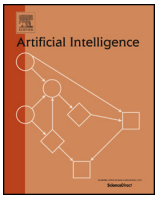
Artificial Intelligence 322 (2023) 103962

Contents lists available at ScienceDirect

Artificial Intelligence

journal homepage: www.elsevier.com/locate/artint



Iterative genetic improvement: Scaling stochastic program synthesis

Yuan Yuan [a,b], Wolfgang Banzhaf [b,c,*]

[a] School of Computer Science and Engineering, Beihang University, Beijing, 100191, China
[b] Department of Computer Science and Engineering, Michigan State University, East Lansing, 48864, MI, USA
[c] BEACON Center for the Study of Evolution, Michigan State University, East Lansing, 48864, MI, USA

ARTICLE INFO

Article history:
Received 2 March 2022
Received in revised form 7 June 2023
Accepted 9 June 2023
Available online 15 June 2023

Keywords:
Genetic programming
Genetic improvement
Evolutionary computation
Program synthesis
Artificial intelligence

ABSTRACT

Program synthesis aims to *automatically* find programs from an underlying programming language that satisfy a given specification. While this has the potential to revolutionize computing, how to search over the vast space of programs efficiently is an unsolved challenge in program synthesis. In cases where large programs are required for a solution, it is generally believed that *stochastic* search has advantages over other classes of search techniques. Unfortunately, existing stochastic program synthesizers do not meet this expectation very well, suffering from the scalability issue. To overcome this problem, we propose a new framework for stochastic program synthesis, called iterative genetic improvement. The key idea is to apply genetic improvement to improve a current reference program, and then iteratively replace the reference program by the best program found. Compared to traditional stochastic synthesis approaches, iterative genetic improvement can build up the complexity of programs incrementally in a more robust way. We evaluate the approach on two program synthesis domains: list manipulation and string transformation, along with a number of general program synthesis problems. Our empirical results indicate that this method has considerable advantages over several representative stochastic program synthesizer techniques, both in terms of scalability and of solution quality.

© 2023 Elsevier B.V. All rights reserved.

1. Introduction

Program synthesis is a longstanding challenge in artificial intelligence (AI) and has even been considered by some as the "holy grail of computer science" [1]. The goal of program synthesis is to *automatically* write a program that has a behavior consistent with a specification. The specification itself can be expressed in various forms such as logical specification [2,3], examples [4–7] or in natural language descriptions [8–10]. Program synthesis techniques have been used successfully in many real-world application domains including data wrangling [5,11,12], program repair [13,14], computer graphics [15,16] and others. Furthermore, viewing machine learning tasks as program synthesis [17,18] can potentially address some of the difficulties of modern deep learning approaches, e.g., data hunger or poor interpretability, leading to more reliable and interpretable AI.

* Corresponding author.
E-mail address: banzhafw@msu.edu (W. Banzhaf).

https://doi.org/10.1016/j.artint.2023.103962
0004-3702/© 2023 Elsevier B.V. All rights reserved.

Recently, program synthesis has seen a renaissance in several different research communities, particularly in the programming language and the machine learning community. Most research efforts focus on developing more effective search techniques since program synthesis is a notoriously difficult combinatorial search problem. Enumerative search-based synthesis [19] enumerates programs in the search space according to a specific order, a topic well studied in the literature. But this class of techniques is usually inefficient and needs to be augmented with strategies (i) to prune the search space [5,20], (ii) to bias the search using probabilistic models [7,21,22], or (iii) to split a large problem via divide-and-conquer strategies [20,23]. Despite such augmentations, enumerative search still struggles to scale to large program sizes, as the search space grows exponentially with program size. Another popular class of search techniques is to reduce the program synthesis problem by constraint solving, and leverage off-the-shelf SAT/SMT solvers to efficiently explore the search space [24–26]. Although this class of techniques has achieved impressive results [27], it also has difficulty in synthesizing large programs due to the limited power of the underlying SAT/SMT solvers.

Stochastic program synthesis (SPS) employs *stochastic search methods* such as the Metropolis-Hastings (MH) algorithm [28] or genetic programming (GP) [29,30] in order to explore the space of programs. Compared to the above mentioned search techniques, SPS is a very promising technique for addressing harder synthesis problems that require larger programs [31]. However, this potential of SPS is currently far from being fully exploited. According to the experiments done in [32], a typical MH approach for program synthesis has been shown to be not very effective compared to other approaches. One possible drawback of the MH approach is that it cannot make the corresponding local changes when a program is close to being correct, because the proposal distribution used can only lead to big changes in a program [33].

The Genetic Programming (GP) method was always intended for automatic programming, but has suffered from scalability issues for quite a long time [34]. For hard problems, traditional GP can be very slow and the well-known continued growth of programs with little or even no fitness improvement ("bloat") greatly limits its applicability [35]. Although there have been some methods to handle code growth [36], they usually result in unsatisfactory or even worse performance [35,36]. Besides MH and GP, stochastic local search (SLS) [37], such as simulated annealing, would be another alternative, but it has received little attention [38] in program synthesis. One serious limitation of SLS is that it can easily get trapped in local minima, given that the search space of programs is often highly rugged and contains many plateaus [39,40].

In this paper, we propose an *iterative genetic improvement* (IGI) method to make SPS more scalable for finding large programs as solutions. Our approach is inspired by a practical software development technique called iterative enhancement [41], where human programmers write a simple initial implementation and then enhance it iteratively until a final implementation is achieved. Based on this paradigm, our basic idea is to consider a sequence of program improvements just like the evolution from a primitive cell to a sophisticated and specialized cell [42]. Our proposed IGI starts with a random program, then the current program is improved iteratively by applying genetic improvement (GI) [43] which evolves modifications to the current program. When no improvements can be made any more by GI, a perturbation operator is applied that will allow to continue the iterative improvement process. Because IGI carefully rewrites small parts of a current program via GI in each iteration, it can largely avoid unnecessarily big code changes like those of the MH approach [32], or the code growth problem traditional GP faces, leading to faster search. Moreover, IGI considers a neighborhood of the current program that is much larger than SLS based techniques such as SA [38], so it is more able to avoid or escape local minima and plateaus. We demonstrate the superiority of IGI on two different program synthesis domains in comparison with several representative SPS techniques. Our experimental results show that IGI can outperform all of the compared techniques by a large margin in terms of scalability, and our results also indicate that IGI appears to be less prone to overfitting. Moreover, to demonstrate the power of IGI as a general program synthesizer, we compare IGI using expression trees with two state-of-the-art GP systems for general program synthesis.

The rest of this paper is organized as follows. In Section 2 we discuss some background material for our study. Section 3 describes the proposed IGI method in detail. Section 4 and Section 5 present empirical results on domain-specific program synthesis and general program synthesis, respectively. Section 6 briefly introduces other work related to this study. Section 7 summarizes and concludes.

2. Preliminaries and background

2.1. Domain-specific language

A domain-specific language (DSL) is a computer language that is specifically created for a particular domain. Program synthesis is usually based on a given DSL in order to take a first cut at the space of possible programs.

A DSL can be expressed as a *primitive set* which includes *functions* and *terminals* assumed to be useful for solving problems in a specific domain. Consider a toy DSL with the primitive set {`ADD`, `SUB`, `EQ`, `ITE`, 0, 1, IN0, IN1}. In this primitive set, `ADD`, `SUB`, `EQ` and `ITE` are functions explained in Table 1, and the remaining four entities are terminals. Among the four terminals, integers 0 and 1 are constants, and integers IN0 and IN1 are external inputs to the program.

The primitive set of a DSL defines the complete hypothesis space of possible programs. Each program can be represented as an *expression tree*. Fig. 1 shows such a tree for a valid program in the above toy DSL.

In our experiments, we will consider two DSLs. One is useful for list manipulation (referred to as DSL-LM), and the other is useful for string transformation (referred to as DSL-ST). They are described in detail in Appendix A and Appendix B, respectively.

Table 1
Description of functions in the DSL.

Symbol	Arguments	Return Type	Description
`ADD`	x: Integer; y: Integer	Integer	Return $x + y$.
`SUB`	x: Integer; y: Integer	Integer	Return $x - y$.
`EQ`	x: Integer; y: Integer	Boolean	If x is equal to y, return true, otherwise return false.
`ITE`	c: Boolean; x: Integer; y: Integer	Integer	If c is true, return x, otherwise return y.

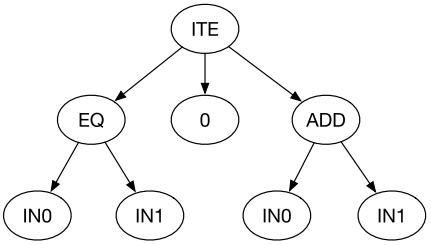


Fig. 1. The tree of the valid program `ITE(EQ(IN0, IN1), 0, ADD(IN0, IN1))` in the above toy DSL.

Table 2
A PBE task with four I/Oexamples (each input contains two integers) and a possible program in the toy DSL that satisfies all of the four examples.

Input (I_i)	Output (O_i)	Program
3, 3	0	`ITE(EQ(IN0, IN1), 0, ADD(IN0, IN1))`
2, 5	7	
8, 1	9	
6, 6	0	

2.2. Programming by example

Programming by example (PBE) is a subfield of program synthesis, where the specification is given in the form of input-output (IO) examples.

Assume that we have a DSL that defines the space of all possible programs denoted by $\mathcal{P}$. In PBE, a task is described by a set of I/Oexamples $\mathcal{X} = \{(I_1, O_1), \ldots, (I_n, O_n)\}$. We can then say we have solved this task if we find a program $\mathbf{p} \in \mathcal{P}$ that can map correctly every input in $\mathcal{X}$ to the corresponding output, i.e., $\mathbf{p}(I_i) = O_i, \forall i = 1, 2, \ldots, n$. Table 2 shows a PBE task with four I/Oexamples and a potential correct program from the toy DSL described in Section 2.1.

Our goal is to build a program synthesizer which can find such a program $\mathbf{p}$ in $\mathcal{P}$ according to the I/Ospecification $\mathcal{X}$ within a certain time limit.

2.3. Genetic improvement

Genetic improvement (GI) [43] is the use of an automated search algorithm, genetic programming, to improve an existing program. GI typically conducts the search over the space of *patches*, where a patch constitutes a sequence of edits that are applied to the original program and corresponds to a modified program. Compared to the original program, an improved version needs to have better functional properties (e.g., by eliminating buggy behavior) [44,13,14,45] or better non-functional properties (e.g., shorter execution time or smaller energy consumption) [46–48], depending on the application scenario or an user's goal. GI has achieved notable success in software repair and optimization. GenProg [44] is a pioneer system for test-suite based software repair, which is typically based on GI. Inspired by GenProg's limitations, ARJA systems [14,45] introduce a number of enhancements such as lower-granularity patch representation, finer-grained fitness function and multi-objective formulation, which can achieve state-of-the-art repair performance on a set of real Java bugs. Recently, a GI approach called SapFix [49] has been deployed successfully at Facebook to suggest fixes for six production systems. As for software optimization, GI has been recently extended to more novel applications [50]. Some examples include the performance tuning of GPU kernels [48] and the speedup of a genetic programming system [51]. In addition, there are several emerging tools [52,53] available that have been developed to facilitate the use of GI for improving non-functional properties of software.

Note that GI normally takes real-world source code as a starting point and improves it by introducing some small changes. So the programs studied in GI applications are usually written in a general-purpose programming language (GPL) such as C or Java. In contrast, the goal of program synthesis is to create a program that realizes user intent from scratch. So

traditional program synthesis is largely applied to a particular domain and works over a DSL. The DSL should be expressive enough to represent a variety of tasks in the target domain, but also should be restricted enough to allow efficient search. Program synthesis in a GPL can automate human programming in a more general sense, but would be less efficient when applied to a domain where a DSL can be properly defined.

In our proposed approach, GI improves a program with respect to a fitness function measuring how well the program performs over a given set of input-output examples.

3. Iterative genetic improvement

3.1. Overview

The IGI framework proposed here is described in Algorithm 1. First, in Step 1, we create K program trees randomly using ramped half-and-half initialization [29] with a depth range of $[d_{\min}, d_{\max}]$. The best among these K random programs is set to become the initial program $\mathbf{p}_0$. This step is intended to find a good starting point for the search by sampling from a (substantial) number of programs instead of a single one.

