806.

[12] Q. Chen, M. Zhang, and B. Xue, "Feature selection to improve generalization of genetic programming for high-dimensional symbolic regression," *IEEE Trans. Evol. Comput.*, vol. 21, no. 5, pp. 792–806, Oct. 2017.

[13] H. Zhang, A. Zhou, H. Qian, and H. Zhang, "PS-Tree: A piecewise symbolic regression tree," *Swarm Evol. Comput.*, vol. 71, Jun. 2022, Art. no. 101061.

[14] R. Poli, "A simple but theoretically-motivated method to control bloat in genetic programming," in *Proc. EuroGP*, 2003, pp. 204–217.

[15] S. Luke and L. Panait, "A comparison of bloat control methods for genetic programming," *Evol. Comput.*, vol. 14, no. 3, pp. 309–344, 2006.

[16] E. Alfaro-Cid, J. Merelo, F. F. de Vega, A. I. Esparcia-Alcázar, and K. Sharman, "Bloat control operators and diversity in genetic programming: A comparative study," *Evol. Comput.*, vol. 18, no. 2, pp. 305–332, 2010.

[17] S. Wang, Y. Mei, and M. Zhang, "A multi-objective genetic programming algorithm with α dominance and archive for uncertain capacitated arc routing problem," *IEEE Trans. Evol. Comput.*, early access, Jul. 29, 2022, doi: 10.1109/TEVC.2022.3195165.

[18] J. R. Woodward, "Modularity in genetic programming," in *Proc. EuroGP*, 2003, pp. 254–263.

[19] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," *Stat. Comput.*, vol. 4, no. 2, pp. 87–112, 1994.

[20] S. Kelly, R. J. Smith, M. I. Heywood, and W. Banzhaf, "Emergent tangled program graphs in partially observable recursive forecasting and ViZDoom navigation tasks," *ACM Trans. Evol. Learn. Optim.*, vol. 1, no. 3, pp. 1–41, 2021.

[21] J. F. Miller and S. L. Harding, "Cartesian genetic programming," in *Proc. GECCO Companion*, 2008, pp. 2701–2726.

[22] M. F. Brameier and W. Banzhaf, "Basic concepts of linear genetic programming," in *Linear Genetic Programming*. Boston, MA, USA: Springer, 2007, pp. 13–34.

[23] T. Perkis, "Stack-based genetic programming," in *Proc. 1st IEEE Conf. Evol. Computat. IEEE World Congr. Comput. Intell.*, 1994, pp. 148–153.

[24] J. Zhong, Y.-S. Ong, and W. Cai, "Self-learning gene expression programming," *IEEE Trans. Evol. Comput.*, vol. 20, no. 1, pp. 65–80, Feb. 2016.

[25] M. Suganuma, M. Kobayashi, S. Shirakawa, and T. Nagao, "Evolution of deep convolutional neural networks using cartesian genetic programming," *Evol. Comput.*, vol. 28, no. 1, pp. 141–163, 2020.

[26] L. Françoso Dal Piccol Sotto, P. Kaufmann, T. Atkinson, R. Kalkreuth, and M. Porto Basgalupp, "Graph representations in genetic programming," *Genet. Program. Evol. Mach.*, vol. 22, no. 4, pp. 607–636, 2021.

[27] D. Sobania, D. Schweim, and F. Rothlauf, "A comprehensive survey on program synthesis with evolutionary algorithms," *IEEE Trans. Evol. Comput.*, vol. 27, no. 1, pp. 82–97, Feb. 2023.

[28] W. L. Cava et al., "Contemporary symbolic regression methods and their relative performance," in *Proc. 35th Conf. Neural Inf. Process. Syst. Datasets Benchmarks Track (Round 1)*, 2021, pp. 1–16.

[29] W. La Cava, L. Spector, and K. Danai, "Epsilon-lexicase selection for regression," in *Proc. GECCO*, 2016, pp. 741–748.

[30] W. A. Tackett, "Recombination, selection, and the genetic construction of computer programs," Ph.D. dissertation, Computer Science Dept., Univ. Southern California Los Angeles, Los Angeles, CA, USA, 1994.

[31] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and its Applications*. San Mateo, CA, USA: Morgan Kaufmann, 1998.

[32] T. Soule and J. A. Foster, "Removal bias: A new cause of code growth in tree based evolutionary programming," in *Proc. IEEE Int. Conf. Evol. Comput. Proc*, 1998, pp. 781–786.

[33] W. B. Langdon and R. Poli, "Fitness causes bloat," in *Proc. Soft Comput. Eng. Design Manuf.*, 1998, pp. 13–22.