Algorithm 1 Framework of the Proposed IGI.

1: $\mathbf{p}_0 \leftarrow$ `InitProg`$(d_{\min}, d_{\max}, K)$
2: $\mathbf{p} \leftarrow$ `IterGenImprov`$(\mathbf{p}_0)$
3: **while** termination criterion is not satisfied **do**
4: $\quad\mathbf{p}' \leftarrow$ `Perturbation`$(\mathbf{p})$
5: $\quad\mathbf{p}'' \leftarrow$ `IterGenImprov`$(\mathbf{p}')$
6: $\quad\mathbf{p} \leftarrow$ `AcceptanceCriterion`$(\mathbf{p}, \mathbf{p}'')$
7: **end while**
8: **return** the best program found

In Step 2, we make incremental improvements from $\mathbf{p}_0$ by applying the GI procedure *iteratively* until a program $\mathbf{p}$ is reached that cannot be further improved by the adopted GI. In each iteration GI tries to produce an improved version p_{i+1} by searching for modifications to the current program p_i. Fig. 2 illustrates this process where $\mathbf{p}_l = \mathbf{p}$, and we call the process from $\mathbf{p}_i$ to $\mathbf{p}_{i+1}$ an *epoch*, where $\mathbf{p}_{i+1}$ is an improved version of $\mathbf{p}_i$ obtained through GI.

After Step 2, to continue the search, we need to generate a new starting program for `IterGenImprov`. To do this, a naive strategy is to randomly produce a new program as the starting program. However this strategy is obviously not efficient because the search history is completely discarded. Here we follow the basic idea of iterated local search [54]. That is, we apply some perturbation operator to the new $\mathbf{p}$ that leads to an intermediate program $\mathbf{p}'$ (Step 4 in Algorithm 1). Then `IterGenImprov` restarts the search from $\mathbf{p}'$ and returns $\mathbf{p}''$ (Step 5 in Algorithm 1). In Step 6, `AcceptanceCriterion` will decide to which program the next time `Perturbation` is applied. In this paper, this criterion just simply returns the better of $\mathbf{p}$ and $\mathbf{p}''$. Steps 4–6 are iterated until some termination criterion is met. In Fig. 3, we further provide a schematic overview of our IGI approach.

As can be seen in IGI, we need to compare the quality of two programs frequently. This is aided by a predefined *fitness function* which can measure how well a program satisfies the given specification. Suppose we are given a set of n input-output examples $\{(I_1, O_1), \ldots, (I_n, O_n)\}$, the fitness function of a program $\mathbf{p}$ can be defined as $fitness(\mathbf{p}) = \frac{1}{n}\sum_{i=1}^{n} sim(O_i, \mathbf{p}(I_i))$, where sim function indicates the similarity between the expected output O_i and the actual output $\mathbf{p}(I_i)$. In our study, for DSL-LM, sim returns 1 if $O_i = \mathbf{p}(I_i)$ otherwise returns 0. As for DSL-ST, we use a finer-grained sim function which returns the normalized Levenshtein similarity[1] between two strings. In our approach, a program with larger fitness is better, while in the case of two programs having the same fitness, the program with smaller size is deemed better according to Occam's razor. A program $\mathbf{p}$ will satisfy the given specification iff it achieves maximum fitness (i.e., $fitness(\mathbf{p}) = 1$).

Section 3.2 will detail how to apply GI to the current program $\mathbf{p}_i$ to get an improved version $\mathbf{p}_{i+1}$, which corresponds to one epoch in `IterGenImprov` (see Fig. 2). Section 3.3 will explain how to conduct the perturbation operator, which corresponds to Step 4 in Algorithm 1.

3.2. Applying genetic improvement

3.2.1. Patch representation

A program different from the program we want to improve can be coded as the difference between the original and the new program. In computing, this is known as a program patch, represented as a *sequence* of edits to the program's expression tree. In this patch representation, we define three kinds of edits: *replacement*, *insertion* and *deletion*. Syntactic and type constraints are considered in these edits in order to ensure that the modified tree remains legal. To randomly

[1] We use the `levenshtein.normalized_similarity` function from https://github.com/life4/textdistance, calculating a value between [0, 1], with 1 returned if the two strings are the same.

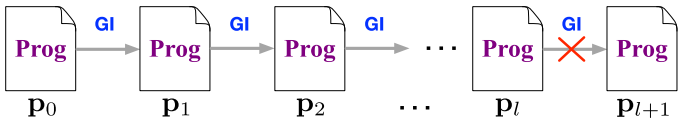


Fig. 2. Illustration of Step 2 in Algorithm 1. $\mathbf{p}_{i+1}$ is an improved version of $\mathbf{p}_i$ for $i = 0, 1, \ldots, l-1$, that is obtained by applying GI to $\mathbf{p}_i$. Since $\mathbf{p}_{l+1}$ is not better than $\mathbf{p}_l$ (i.e., GI fails to improve $\mathbf{p}_l$), $\mathbf{p}_l$ is finally returned by `IterGenImprov`.

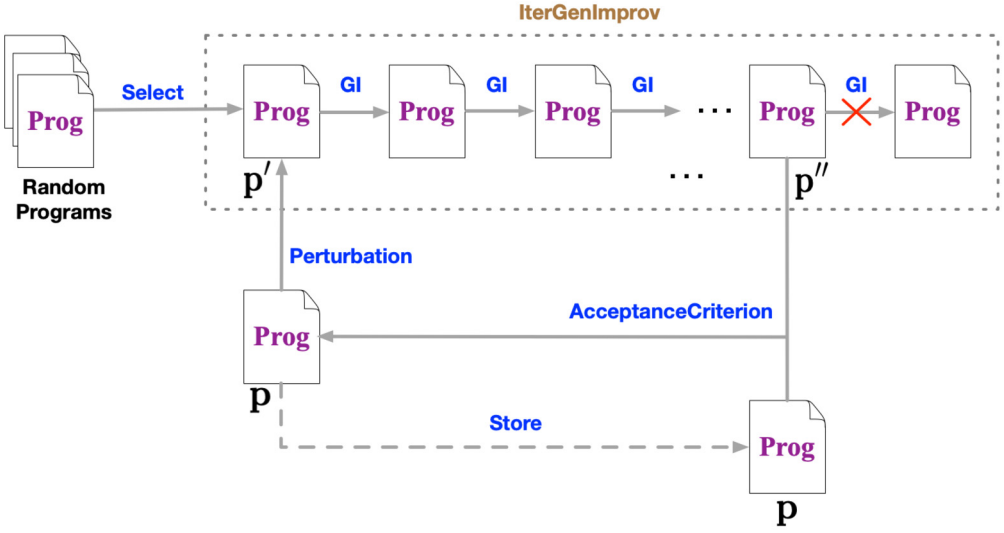


Fig. 3. Schematic overview of the proposed IGI.

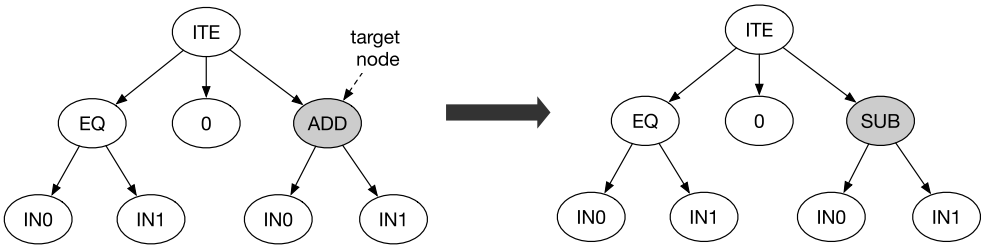


Fig. 4. Illustration of a replacement edit (shaded nodes are changed).

generate an edit, we first select a node randomly in the expression tree that is called target node. Then we choose one of the three kinds of edits randomly.

1) **Replacement**: If we choose to perform replacement, the target node is replaced by another random primitive from the primitive set which has the same number of arguments, the same argument types and the same return type. This is illustrated in Fig. 4.

2) **Insertion**: If we choose to perform insertion, for the primitive that can be inserted, its return type and at least one of its argument types should be the same as the return type of the target node. Such a primitive is selected at random from the primitive set. Then this primitive will replace the target node, and the subtree rooted at the target node will become one of its child tree that requires the same data type. All the remaining children of this primitive will be selected randomly from the set of terminals with the corresponding data types. This is illustrated in Fig. 5.

3) **Deletion**: If we choose to perform deletion, one node is randomly chosen from the children of the target node that have the same return type as the target node. The subtree rooted at this node will replace the subtree rooted at the target node. This is illustrated in Fig. 6.

A patch contains a list of concrete edits and each edit is performed sequentially when applying the patch. This is illustrated in Fig. 7.

In our study, we find that the above replacement/insertion edits sometimes struggle to introduce the following two kinds of primitives into the program. First, if the return type of a primitive is different from all its argument types (e.g., the primitive `LEN` in the DSL-ST), we know that it is hard or even impossible to bring the two kinds of primitives using the above insertion edit. Second, if there are no corresponding terminals for some of its argument types of a primitive (e.g., the primitive `Take` in the DSL-LM), the situation is similar. Also, a replacement edit might have little chance to get introduced

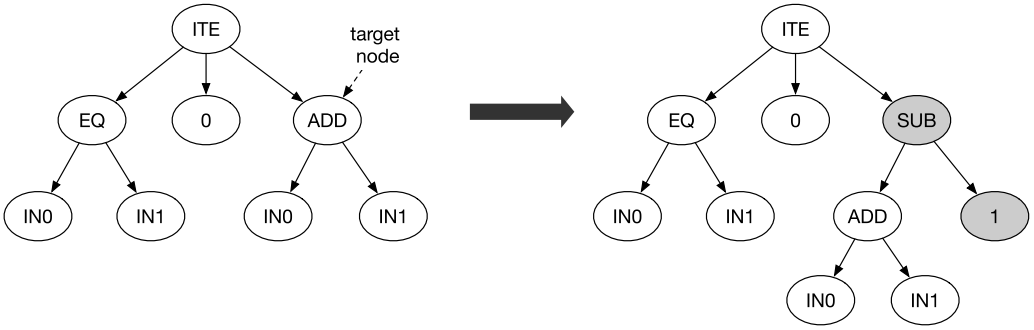


Fig. 5. Illustration of an insertion edit (shaded nodes are the newly inserted nodes).

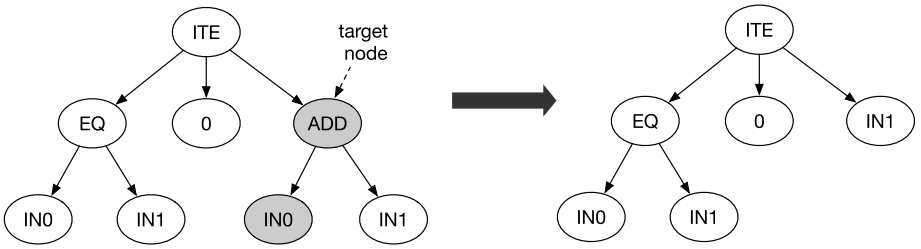


Fig. 6. Illustration of a deletion edit (shaded nodes are removed from the original tree).

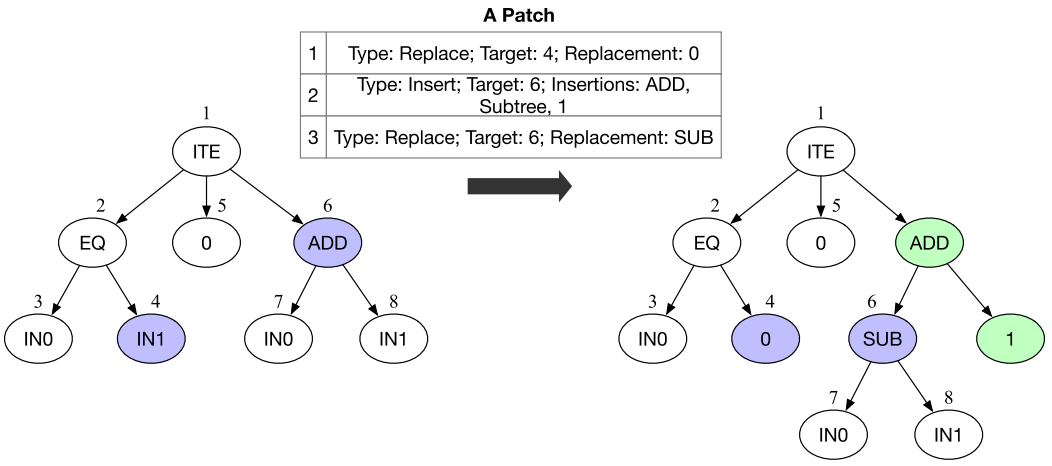


Fig. 7. Illustration of a patch that contains three edits. The number above the node is the ID of the node. The blue shaded nodes are changed by replacement edits and the green shaded nodes are the new nodes added by the insertion edit. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

into a program, if there are few (or even no) primitives with the same return type and argument types in the set. Should the desired program really require this primitive, the search will become inefficient. We address the generation of replacement and insertion edits to introduce the two kinds of primitives easily as follows:

For the replacement edit, when the target node is a leaf, a function with the same return type is also allowed to replace the target node and the arguments of this function will be filled with random terminals having the corresponding types. This is demonstrated in Fig. 8.

As for the insertion edit, when the algorithm fails to find a terminal for a child of the inserted node, it instead chooses a function with the desired return type and fill its arguments with random terminals having the corresponding types. This is demonstrated in Fig. 9.

Note that there may exist conflicts between two edits. For example, a replacement edit as shown in Fig. 4 and a deletion edit as shown in Fig. 6 cannot take effect at the same time. We resolve such conflicts at the time of generating or applying a patch by disabling the latter one if it conflicts with a previous edit.

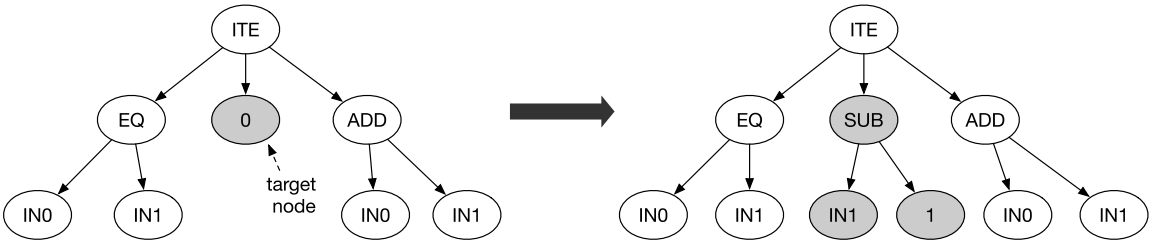


Fig. 8. Illustration of a replacement edit (shaded node in the original tree is replaced by the shaded nodes in the new tree).

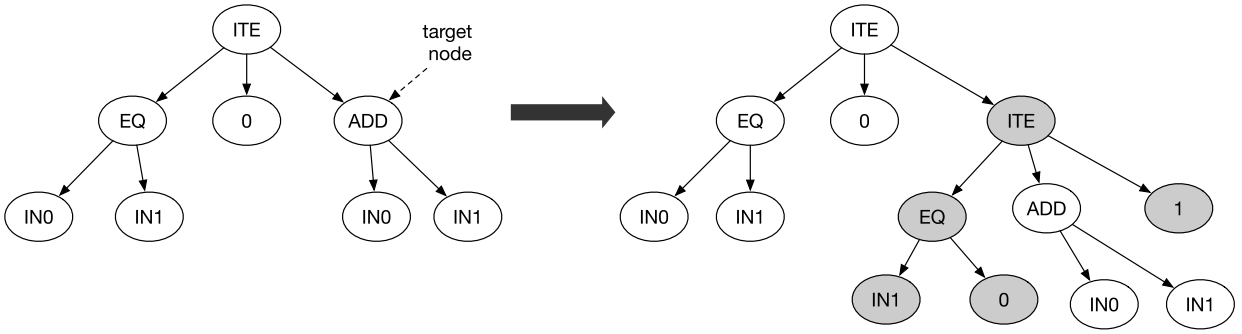


Fig. 9. Illustration of an insertion edit (shaded node are the newly inserted nodes).

Using this patch representation, we can search for patches to the current program $\mathbf{p}_i$ that are able to produce an improved version $\mathbf{p}_{i+1}$. In our framework, we provide two alternative search algorithms: stochastic beam search (SBS) and linear genetic programming (LGP) [55], which are described in the next subsections 3.2.2 and 3.2.3, respectively.

3.2.2. Stochastic beam search

In stochastic beam search (SBS), the search for patches of length $L+1$ is based on the patches of length L (i.e., there are already L edits in the patch). Suppose that we currently have a set of B patches of length L. Then for each of the B patches, we produce C copies and append a randomly generated edit to each copy where the appended edit should not conflict with the previous L edits. Having produced a total of $B \times C$ patches of length $L+1$, tournament selection is used to select B patches from the total $B \times C$. Based on these B patches of length $L+1$, SBS continues to explore further patches, this time of length $L+2$.

SBS starts the search from B empty patches (i.e., length $L=0$), with a limiting parameter $L_{\max}$ used to restrict the maximum length of allowed patches. Once SBS finds a program $\mathbf{p}_i'$ that is better than the current program $\mathbf{p}_i$, we need to decide whether to continue SBS or just return $\mathbf{p}_i'$ as $\mathbf{p}_{i+1}$ for the next GI epoch. In this study, our strategy is that if $\mathbf{p}_i'$ is the best program ever found from the beginning of IGI, SBS in this GI epoch is deemed to be fruitful and continues searching for better return programs until it reaches $L_{\max}$. Otherwise $\mathbf{p}_i'$ can be simply returned as $\mathbf{p}_{i+1}$. Note that SBS may fail to find any improved version of the current program $\mathbf{p}_i$ after reaching $L_{\max}$, in this case the perturbation operator will be invoked.

3.2.3. Linear genetic programming

In the linear genetic programming (LGP) approach, the search starts with a population of N random patches denoted $\mathcal{P}_0$. To produce patches with diverse lengths, each patch length in $\mathcal{P}_0$ is drawn from a $1 + Poisson(1)$ distribution, where $Poisson(\lambda)$ is a Poisson distribution with parameter $\lambda = 1$.

In the g-th generation of LGP, we use tournament selection to select two parent patches from $\mathcal{P}_g$ and apply *crossover* and *mutation* to the two patches to generate two offspring. By repeating this process $N/2$ times, N offspring patches are generated which constitute the next population $\mathcal{P}_{g+1}$.

One-point crossover between two parent patches is used. Suppose that the length of the two parents is L_1 and L_2, respectively. We use a cut point in the two parents given by $\lfloor \alpha L_1 \rfloor$ and $\lfloor \alpha L_2 \rfloor$, where $\alpha \in (0, 1)$ is a random value. The edits after the cut points are swapped between the two parent patches, leading to two offspring.

Mutation is then applied to each of the two offspring patches. When applying mutation to a patch, an edit in the patch is selected uniformly at random, and with equal probabilities removed, replaced with a random edit, or a random edit is inserted after the selected one.

In LGP, we use the same strategy as in SBS to determine when to terminate the search and return the best program LGP has found during the current epoch of GI.

3.3. Perturbation operator

The goal of the perturbation operator is to provide a new good starting program for the next application of `IterGenImprov` in case we get stuck.

When applying the perturbation to $\mathbf{p}$ (Step 4 in Algorithm 1), some useful components of $\mathbf{p}$ need to be conserved, in order to provide a new good starting program for the next application of `IterGenImprov`. At the same time the perturbation should not be too small, in order to avoid staying stuck in the same local optimum as the previous epoch of `IterGenImprov`.

In our framework, the perturbation operator works as follows: First, we collect a set of nodes denoted by $\mathcal{N}$ from the expression tree of $\mathbf{p}$, where the size of the subtree rooted at each node is not smaller than $S_{\min}$. Second, we randomly select a node ν from $\mathcal{N}$. For convenience, the subtree rooted at node ν is denoted by $\mathcal{T}$ and its size is denoted by $S_{\mathcal{T}}$. Third, we randomly generate M trees with size in the range $[1, S_{\mathcal{T}}]$ and the same return type as $\mathcal{T}$. Fourth, we replace $\mathcal{T}$ in the tree of $\mathbf{p}$ with each of the M trees in turn, and obtain M new programs. Last, the best program among the M programs is returned as the perturbed program $\mathbf{p}'$.

In addition, we handle two special cases: One is that $\mathcal{N} = \emptyset$ and the other is that the selected node ν is the root node. For both of these cases, we invoke the initialization function `InitProg` to produce a program that is used as the perturbed program $\mathbf{p}'$.

As can be seen from above, $S_{\min}$ is the parameter that restricts the minimum strength of a perturbation. Since we know that a random subtree replacement usually leads to a bad fitness of the new program in program synthesis, our perturbation operator generates M candidate program variants instead of a single one. Moreover, with a relatively low probability, the perturbation operator can ignore much (when ν is the node very close to root) or even all (when ν is the root node) of the information of $\mathbf{p}$, which can help to explore other promising regions of the program space.

4. Experiments on domain-specific program synthesis

In this section, we conduct experiments on two domain-specific applications. All techniques investigated here explore the same *search space* of expression trees. We mainly want to demonstrate the superiority of the *search strategy* of IGI.

4.1. Experimental setup

The IGI framework is instantiated here with SBS (see Section 3.2.2) and LGP (see Section 3.2.3), resulting in two algorithms IGI-SBS and IGI-LGP, respectively. The source code is implemented in Python and has been made available at GitHub.[2] All the experiments in this section are conducted on the Intel Xeon E5-2680 2.4 GHz CPU processor with 20 GB memory.

4.1.1. Benchmarks

We evaluate the algorithms on synthesis tasks from two different application domains: list manipulation and string transformation. The corresponding DSLs for the two domains are described in Appendix A and Appendix B.

For the list manipulation domain, 200 benchmarks[3] are generated, similar to [7]. In particular, when generating a benchmark, we first randomly generate a program with a number of tree nodes in the range between 10 and 15. Then we continuously feed random inputs to the program and obtain the corresponding outputs. This process is terminated until we collect 100 valid input-output examples (i.e., only integers in $[-256, 255]$ are allowed following [7]) or run out of the computing budget. In the latter case, we just discard the program and start over again. Note that in order to test the scalability of SPS techniques, we consider larger oracle programs than existing studies [7,26,56,57]. For example, oracle programs with at most seven nodes were investigated in [7].

As for the second domain, string transformations, we use the dataset consisting of all SyGuS tasks whose outputs are only strings from the PBE-Strings track in 2018 and 2019 [58]. This results in 185 tasks in total. The semantic specifications consist of 2–400 examples.

4.1.2. Baseline algorithms

IGI-SBS and IGI-LGP are compared to the following representative SPS techniques:

MH [32] — This approach is an adaption of the algorithm proposed in [59] for program superoptimization. It uses the Metropolis-Hastings procedure to sample programs. Given the current program $\mathbf{p}$, a variant $\mathbf{p}'$ with the same size is obtained by conducting a random subtree replacement. The probability of adopting $\mathbf{p}'$ as the new current program is given by the Metropolis-Hastings acceptance ratio $\alpha(\mathbf{p}, \mathbf{p}') = \min\{1, \exp(\beta(C(\mathbf{p}) - C(\mathbf{p}'))\}$, where β is a smoothing constant and $C(\mathbf{p}) = \sum_{i=1}^{n}\{1 - sim(O_i, \mathbf{p}(I_i))\}$ indicates the degree to which $\mathbf{p}$ violates a set of given input-output examples. The approach starts search for programs of size $k = 1$, but switches at each step with some probability p_m to search for programs of size $k + 1$ or $k - 1$.

[2] The source code is available at https://github.com/yyxhdy/igi/.

[3] These benchmarks are available at https://github.com/yyxhdy/igi/tree/main/dataset.

Table 3
Main result (in the list manipulation domain) comparing the performance of all algorithms considered.