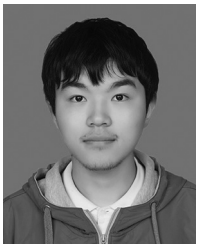
[34] S. Silva and E. Costa, "Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories," *Genet. Program. Evol. Mach.*, vol. 10, no. 2, pp. 141–179, 2009.

[35] Y. Mei, Q. Chen, A. Lensen, B. Xue, and M. Zhang, "Explainable artificial intelligence by genetic programming: A survey," *IEEE Trans. Evol. Comput.*, vol. 27, no. 3, pp. 621–641, Jun. 2023.

[36] B.-T. Zhang and H. Mühlenbein, "Balancing accuracy and parsimony in genetic programming," *Evol. Comput.*, vol. 3, no. 1, pp. 17–38, 1995.

[37] S. Dignum and R. Poli, "Operator equalisation and bloat free GP," in *Proc. EuroGP*, 2008, pp. 110–121.

[38] R. Poli and N. F. McPhee, "Parsimony pressure made easy," in *Proc. GECCO*, 2008, pp. 1267–1274.
[39] S. Silva, S. Dignum, and L. Vanneschi, "Operator equalisation for bloat free genetic programming and a survey of bloat control methods," *Genet. Program. Evol. Mach.*, vol. 13, no. 2, pp. 197–238, 2012.
[40] E. D. De Jong and J. B. Pollack, "Multi-objective methods for tree size control," *Genet. Program. Evol. Mach.*, vol. 4, no. 3, pp. 211–233, 2003.
[41] M. Kommenda, G. Kronberger, M. Affenzeller, S. M. Winkler, and B. Burlacu, "Evolving simple symbolic regression models by multi-objective genetic programming," in *Genetic Programming Theory and Practice XIII*. Cham, Switzerland: Springer, 2016, pp. 1–19.
[42] D. Liu, M. Virgolin, T. Alderliesten, and P. A. Bosman, "Evolvability degeneration in multi-objective genetic programming for symbolic regression," 2022, *arXiv:2202.06983*.
[43] S. Luke and L. Panait, "Lexicographic parsimony pressure," in *Proc. GECCO*, 2002, pp. 829–836.
[44] W. B. Langdon, "Size fair and homologous tree genetic programming crossovers," *Genet. Program. Evol. Mach.*, vol. 1, no. 1/2, pp. 95–119, 2000.
[45] E. Alfaro-Cid, A. Esparcia-Alcázar, K. Sharman, and F. F. de Vega, "Prune and plant: A new bloat control method for genetic programming," in *Proc. HIS*, 2008, pp. 31–35.
[46] N. Javed, F. Gobet, and P. Lane, "Simplification of genetic programs: A literature survey," *Data Min. Knowl. Discov.*, vol. 36, pp. 1279–1300, Apr. 2022.
[47] P. Wong and M. Zhang, "Algebraic simplification of GP programs during evolution," in *Proc. GECCO*, 2006, pp. 927–934.
[48] A. Song, D. Chen, and M. Zhang, "Contribution based bloat control in genetic programming," in *Proc. IEEE Congr. Evol. Comput.*, 2010, pp. 1–8.
[49] D. Kinzett, M. Johnston, and M. Zhang, "Numerical simplification for bloat control and analysis of building blocks in genetic programming," *Evol. Intell.*, vol. 2, no. 4, pp. 151–168, 2009.
[50] Q. U. Nguyen and T. H. Chu, "Semantic approximation for reducing code bloat in genetic programming," *Swarm Evol. Comput.*, vol. 58, Nov. 2020, Art. no. 100729.
[51] H. Zhang, A. Zhou, Q. Chen, B. Xue, and M. Zhang, "SR-forest: A genetic programming based heterogeneous ensemble learning method," *IEEE Trans. Evol. Comput.*, early access, Feb. 7, 2023, doi: 10.1109/TEVC.2023.3243172.
[52] W. La Cava, T. Helmuth, L. Spector, and J. H. Moore, "A probabilistic and multi-objective analysis of lexicase selection and ε-lexicase selection," *Evol. Comput.*, vol. 27, no. 3, pp. 377–402, 2019.
[53] D. Medernach, J. Fitzgerald, R. M. A. Azad, and C. Ryan, "A new wave: A dynamic approach to genetic programming," in *Proc. GECCO*, 2016, pp. 757–764.
[54] R. S. Olson, W. La Cava, P. Orzechowski, R. J. Urbanowicz, and J. H. Moore, "PMLB: A large benchmark suite for machine learning evaluation and comparison," *BioData Min.*, vol. 10, no. 1, pp. 1–13, 2017.
[55] T. H. Chu, Q. U. Nguyen, and M. O'Neill, "Semantic tournament selection for genetic programming based on statistical analysis of error vectors," *Inf. Sci.*, vol. 436, pp. 352–366, Apr. 2018.
[56] P. Wang, B. Xue, J. Liang, and M. Zhang, "Differential evolution based feature selection: A niching-based multi-objective approach," *IEEE Trans. Evol. Comput.*, vol. 27, no. 2, pp. 296–310, Apr. 2023.
[57] J. Ni, R. H. Drieberg, and P. I. Rockett, "The use of an analytic quotient operator in genetic programming," *IEEE Trans. Evol. Comput.*, vol. 17, no. 1, pp. 146–152, Feb. 2013.
[58] W. S. Cleveland, *Visualizing Data*. Thousand Oaks, CA, USA: Hobart Press, 1993.