Statistics	IGI-SBS	IGI-LGP	MH	GP	SIHC	SA
Total solved	141	136	24	65	80	62
Fastest solved	76	51	1	2	22	4
Smallest solved	74	64	6	14	49	14
Average time	581.76	596.28	474.77	726.97	615.26	1135.35
Median time	229.78	173.27	28.85	404.61	198.15	884.54
Average size	12.09	12.31	12.75	16.32	10.41	22.5
Median size	12.0	12.0	8.5	12.0	10.0	15.5

GP [60] — This is a highly optimized genetic programming system, often referred to as TinyGP. It employs a steady state algorithm where only one individual in the population is replaced each generation. During program evolution the system uses tournament selection to select mating parents, and subtree crossover and point mutation to generate offspring.

SIHC [38] — This algorithm uses stochastic iterated hill climbing (SIHC) to discover programs. It starts with a random program as the current program $\mathbf{p}_c$, then a mutation operator called HVL-Mutate is applied to the current program to obtain a variant $\mathbf{p}'_c$. If $\mathbf{p}'_c$ is better than $\mathbf{p}_c$, $\mathbf{p}'_c$ will replace $\mathbf{p}_c$ as the new current program and the search will move onward from it. Otherwise, another mutation to $\mathbf{p}_c$ is tried. The maximum number of mutations that can be applied to $\mathbf{p}_c$ is given by a parameter $T_{\max}$. If none of the $T_{\max}$ variants is better than $\mathbf{p}_c$, $\mathbf{p}_c$ is discarded and a new current program is generated at random.

SA [38] — This algorithm uses simulated annealing (SA) to discover programs. It is somewhat similar to SIHC, but at each step only a single variant $\mathbf{p}'_c$ is generated from $\mathbf{p}_c$ using HVL-Mutate, and the variant $\mathbf{p}'_c$ will be accepted as the new current program with probability $\min\{1, \exp((fitness(\mathbf{p}') - fitness(\mathbf{p}))/T_c)\}$, where T_c is the current temperature which is decreased by an exponential rate.

Note that the original HVL-Mutate does not consider data-type constraints in a DSL, so it is not applicable to synthesis tasks where there are multiple data-types. In our experiments, we replace the HVL-Mutate in SIHC and SA with the replacement/insertion/deletion edit described in Section 3.2.1, in order to ensure a fair comparison.

4.1.3. Parameter settings

The timeout for each run of the algorithm on a benchmark is set to one hour in these experiments, and in each run the algorithm will be terminated at once when a solution program is found.

The key parameters of all algorithms considered are tuned on two additional difficult synthesis tasks,[4] one for each domain. Grid search is used to find the best parameter combination among 81 combinations for IGI-SBS, 72 combinations for IGI-LGP, 50 combinations for MH, 81 combinations for GP, 40 combinations for SIHC and 50 combinations for SA. Details can be seen in Appendix C.

4.2. Results for list manipulation

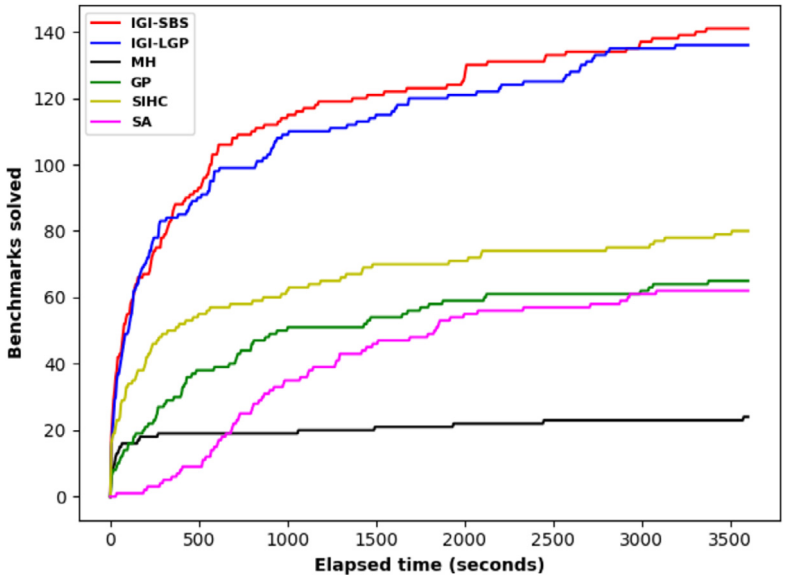
We evaluate IGI-SBS and IGI-LGP on 200 benchmarks from the list manipulation domain and compare them with MH, GP, SIHC and SA. Each algorithm runs only once on each benchmark. Table 3 summarizes the comparison results, where we list for each algorithm the number of benchmarks solved ("Total solved"), the number of benchmarks solved with the fastest solving time ("Fastest solved"), the number of benchmarks solved with the smallest program size ("Smallest solved"), the average and median times to find a solution, and the average and median sizes of solution programs. Here program size refers to the number of nodes in the program's expression tree.

Out of 200 benchmarks, IGI-SBS and IGI-LGP can solve 141 and 136 benchmarks, respectively. Other algorithms perform much worse than IGI-SBS and IGI-LGP in terms of total number of benchmarks solved. The most competitive one is SIHC, but it can only solve 80 benchmarks, which is just about 57% of that by IGI-SBS. GP and SA solve similar number of benchmarks (65 for GP and 62 for SA). MH can only solve 24 benchmarks and is significantly outperformed by all other algorithms.

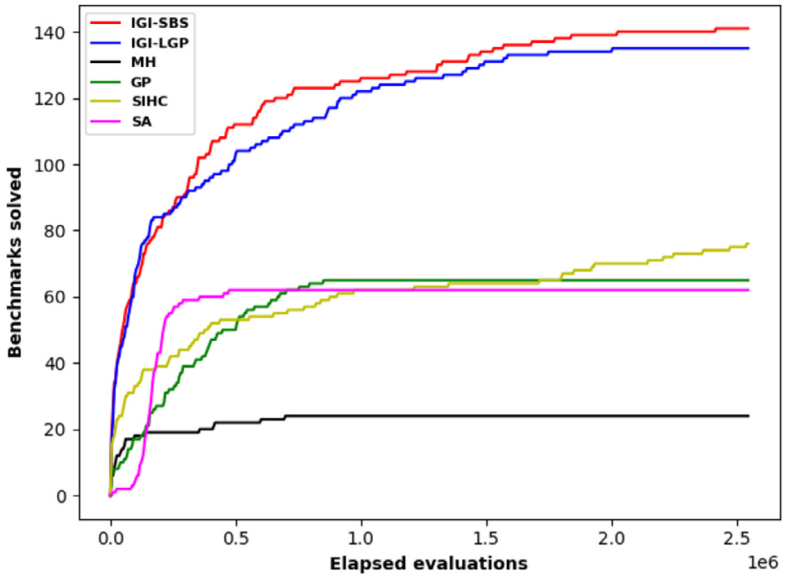
IGI-SBS and IGI-LGP are the fastest solvers in 76 and 51 benchmarks, respectively. Next to IGI-SBS and IGI-LGP, SIHC is the fastest only in 22 benchmarks. In terms of average and median times, IGI-SBS and IGI-LGP show clear advantages over GP and SA, and show similar performance to SIHC. Although both, average and median times consumed by MH are smallest, MH scales poorly.

We also judge solution quality based on the size of a solution program [61]. According to average and median sizes, solutions found by IGI-SBS and IGI-LGP have overall better quality than those found by GP and SA. Compared to IGI-SBS and IGI-LGP, SIHC can generate solutions with smaller average and median sizes. This is reasonable because IGI-SBS and IGI-LGP can solve many benchmarks that require larger solution programs whereas SIHC can not. Moreover, IGI-SBS and IGI-LGP can provide the smallest solutions for 74 and 64 benchmarks respectively, whereas SIHC can only provide the smallest solutions for 49 benchmarks.

[4] The definitions of the two tasks are available at https://github.com/yyxhdy/igi/tree/main/pt.



(a)

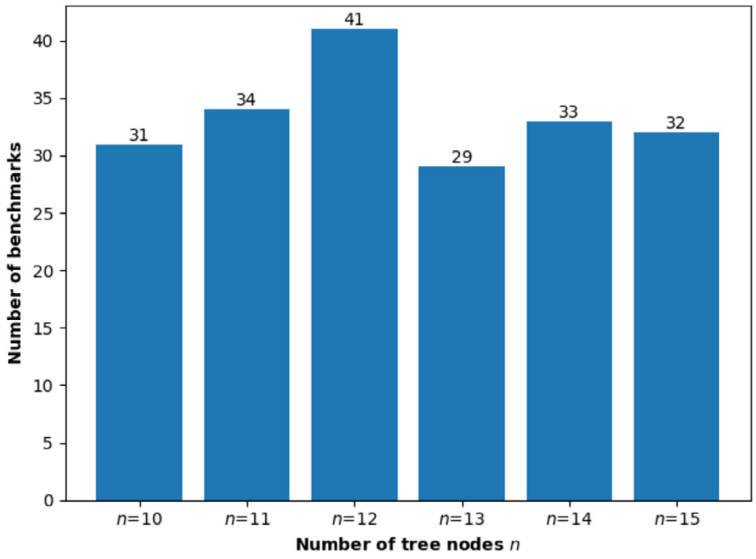


(b)

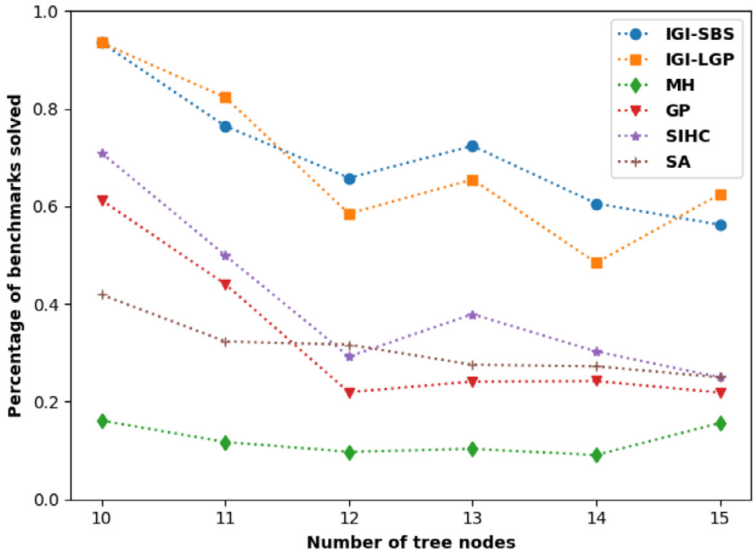
Fig. 10. (a) Number of benchmarks solved versus the computation time; (b) Number of benchmarks solved versus the number of evaluations in the list manipulation domain.

Interestingly, SA needs much more time to find a solution compared to the other algorithms. In addition, the solutions found by SA are also much larger than those by the other algorithms. One possible reason is that the space of programs contains a series of discrete plateaus. SA always accepts solutions with the same fitness, so it may spend much time to traverse these plateaus. Moreover, during this process, many subtree structures with no effect on fitness [62] could be added into the code, producing larger and larger programs. Our proposed IGI explores a large neighborhood of the current program using GI techniques, so it can avoid or escape from plateaus more easily.

In Fig. 10, we plot the number of benchmarks solved versus computation time (Fig. 10(a)) and the number of evaluations (Fig. 10(b)) for each algorithm. Compared to the baselines, the number of benchmarks solved by IGI-SBS or IGI-LGP increases more steadily with elapsed time/evaluations. For MH, almost all the solutions are found during the very early period of the search. Considering that MH searches for progressively larger programs, this implies that its search mechanism is inadequate for synthesizing larger programs. Note that SA gets stuck after a relatively small number of evaluations (i.e., about 5×10^5),



(a)



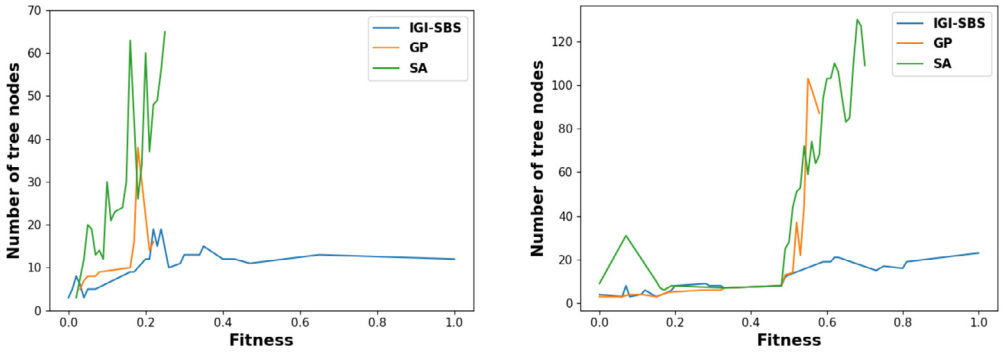
(b)

Fig. 11. (a) Number of benchmarks in each range of program tree size; (b) Percentage of benchmarks solved in each range of program tree size by the considered algorithms.

but these evaluations cost most of the budget time (i.e., about 3000 seconds). This implies that SA tends to examine larger programs whose execution times are longer.

In Fig. 11(a), we bin the generated 200 benchmarks according to the oracle program size (in terms of the number of tree nodes). Fig. 11(b) further shows the percentage of benchmarks solved in each range by the considered algorithms. It is reasonable that the performance of our IGI tends to degrade when the program size gets larger, because the search space of programs would increase exponentially with the increasing of program size. But compared to the other algorithms, our IGI can maintain a large performance margin in all ranges of program size. Note that although our IGI algorithms perform much better, they still fail to solve about 30% benchmarks here. The possible reason is that the fitness landscapes of these missing benchmarks are extremely complex which require more efficient search, and a larger computational budget may help to solve them.

In Fig. 12, we show the program complexity versus fitness on problems L129 and L197, in order to demonstrate the robustness of IGI in terms of building up the complexity of programs. For brevity, we only present the results of IGI-SBS,



(a) On problem L129 (b) On problem L197

Fig. 12. Program complexity versus fitness. The program complexity is measured in terms of the number of tree nodes.

Table 4
Results for 10 hard benchmarks (in the list manipulation domain) that are randomly chosen. The data indicates the number of successful runs out of 50. Best performance is shown in bold.

Benchmarks	IGI-SBS	IGI-LGP	MH	GP	SIHC	SA
L15	20	**24**	0	1	0	0
L53	**20**	14	0	0	0	0
L82	4	**8**	0	1	0	7
L129	**37**	36	0	1	0	0
L172	**15**	12	0	0	0	0
L179	29	**35**	0	12	0	10
L180	26	**32**	0	0	4	4
L183	30	**39**	4	13	1	0
L188	**32**	25	0	0	0	0
L197	**32**	28	0	0	0	0

GP and SA in the figure. We can clearly see that in GP and SA, the program complexity can dramatically increase without much fitness improvement. Whereas in IGI-SBS, the program complexity can be built up very stably along with a series of significant fitness improvements.

Due to the stochastic nature of SPS techniques we measure the performance of each technique as the number of runs out of 50 that solve the benchmark, called success rate. Table 4 lists the success rates of each algorithm on 10 hard benchmarks. These benchmarks are randomly chosen out of the 200 benchmarks and can be solved by no more than three techniques, according to the results in Table 3. It can be seen from Table 4 that IGI-SBS and IGI-LGP perform similarly well and obtain the best success rates on some of the 10 benchmarks. IGI-SBS and IGI-LGP perform much better than all baselines except on L82 where SA is very competitive. On L53, L172, L188 and L197, IGI-SBS and IGI-LGP can achieve decent success rates, whereas all the baselines never succeed.

In summary, the IGI algorithms clearly outperform all the baselines in terms of scalability in the list manipulation domain with overall better solution quality. Moreover, they also have an overwhelming advantage in solving hard synthesis tasks.

4.3. Results for string transformation

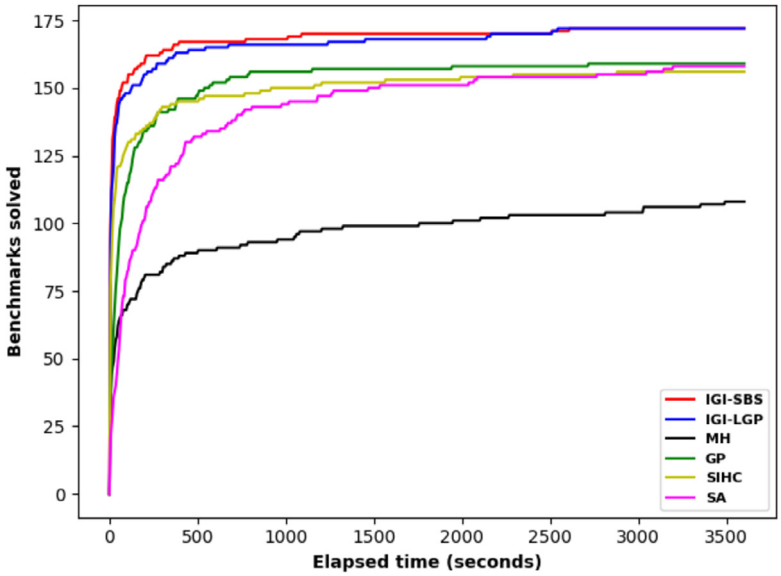
Table 5 shows the main result in the string transformation domain. IGI-SBS and IGI-LGP solve the most number of benchmarks (i.e., 172 for both of them), followed by GP that solves 159. So IGI algorithms again outperform all the other algorithms in terms of scalability. Moreover, they are also superior to others in terms of synthesis time. In particular, IGI-SBS finds solutions in an average time of 73.39 seconds, whereas SIHC, the fastest baseline, consumes 128.16 seconds on average.

In terms of solution size, SA severely suffers from the code growth problem and average solution size reaches over 277, possibly due to the same reason as in Section 4.2. GP suffers from a similar problem with average and median sizes of 42.37 and 27. IGI-SBS and IGI-LGP find solutions that are of comparable size to those by MH and SIHC, but are better in scalability.

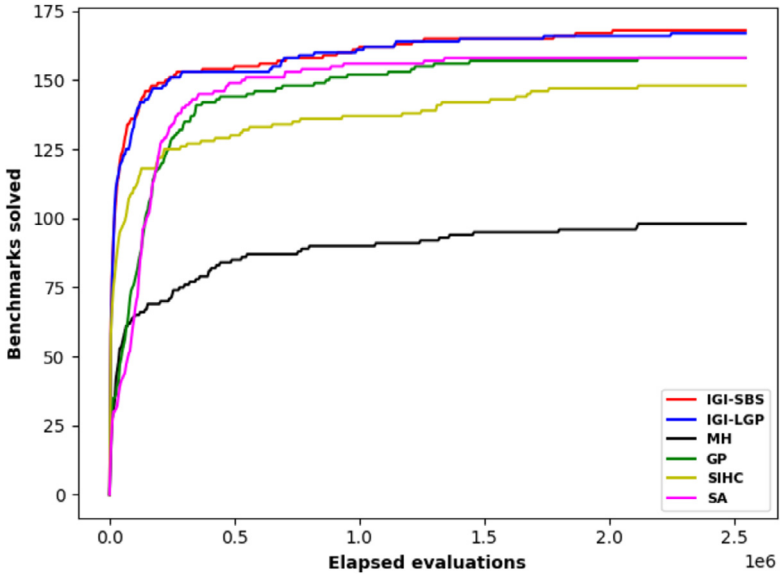
Fig. 13 plots the number of benchmarks solved with the increasing of computational time (Fig. 13(a)) and evaluations (Fig. 13(b)). Unlike in the list manipulation domain, the trajectories of IGI-SBS and IGI-LGP start steep and almost flatten very quickly, indicating that most of benchmarks in this domain do not pose great challenge to IGI.

Table 5
Main result (in the string transformation domain) comparing the performance of all algorithms considered.

Statistics	IGI-SBS	IGI-LGP	MH	GP	SIHC	SA
Total solved	172	172	108	159	156	158
Fastest solved	63	50	2	5	51	3
Smallest solved	56	38	38	14	88	6
Average time	73.39	109.45	360.04	135.3	128.16	325.29
Median time	4.02	5.25	29.56	39.39	6.94	91.79
Average size	19.1	20.78	24.82	42.37	14.28	277.43
Median size	17.0	19.0	15.0	27.0	12.5	120.0

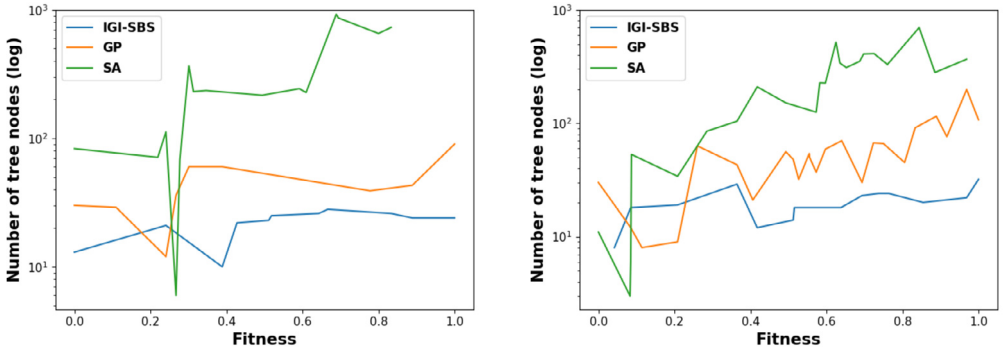


(a)



(b)

Fig. 13. (a) Number of benchmarks solved versus the computation time; (b) Number of benchmarks solved versus the number of evaluations in the string transformation domain.



(a) On problem exceljet2 (b) On problem enwfts

Fig. 14. Program complexity versus fitness. The program complexity is measured in terms of the number of tree nodes. enwfts refers to extract-nth-word-from-text-string.

Table 6
Results for 10 hard benchmarks (in the string transformation domain) that are randomly chosen, where n shows the number of examples for each benchmark. The data indicates the number of successful runs out of 50. Best performance is shown in bold.

Benchmarks	n	IGI-SBS	IGI-LGP	MH	GP	SIHC	SA
31753108	3	**50**	**50**	0	28	1	22
44789427	4	**50**	**50**	0	35	12	42
exceljet2	3	**50**	**50**	0	39	9	18
enwfts	4	**44**	36	0	14	0	23
ewtbwsc	3	**50**	49	0	47	6	10
gmnffn	4	**50**	49	0	22	0	18
p10lr	400	**46**	42	9	26	18	34
stsasc	4	**50**	**50**	0	43	1	31
sncfc	3	46	45	0	**48**	10	41
univ_4	8	**49**	42	0	32	1	27

The full name of some benchmarks are as follows. enwfts: extract-nth-word-from-text-string; ewtbwsc: extract-word-that-begins-with-specific-character; gmnffn: get-middle-name-from-full-name; p10lr: phone-10-long-repeat; stsasc: split-text-string-at-specific-character; sncfc: strip-numeric-characters-from-cell.

Fig. 14 plots the program complexity with the improvement of fitness on problems exceljet2 and enwfts. The observation is similar to that of Fig. 12, which demonstrates the robustness of IGI in building up the program complexity during evolution.

To further examine the performance of IGI, we select 10 hard benchmarks in the same way as we do in the list manipulation domain. Table 6 shows the success rates for each algorithm on these 10 benchmarks. From Table 6, IGI-SBS always succeeds in 6 benchmarks, while IGI-LGP always succeeds in 4. Both IGI-SBS and IGI-LGP achieve higher success rates than all the baselines on all the benchmarks except on sncfc where GP is slightly better. MH performs very poorly, which always fails in 9 out of 10 benchmarks. It seems that although SIHC is very competitive to GP and SA in terms of the number of benchmarks solved, it generally performs worse than GP and SA on hard benchmarks. The similar phenomenon can be observed in the list manipulation domain. This implies that it is more difficult for the simple search mechanism of SIHC to adapt to hard synthesis tasks that may have complex fitness landscapes.

In summary, IGI algorithms perform better than all the other algorithms in terms of scalability in the string transformation domain, without sacrificing the solution quality. In addition, they spend less average time to find solution programs, and also have much stronger ability to address hard synthesis tasks.