Hengzhe Zhang (Member, IEEE) received the B.Sc. degree in software engineering from Xiangtan University, Xiangtan, Hunan, China, in 2019, and the M.Sc. degree in computer science from East China Normal University, Shanghai, China, in 2022. He is currently pursuing the Ph.D. degree in computer science with the Victoria University of Wellington, Wellington, New Zealand.

His current research interests include symbolic regression, genetic programming, evolution computation, and statistical machine learning.

Qi Chen (Member, IEEE) received the B.E. degree in automation from the University of South China, Hengyang, Hunan, China, in 2005, the M.E. degree in software engineering from the Beijing Institute of Technology, Beijing, China, in 2007, and the Ph.D. degree in computer science from the Victoria University of Wellington (VUW), Wellington, New Zealand, in 2018.

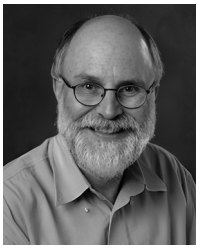
She is currently a Senior Lecturer of Artificial Intelligence with the School of Engineering and Computer Science, VUW. Her research interests, including machine learning, evolutionary computation, feature selection, feature construction, transfer learning, domain adaptation, and statistical learning theory.

Dr. Chen serves as a Reviewer for international conferences, including AAAI and IJCAI, and for international journals, including IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION and IEEE TRANSACTIONS ON CYBERNETICS.

Bing Xue (Senior Member, IEEE) received the B.Sc. degree from the Henan University of Economics and Law, Zhengzhou, China, in 2007, the M.Sc. degree in management from Shenzhen University, Shenzhen, China, in 2010, and the Ph.D. degree in computer science from the Victoria University of Wellington, Wellington, New Zealand, in 2014.

She is currently a Professor of Artificial Intelligence, the Deputy Director of Centre for Data Science and Artificial Intelligence, and the Deputy Head of the School of Engineering and Computer Science, Victoria University of Wellington. She has over 300 papers published in fully refereed international journals and conferences and her research focuses mainly on evolutionary computation and machine learning.

Dr. Xue is currently the Chair of IEEE CIS Evolutionary Computation Technical Committee and an Editor of IEEE CIS NEWSLETTER. She has also served as an Associate Editor for several international journals, such as *IEEE Computational Intelligence Magazine*, IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, and *ACM Transactions on Evolutionary Learning and Optimization*. She is also a Fellow of Engineering New Zealand.



Wolfgang Banzhaf (Member, IEEE) received the Dr.rer.nat (Ph.D.) degree from the Department of Physics, Technische Hochschule Karlsruhe (currently, Karlsruhe Institute of Technology), Karlsruhe, Germany, in 1985.

He was a University Research Professor with the Department of Computer Science, Memorial University of Newfoundland, St. John's, NL, Canada, where he served as the Head of Department from 2003 to 2009 and from 2012 to 2016. He is the John R. Koza Chair of Genetic Programming with the Department of Computer Science and Engineering and a member of the BEACON Center for the Study of Evolution in Action, Michigan State University, East Lansing, MI, USA. His interests include studies of self-organization and the field of artificial life. He has become more involved with network research as it applies to natural and man-made systems. His research interests are in the field of bioinspired computing, notably evolutionary computation, and complex adaptive systems.

Mengjie Zhang (Fellow, IEEE) received the B.E. and M.E. degrees from the Artificial Intelligence Research Center, Agricultural University of Hebei, Hebei, China, in 1989 and 1992, respectively, and the Ph.D. degree in computer science from RMIT University, Melbourne, VIC, Australia, in 2000.

He is currently a Professor of Computer Science and the Director of Centre for Data Science and Artificial Intelligence, Victoria University of Wellington, Wellington, New Zealand. He has published over 800 research papers in refereed international journals and conferences. His current research interests include genetic programming, image analysis, feature selection and reduction, job-shop scheduling, and evolutionary deep learning and transfer learning.

Prof. Zhang is a Fellow of the Royal Society of New Zealand and the Engineering New Zealand, and an IEEE Distinguished Lecturer.