4.4. On the issue of generalization

Although the issue of generalization is not the focus of this paper, it is interesting to see how well the solution programs found by IGI algorithms generalize to unseen input-output examples. We generated 100 additional input-output examples for each of the 200 benchmarks in the list manipulation domain using the corresponding oracle programs. Note that this cannot be done in the string transformation domain since oracle programs are unknown for those benchmarks.

Table 7 reports the percentage of solution programs that can generalize to 100 hold-out input-output examples for each considered algorithm. It can be seen that the IGI algorithms have better generalization ability than GP, SIHC and SA. This also implies that the better scalability of IGI is not due to overfitting on given input-output examples. MH achieves the highest generalization percentage, but from a small basis as it only solves 24 relatively easy tasks with overfitting less likely

Table 7
Percentage of solution programs that can generalize to 100 hold-out input-output examples.

Statistics	IGI-SBS	IGI-LGP	MH	GP	SIHC	SA
Total solved	141	136	24	65	80	62
Generalizable	133	122	23	57	68	51
Percentage (%)	94.33	89.71	95.83	87.69	85	82.26

Table 8
The data types and fitness function used for each problem.

Benchmarks	Data Types	Fitness Function
Number I/O	integer, float	float error
Small or Large	integer, boolean, string	Levenshtein distance of strings
Compare String Lengths	integer, boolean, string	boolean error
Last Index of Zero	integer, boolean, vector of integers	integer error
Mirror Image	integer, boolean, vector of integers	boolean error
Sum of Squares	integer, boolean	integer error
Grade	integer, boolean, char, string	char error for grade char
Median	integer, boolean	integer error
Smallest	integer, boolean	integer error
Syllables	integer, boolean, char, string	integer error

to occur. It is worth mentioning that in our experiments all algorithms considered stop searching once a solution is found. We can further alleviate overfitting by letting the algorithm return all solution programs found within the time budget and then choose the smallest one among them.

5. Experiments on general program synthesis

In this section, we conduct experiments on general programming problems. To support this, we define a primitive set for IGI that contains the basic data types and functions used in PushGP [63], and adopt the same approach in strongly typed genetic programming (STGP) [64] to introduce local variables and control flow structures, leading to 263 different functions in total in our IGI system for general program synthesis. Note that general program synthesis systems usually use various program structures such as Push language [65], derivation trees [66] and tangled program graphs [67], so they indeed work over different *search spaces* in terms of syntax. Our IGI framework focuses on the *search strategy* and can be extended to any program structure once their patch representation and perturbation operators are properly defined. In this paper, expression trees are just used as the program structure within IGI. Here we want to demonstrate that, equipped with a set of instructions with similar functionality, our IGI systems using expression trees can be more effective than state-of-the-art genetic programming (GP) systems on general program synthesis.

5.1. Experimental setup

5.1.1. Benchmarks

In the GP literature, there is a set of problems designed as a benchmark suite for general program synthesis [68], which are selected from introductory computer science textbooks. The solutions to most of these problems would require multiple data types and control flow structures, aiming to assess the capabilities of a program synthesis system in generating programs with similar characteristics to human-written programs. In our experiments, we examine ten test problems of varying difficulty from this suite, which are described in Appendix D in detail. Table 8 shows the data types and fitness function used for each problem.

5.1.2. Baseline systems

We compare our IGI-SBS and IGI-LGP with two advanced program synthesis systems, i.e., PushGP[5] and GE,[6] based on genetic programming. Their descriptions are as follows:

PushGP [65,63] — PushGP evolves programs in a stack-based Turing complete language called Push. Push uses stacks to store data and maintains a separate stack for each data type. One salient feature of Push is that it treats code as data and supports a special data type, i.e., `CODE` type. A Push program is made up of a list of instructions and literals. During the execution of an instruction, all its required arguments are popped from their stacks and the results are pushed onto the stacks with the appropriate types. Instructions that explicitly manipulate data of the `CODE` stack can alter the computation process at runtime, making it easy to express complex control structures such as recursion.

[5] An implementation of PushGP in Python is available at https://github.com/erp12/pyshgp.
[6] An implementation of GE in Python is available at https://github.com/PonyGE/PonyGE2.

Table 9
Results for general program synthesis benchmarks, where n shows the number of examples for each benchmark. The data indicates the number of successful runs out of 50. Best performance is shown in bold.

Benchmarks	n	IGI-SBS	IGI-LGP	PushGP	GE
Number I/O	25	**50**	**50**	**50**	49
Small or Large	100	**32**	23	13	17
Compare String Lengths	100	**50**	**50**	14	9
Last Index of Zero	150	8	12	18	**21**
Mirror Image	100	**50**	**50**	43	6
Sum of Squares	50	**8**	7	4	2
Grade	200	**22**	14	3	6
Median	100	**50**	**50**	33	40
Smallest	100	**50**	**50**	46	49
Syllables	100	49	**50**	8	0

GE [69,70,66] — Grammatical evolution (GE) is a GP based system, where a context-free Backus-Naur form (BNF) grammar is used in a genotype-to-phenotype mapping process. With a BNF definition and a fitness function as inputs, GE can evolve complete programs in an arbitrary language. Unlike standard GP, GE uses linear genomes or derivation trees to encode genetic information and maps them to programs according to the input BNF grammar definition, which can help to simplify the application of search operators to different programming languages.

5.1.3. Parameter settings

For our IGI systems, we employ the same parameter settings used in Section 4. As for PushGP and GE, we follow the parameter settings used in the previous studies [68] and [70], respectively, where those settings target the same benchmarks considered here. Both PushGP and GE use lexicase selection [71]. Each system will be run 50 times independently on each benchmark. The timeout for a system in each run is set to one hour, and the system will be terminated at once when a solution program is found.

5.2. Results

In Table 9, we report the success rates for each system on these general program synthesis benchmarks. As can be seen, IGI-SBS and IGI-LGP always succeed in 5 and 6 out of the 10 problems, respectively. They also perform considerably better than PushGP and GE on most of these problems. It is worth noting that the IGI systems can address several synthesis problems well where PushGP and GE struggle. For example, on problem "Syllables", both IGI systems can achieve very high success rates, whereas PushGP rarely succeeds and GE even fails to return any solution program in all runs. In Algorithm 2, we further show the pseudocode of a solution program obtained by IGI-SBS on problem "Syllables". Although the program seems not to be complex to human programmers, it is particularly challenging for a general program synthesis system to generate such a program since the search space of programs is extremely large. Taking a closer look at Algorithm 2, we can see that IGI-SBS indeed obtains a clever solution to "Syllables". Essentially, this program first replaces all vowels in the string with char 'e' and then just returns the number of occurrences of 'e' in the string. Another interesting observation is on the "Sum of Squares" problem. This problem intentionally requires a program that loops over integers from 1 to n and then sum their squares. However, such a program seems very fragile and any slight change will destroy its functionality completely. So there is indeed insufficient guidance for a GP system to move toward this solution. Instead of searching for this fragile program, the IGI systems essentially try to evolve an analytical solution to this problem: $n(n+1)(2n+1)/6$.

Algorithm 2 A solution program synthesized by IGI-SBS on "Syllables".

Input: **s**, a string containing symbols, spaces, digits, and lowercase letters.
Output: n, the number of occurrences of vowels (a, e, i, o, u, y) in **s**.
▷ `ReplaceChar(s, c1, c2)`: replace all c_1 in **s** with c_2
▷ `OccurrencesOfChar(s, c)`: the number of times the char c in **s**
1: **s** ← `ReplaceChar`(**s**, 'a', 'o')
2: **s** ← `ReplaceChar`(**s**, 'i', 'o')
3: **s** ← `ReplaceChar`(**s**, 'y', 'e')
4: **s** ← `ReplaceChar`(**s**, 'u', 'o')
5: **s** ← `ReplaceChar`(**s**, 'o', 'e')
6: n ← `OccurrencesOfChar`(**s**, 'e')

Table 10 shows the percentage of solution programs that can generalize to hold-out input-output examples on each benchmark. It can be seen that, compared to PushGP and GE, the IGI systems show better or at least not worse generalization ability on all the problems except "Small or Large", "Compare String Lengths" and "Grade". It is also noted that IGI systems can achieve high generalization rates (≥ 87.5) except on these three problems. The poor generalization ability on the three problems may be largely due to the inadequacy of the input-output examples provided.

Table 10
Percentage (%) of solution programs that can generalize, where n shows the number of hold-out input-output examples for each benchmark.

Benchmarks	n	IGI-SBS	IGI-LGP	PushGP	GE
Number I/O	1000	96	94	94	91.84
Small or Large	1000	6.25	26.09	7.69	58.82
Compare String Lengths	1000	24	22	28.57	11.11
Last Index of Zero	1000	87.5	91.67	55.56	47.62
Mirror Image	1000	92	92	86.05	16.67
Sum of Squares	49	100	100	75	50
Grade	2000	54.55	50	66.67	50
Median	1000	92	96	63.64	75
Smallest	1000	96	98	86.96	89.8
Syllables	1000	97.96	98	87.5	–

In summary, our IGI systems can outperform state-of-the-art GP systems on general program synthesis tasks. They also show promising performance and creativity in addressing some hard synthesis tasks. In the future, we expect that IGI can be extended to other program structures with better evolvability so that its power can be even better harvested.

6. Related work

Program synthesis is an active research topic including a large and diverse body of work, such as enumerative program synthesis [20,21,23,61], constraint-based program synthesis [24,3,25] and neural program synthesis [11,72–74]. In what follows, we review some prior work on stochastic program synthesis that is most closely related to our proposed IGI.

Metropolis-Hastings algorithm. This line of research started from STOKE [59], which tackles superoptimization problems using a stochastic search strategy known as Metropolis-Hastings (MH). Due to the surprisingly good performance of STOKE on superoptimization, it had stirred great interest with its publication and was regarded as a timely and significant contribution [31] to program synthesis at that time. Alur et al. [32] adapted STOKE and introduced a MH approach over programs represented by trees. This MH approach participated in SyGuS competitions 2014–2016 [58], but it did not achieve very competitive performance compared to other categories of techniques. Since then, the enthusiasm of researchers for stochastic synthesis seemed to drop and there was no stochastic solver participating in SyGuS competitions in the following years. Recently, several studies have been conducted to enhance the performance of STOKE. For example, Bunel et al. [75] proposed to learn the proposal distribution in STOKE parameterized by a neural network, which is expected to better exploit the power of MH; Koenig et al. [76] proposed an effective adaptive restart algorithm for addressing a limitation in STOKE where the search often progresses via a series of plateaus. However, these enhancements have mostly targeted superoptimization, and it is still unclear how to make them amenable to stochastic search over trees.

Genetic programming. Genetic programming (GP) [29,30,60] has a much longer history than Metropolis-Hastings algorithm for the purpose of program synthesis. Indeed, achieving automatic programming is arguably the most aspirational goal in the field of GP [34]. In canonical GP, the variation operations are implemented via crossover and mutation. Borrowing ideas from estimation of distribution algorithms (EDAs) [77], an interesting alternative is to replace such variation operations with the process of sampling from a probability distribution [78–80]. One typical example in this family of GP is probabilistic incremental program evolution (PIPE) [78], where the population is replaced by a hierarchy of probability tables with the tree structure. Although EDA-based GP approaches are appealing, they usually fail to offer significant performance gains over standard tree-based GP [60], which remains to be investigated. It is known that the most widespread type of GP expresses programs as syntax trees, but there are other GP based program synthesizers which use different program representations. PushGP [65] and grammar-guided GP (GGGP) [81] are among the most representative ones. PushGP evolves programs expressed in the Push programming language, which supports different data types by providing a stack for each data type as well as for the code that is executed. Recent studies around PushGP focus on designing new mutation operators [82] and parent selection methods [71,83] in order to further improve its performance. GGGP can either use the derivation tree [84] or a linear genome [69] as its program representation, which can be mapped to a resulting program via the context-free grammar. Recently, some work on GGGP has paid attention to grammar design [70], generalizability [85] and the quality of generated code [86,87].

Stochastic local search. Although stochastic local search (SLS) [37] has been used extensively for combinatorial problems, there is surprisingly little research on SLS for program synthesis. The most notable work in this direction is O'Reilly's PhD thesis [38], where stochastic iterated hill climbing (SIHC) and simulated annealing (SA) were investigated for solving program discovery problems. Her results indicate that SIHC and SA are generally comparable to GP and even sometimes outperform GP. In addition to this work, there are some other research efforts on SLS that target a specific synthesis problem. For example, Nguyen et al. [88] proposed to evolve dispatching rules in job-sop scheduling via iterated local search; Kantor et al. [89] presented simulated annealing using the interaction-transformation representation for symbolic regression.

But all these methods have not (yet) succeeded in bringing into being a comprehensive methodology for program synthesis. So it remains to be seen which technique has the most power and will be adopted by the wider community.

7. Conclusion

In this paper, we have proposed IGI, a new framework for stochastic program synthesis. IGI considers a sequence of program improvements by iteratively searching for modifications to the current reference program. In terms of solution representation, IGI uses a differential code representation that will undergo epochs of evolution under the control of input-output examples. IGI can therefore also be seen as a kind of generative or developmental genetic programming [90,91], which allows the reuse of code and helps to scale up the complexity of evolved programs. Experimental results on two different application domains have demonstrated the clear advantage of IGI over several representative SPS techniques in terms of scalability and solution quality. In addition, we have shown the superiority of IGI as a general program synthesizer over two state-of-the-art general program synthesis systems.

It is promising to incorporate lexicase selection [71] into our IGI framework in the future, to enhance performance. We are also interested to apply the IGI approach to other GP domains such as symbolic regression [92]. Another future research direction is to investigate how to extend IGI to popular general-purpose programming languages, aided by large language models of code [74]. Finally, it would also be interesting to use IGI as the symbolic search engine in neurosymbolic programming [93].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Code is made available on github.

Acknowledgements

Authors gratefully acknowledge support from the Koza Endowment fund to Michigan State University (grant No. RT100614). Runs were done on MSU's iCER HPCC system.

Appendix A. DSL for list manipulation (DSL-LM)

This DSL is introduced in the DeepCoder paper [7], which is inspired by query languages such as SQL. It contains a number of high-level functions that can be used to manipulate integer arrays or integers. In Tables A.11 to A.13 we list all of the functions of DSL-LM and give their symbols, argument types, return types and detailed descriptions. For each benchmark task in this domain, all these functions are available. Note that there are no constants in DSL-LM, and terminals are all from the program's external inputs.

Table A.11
Description of functions in the DSL-LM.

Symbol	Arguments	Return Type	Description
`HEAD`	x: Integer array	Integer	Return the first element of a given array x (or NULL if x is empty).
`LAST`	x: Integer array	Integer	Return the last element of a given array x (or NULL if x is empty).
`TAKE`	n: Integer; x: Integer array	Integer array	Given an integer n and an array x, return the array truncated after the n-th element. (If the length of x is not larger than n, return x without modification.)
`DROP`	n: Integer; x: Integer array	Integer array	Given an integer n and an array x, return the array with the first n elements dropped. (If the length of x is not larger than n, return an empty array.)
`ACCESS`	n: Integer; x: Integer array	Integer	Given an integer n and an array x, return the $(n+1)$-th element of x. (If the length of x is not larger than n, return NULL.)
`MINIMUM`	x: Integer array	Integer	Return the minimum of a given array (or NULL if x is empty).
`MAXIMUM`	x: Integer array	Integer	Return the maximum of a given array (or NULL if x is empty).
`REVERSE`	x: Integer array	Integer array	Return the elements of a given array x in reversed order.
`SORT`	x: Integer array	Integer array	Return the elements of a given array x in non-decreasing order.
`SUM`	x: Integer array	Integer	Return the sum of the elements in a given array x.
`MAPA1`	x: Integer array	Integer array	Each element in a given array x plus 1, and the modified array is returned.

Table A.11 (*continued*)

Symbol	Arguments	Return Type	Description
MAPM1	x: Integer array	Integer array	Each element in a given array x minus 1, and the modified array is returned.
MAPT2	x: Integer array	Integer array	Each element in a given array x is multiplied by 2, and the modified array is returned.
MAPT3	x: Integer array	Integer array	Each element in a given array x is multiplied by 3, and the modified array is returned.
MAPT4	x: Integer array	Integer array	Each element in a given array x is multiplied by 4, and the modified array is returned.
MAPD2	x: Integer array	Integer array	Each element in a given array x divided by 2 (fractions are rounded down), and the modified array is returned.
MAPD3	x: Integer array	Integer array	Each element in a given array x divided by 3 (fractions are rounded down), and the modified array is returned.
MAPD4	x: Integer array	Integer array	Each element in a given array x divided by 4 (fractions are rounded down), and the modified array is returned.

Table A.12
Description of functions in the DSL-LM (*continued*).

Symbol	Arguments	Return Type	Description
MAPV1	x: Integer array	Integer array	Each element in a given array x is multiplied by -1, and the modified array is returned.
MAPP2	x: Integer array	Integer array	Each element in a given array x is multiplied by itself, and the modified array is returned.
FILG0	x: Integer array	Integer array	Return the elements of a given array x that are larger than 0 in their original order.
FILL0	x: Integer array	Integer array	Return the elements of a given array x that are less than 0 in their original order.
FILEV	x: Integer array	Integer array	Return the elements of a given array x that are even in their original order.
FILOD	x: Integer array	Integer array	Return the elements of a given array x that are odd in their original order.
COUG0	x: Integer array	Integer	Return the number of elements in a given array x that is larger than 0.
COUL0	x: Integer array	Integer	Return the number of elements in a given array x that is less than 0.
COUEV	x: Integer array	Integer	Return the number of elements in a given array x that is even.
COUOD	x: Integer array	Integer	Return the number of elements in a given array x that is odd.
ZIPSUM	x: Integer array y: Integer array	Integer array	Return an array z with length n where $z[i] = x[i] + y[i]$, $i = 0, 1, \ldots n-1$ and n is the minimum of the lengths of x and y.
ZIPDIF	x: Integer array y: Integer array	Integer array	Return an array z with length n where $z[i] = x[i] - y[i]$, $i = 0, 1, \ldots n-1$ and n is the minimum of the lengths of x and y.
ZIPMUL	x: Integer array y: Integer array	Integer array	Return an array z with length n where $z[i] = x[i] * y[i]$, $i = 0, 1, \ldots n-1$ and n is the minimum of the lengths of x and y.
ZIPMAX	x: Integer array y: Integer array	Integer array	Return an array z with length n where $z[i] = \max\{x[i], y[i]\}$, $i = 0, 1, \ldots n-1$ and n is the minimum of the lengths of x and y.
ZIPMIN	x: Integer array y: Integer array	Integer array	Return an array z with length n where $z[i] = \min\{x[i], y[i]\}$, $i = 0, 1, \ldots n-1$ and n is the minimum of the lengths of x and y.

Table A.13
Description of functions in the DSL-LM (*continued*).

Symbol	Arguments	Return Type	Description
SCANSUM	x: Integer array	Integer array	Returns an array y of the same length as x and with its content defined by the recurrence: $y[0] = x[0]$, $y[i] = y[i-1] + x[i]$, for $i \geq 1$.
SCANDIF	x: Integer array	Integer array	Returns an array y of the same length as x and with its content defined by the recurrence: $y[0] = x[0]$, $y[i] = y[i-1] - x[i]$, for $i \geq 1$.
SCANMUL	x: Integer array	Integer array	Returns an array y of the same length as x and with its content defined by the recurrence: $y[0] = x[0]$, $y[i] = y[i-1] * x[i]$, for $i \geq 1$.
SCANMAX	x: Integer array	Integer array	Returns an array y of the same length as x and with its content defined by the recurrence: $y[0] = x[0]$, $y[i] = \max\{y[i-1], x[i]\}$, for $i \geq 1$.
SCANMIN	x: Integer array	Integer array	Returns an array y of the same length as x and with its content defined by the recurrence: $y[0] = x[0]$, $y[i] = \min\{y[i-1], x[i]\}$, for $i \geq 1$.

Appendix B. DSL for string transformation (DSL-ST)

This DSL is designed for the PBE-Strings track in the SyGuS competition [58]. Table B.14 describes all of the functions of DSL-ST in detail. Terminals include constants and the program's external inputs. For each task of the SyGuS benchmarks the definition file specifies – besides the input-output examples – what functions in Table B.14 are used and provides some string, integer and Boolean constants.

Table B.14
Description of functions in the DSL-ST.

Symbol	Arguments	Return Type	Description
CAT	s: String t: String	String	Concatenate the strings s and t, and return the combined string.
REP	s: String t: String r: String	String	Return a copy of the string s where the first occurrence of a substring t is replaced with another substring r.
AT	s: String i: Integer	String	Return the string containing a single character at index i in s, i.e., $s[i]$. (If $i < 0$ or i is no smaller than the length of s, return an empty string.)
ITS	x: Integer	String	If integer x is not smaller than 0, convert x into the string and return the string. Otherwise, return an empty string.
SITE	b: Boolean s: String t: String	String	If b is true, return s, otherwise return t.
SUBSTR	s: String i: Integer j: Integer	String	Return the substring of a given string s with the start index i and the end index $\min\{n, i+j\}-1$, where n is the length of s. (If $i < 0$ or $j < 0$ or $i \geq n$, return an empty string.)
ADD	x: Integer y: Integer	Integer	Return the sum of integers x and y.
SUB	x: Integer y: Integer	Integer	Return the difference between integers x and y.
LEN	s: String	Integer	Return the length of a given string s.
STI	s: String	Integer	If all characters in the string s are digits, convert s into the integer and return the integer. Otherwise return -1.
IITE	b: Boolean x: Integer y: Integer	Integer	If b is true, return x, otherwise return y.
IND	s: String t: String i: Integer	Integer	Search the string t in the substring of s that starts at index i and ends at index $n-1$, where n is the length of s, and return the lowest index in s where t is found. (If $i < 0$ or $i \geq n$ or t is not found in s, return -1.)
EQ	x: Integer y: Integer	Boolean	If x equals to y, return true, otherwise return false.
PRF	s: String t: String	Boolean	If s is the prefix of t, return true, otherwise return false.
SUF	s: String t: String	Boolean	If s is the suffix of t, return true, otherwise return false.
CONT	s: String t: String	Boolean	If t is found in s, return true, otherwise return false.

Appendix C. Parameter selection

The key parameters of each algorithm are tuned by performing grid search on two hard synthesis tasks from the list manipulation domain and the string transformation domain respectively. For each parameter combination of an algorithm, we perform 10 independent runs of this algorithm using a timeout of 20 minutes each run on the two tasks respectively. Then for each of the two tasks, we can rank all the parameter combinations of an algorithm according to the average maximum fitness achieved over 10 runs. We select the parameter combination with the lowest average rank over the two tasks. For IGI-SBS, the space of parameters considered is beam width $\in \{25, 50, 100\}$, the number of successors for each patch $\in \{5, 10, 20\}$, the maximum length of patches considered $\in \{2, 3, 4\}$, and tournament size $\in \{2, 5, 8\}$ (81 combinations). For IGI-LGP, the space of parameters considered is population size $\in \{50 * i | i = 1, \ldots, 8\}$, the maximum number of generations in each epoch $\in \{5, 10, 20\}$ and tournament size $\in \{2, 5, 8\}$ (72 combinations). For MH, the space of parameters considered is switch probability $\in \{0.002 * i | i = 1, 2, \ldots, 10\}$ and smoothing constant $\in \{0.1, 0.3, 0.5, 0.7, 0.9\}$ (50 combinations). For GP, the space of parameters considered is population size $\in \{5000, 10000, 20000\}$, crossover probability $\in \{0.8, 0.9, 1.0\}$, mutation probability (per node) $\in \{0.05, 0.08, 0.1\}$ and tournament size $\in \{2, 5, 8\}$ (81 combinations). For SIHC, the space of parameters considered is the maximum number of mutations $\in \{500 * i | i = 1, 2, \ldots, 40\}$ (40 combinations). As for SA, the space of parameters considered is final temperature $\in \{0.0001, 0.0005, 0.001, 0.005, 0.01\}$ and stepsize $\in \{500 * i | i = 1, 2, \ldots 10\}$ (50 combinations).

The final tuned parameters for all the considered algorithms are shown in Tables C.15, C.16, C.17, C.18, C.19 and C.20, respectively. These parameters are used throughout our experiments.

Table C.15
Parameter setting of IGI-SBS.

Parameter	Value
Beam width (B)	50
Number of successors of each patch (C)	5
Maximum length of patches considered ($L_{\max}$)	3
Tournament size	2
Number of initial programs (K)	$B \times C \times L_{\max}$
Number of perturbations (M)	200
Minimum perturbation strength ($S_{\min}$)	4
Minimum depth of initial programs ($d_{\min}$)	2
Maximum depth of initial programs ($d_{\max}$)	4

Table C.16
Parameter setting of IGI-LGP.

Parameter	Value
Population size (N)	100
Maximum generations (G)	5
Tournament size	2
Crossover probability	1.0
Mutation probability	1.0
Number of initial programs (K)	$N \times G$
Number of perturbations (M)	200
Minimum perturbation strength ($S_{\min}$)	4
Minimum depth of initial programs ($d_{\min}$)	2
Maximum depth of initial programs ($d_{\max}$)	4

Table C.17
Parameter setting of MH.

Parameter	Value
Switch probability (p_m)	0.006
Smoothing constant (β)	0.7
Maximum allowed depth of programs	30

Table C.18
Parameter setting of GP.

Parameter	Value
Population size	20000
Crossover probability	0.9
Mutation probability (per node)	0.1
Tournament size	2
Minimum depth of initial programs ($d_{\min}$)	2
Maximum depth of initial programs ($d_{\max}$)	4
Maximum allowed depth of programs	30

Table C.19
Parameter setting of SIHC.

Parameter	Value
Maximum number of mutations ($T_{\max}$)	500
Minimum depth of initial programs ($d_{\min}$)	2
Maximum depth of initial programs ($d_{\max}$)	4
Maximum allowed depth of programs	30

Table C.20
Parameter setting of SA.

Parameter	Value
Starting temperature	1.5
Final temperature	0.001
Stepsize	500
Minimum depth of initial programs ($d_{\min}$)	2
Maximum depth of initial programs ($d_{\max}$)	4
Maximum allowed depth of programs	30

Appendix D. Descriptions of general program synthesis benchmarks

The general program synthesis benchmark problems used in our experiments are described as follows:

1. **Number IO**: Given an integer and a float, return their sum.
2. **Small or Large**: Given an integer n, return a string "small" if $n < 1000$ and a string "large" if $n \geq 2000$ (and nothing if $1000 \leq n < 2000$).
3. **Compare String Lengths**: Given three strings $s1$, $s2$, and $s3$, return true if $length(s1) < length(s2) < length(s3)$, and false otherwise.
4. **Last Index of Zero**: Given a vector of integers, at least one of which is 0, return the index of the last occurrence of 0 in the vector.
5. **Mirror Image**: Given two vectors of integers, return true if one vector is the reverse of the other, and false otherwise.
6. **Sum of Squares**: Given an integer n, return the sum of squaring each integer in the range $[1, n]$.
7. **Grade**: Given 5 integers, the first four represent the lower numeric thresholds for achieving an A, B, C, and D, and will be distinct and in descending order. The fifth represents the student's numeric grade. The program must return the grade X for a student, where X is A, B, C, D, or F depending on the thresholds and the numeric grade.
8. **Median**: Given 3 integers, return their median.
9. **Smallest**: Given 4 integers, return the smallest of them.
10. **Syllables**: Given a string containing symbols, spaces, digits, and lowercase letters, return the number of occurrences of vowels (a, e, i, o, u, y) in this string.

References

[1] S. Gulwani, O. Polozov, R. Singh, et al., Program synthesis, Found. Trends® Program. Lang. 4 (2017) 1–119.
[2] Z. Manna, R. Waldinger, A deductive approach to program synthesis, ACM Trans. Program. Lang. Syst. 2 (1980) 90–121.
[3] S. Srivastava, S. Gulwani, J.S. Foster, From program verification to program synthesis, in: Proceedings of the 37th Annual Symposium on Principles of Programming Languages, 2010, pp. 313–326.
[4] M.A. Bauer, Programming by examples, Artif. Intell. 12 (1979) 1–21.
[5] S. Gulwani, Automating string processing in spreadsheets using input-output examples, ACM SIGPLAN Not. 46 (2011) 317–330.
[6] S. Gulwani, Programming by examples: applications, algorithms, and ambiguity resolution, in: International Joint Conference on Automated Reasoning, Springer, 2016, pp. 9–14.
[7] M. Balog, A. Gaunt, M. Brockschmidt, S. Nowozin, D. Tarlow, DeepCoder: learning to write programs, in: The 5th International Conference on Learning Representations, 2017.
[8] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. Roy, Program synthesis using natural language, in: Proceedings of the 38th International Conference on Software Engineering, ICSE, 2016, pp. 345–356.
[9] N. Yaghmazadeh, Y. Wang, I. Dillig, T. Dillig, Sqlizer: query synthesis from natural language, Proc. ACM Program. Lang. 1 (2017) 1–26.
[10] E.C. Shin, M. Allamanis, M. Brockschmidt, A. Polozov, Program synthesis and semantic parsing with learned code idioms, Adv. Neural Inf. Process. Syst. 32 (2019) 10825–10835.
[11] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, P. Kohli, RobustFill: neural program learning under noisy I/O, in: International Conference on Machine Learning, PMLR, 2017, pp. 990–998.
[12] X. Chen, P. Maniatis, R. Singh, C. Sutton, H. Dai, M. Lin, D. Zhou, Spreadsheetcoder: formula prediction from semi-structured context, in: International Conference on Machine Learning, PMLR, 2021, pp. 1661–1672.
[13] C. Le Goues, M. Dewey-Vogt, S. Forrest, W. Weimer, A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each, in: The 34th International Conference on Software Engineering, ICSE, IEEE, 2012, pp. 3–13.
[14] Y. Yuan, W. Banzhaf Arja, Automated repair of Java programs via multi-objective genetic programming, IEEE Trans. Softw. Eng. 46 (2020) 1040–1067.
[15] S. Gulwani, V.A. Korthikanti, A. Tiwari, Synthesizing geometry constructions, ACM SIGPLAN Not. 46 (2011) 50–61.
[16] K. Ellis, D. Ritchie, A. Solar-Lezama, J.B. Tenenbaum, Learning to infer graphics programs from hand-drawn images, in: Proceedings of the 32nd International Conference on Neural Information Processing Systems, 2018, pp. 6062–6071.
[17] K. Ellis, A. Solar-Lezama, J. Tenenbaum, Unsupervised learning by program synthesis, Adv. Neural Inf. Process. Syst. 28 (2015) 973–981.
[18] D.K. Trivedi, J. Zhang, S.-H. Sun, J.J. Lim, Learning to synthesize programs as interpretable and generalizable policies, Adv. Neural Inf. Process. Syst. 34 (2021).
[19] R. Alur, R. Singh, D. Fisman, A. Solar-Lezama, Search-based program synthesis, Commun. ACM 61 (2018) 84–93.
[20] R. Alur, A. Radhakrishna, A. Udupa, Scaling enumerative program synthesis via divide and conquer, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2017, pp. 319–336.
[21] W. Lee, K. Heo, R. Alur, M. Naik, Accelerating search-based program synthesis using learned probabilistic models, ACM SIGPLAN Not. 53 (2018) 436–449.
[22] A. Odena, K. Shi, D. Bieber, R. Singh, C. Sutton, H. Dai Bustle, Bottom-up program synthesis through learning-guided exploration, in: International Conference on Learning Representations, 2021.
[23] K. Huang, X. Qiu, P. Shen, Y. Wang, Reconciling enumerative and deductive program synthesis, in: Proceedings of the 41st Conference on Programming Language Design and Implementation, 2020, pp. 1159–1174.
[24] A. Solar-Lezama, Program Synthesis by Sketching, University of California, Berkeley, 2008.
[25] S. Jha, S. Gulwani, S.A. Seshia, A. Tiwari, Oracle-guided component-based program synthesis, in: 2010 ACM/IEEE 32nd International Conference on Software Engineering, vol. 1, IEEE, 2010, pp. 215–224.
[26] Y. Feng, R. Martins, O. Bastani, I. Dillig, Program synthesis using conflict-driven learning, ACM SIGPLAN Not. 53 (2018) 420–435.
[27] R. Alur, D. Fisman, S. Padhi, R. Singh, A. Udupa, Sygus-comp 2018: results and analysis, arXiv:1904.07146, 2019.
[28] S. Chib, E. Greenberg, Understanding the Metropolis-Hastings algorithm, Am. Stat. 49 (1995) 327–335.
[29] J.R. Koza, J.R. Koza, Genetic Programming: on the Programming of Computers by Means of Natural Selection, vol. 1, MIT Press, 1992.
[30] W. Banzhaf, P. Nordin, R.E. Keller, F.D. Francone, Genetic Programming: an Introduction, Morgan Kaufmann Publishers Inc., 1998.
[31] S. Gulwani, Technical perspective: program synthesis using stochastic techniques, Commun. ACM 59 (2016) 113.
[32] R. Alur, R. Bodik, G. Juniwal, M.M. Martin, M. Raghothaman, S.A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, A. Udupa, Syntax-guided synthesis, in: Formal Methods in Computer–Aided Design, 2013, pp. 1–8.

[33] Lecture 5: Inductive Synthesis with Stochastic Search, https://people.csail.mit.edu/asolar/SynthesisCourse/Lecture5.htm, 2018. (Accessed 1 January 2022).
[34] M. O'Neill, L. Spector, Automatic programming: the open issue?, Genet. Program. Evol. Mach. (2019) 1–12.
[35] S. Gustafson, A. Ekart, E. Burke, G. Kendall, Problem difficulty and code growth in genetic programming, Genet. Program. Evol. Mach. 5 (2004) 271–290.
[36] S. Luke, L. Panait, A comparison of bloat control methods for genetic programming, Evol. Comput. 14 (2006) 309–344.
[37] H.H. Hoos, T. Stützle, Stochastic Local Search: Foundations and Applications, Elsevier, 2004.
[38] U.-M. O'Reilly, An analysis of genetic programming, Ph.D. thesis, Carleton University, 1995.
[39] E. Schulte, Z.P. Fry, E. Fast, W. Weimer, S. Forrest, Software mutational robustness, Genet. Program. Evol. Mach. 15 (2014) 281–312.
[40] W.B. Langdon, J. Petke, Software is not fragile, in: First Complex Systems Digital Campus World E-Conference 2015, Springer, 2017, pp. 203–211.
[41] V.R. Basil, A.J. Turner, Iterative enhancement: a practical technique for software development, IEEE Trans. Softw. Eng. (1975) 390–396.
[42] W. Banzhaf, Some remarks on code evolution with genetic programming, in: Inspired by Nature, Springer, 2018, pp. 145–156.
[43] J. Petke, S.O. Haraldsson, M. Harman, W.B. Langdon, D.R. White, J.R. Woodward, Genetic improvement of software: a comprehensive survey, IEEE Trans. Evol. Comput. 22 (2018) 415–432.
[44] C. Le Goues, T. Nguyen, S. Forrest, W. Weimer, GenProg: a generic method for automatic software repair, IEEE Trans. Softw. Eng. 38 (2011) 54–72.
[45] Y. Yuan, W. Banzhaf, Toward better evolutionary program repair: an integrated approach, ACM Trans. Softw. Eng. Methodol. 29 (2020) 1–53.
[46] W.B. Langdon, M. Harman, Optimizing existing software with genetic programming, IEEE Trans. Evol. Comput. 19 (2015) 118–135.
[47] B.R. Bruce, J. Petke, M. Harman, E.T. Barr, Approximate oracles and synergy in software energy search spaces, IEEE Trans. Softw. Eng. 45 (2019) 1150–1169.
[48] J.-Y. Liou, X. Wang, S. Forrest, C.-J. Wu, GEVO: GPU code optimization using evolutionary computation, ACM Trans. Archit. Code Optim. 17 (2020) 1–28.
[49] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, A. Scott, SapFix: automated end-to-end repair at scale, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP, IEEE, 2019, pp. 269–278.
[50] S. Zuo, A. Blot, J. Petke, Evaluation of Genetic Improvement Tools for Improvement of Non-functional Properties of Software, 2022.
[51] W.B. Langdon, Genetic improvement of genetic programming, in: 2020 IEEE Congress on Evolutionary Computation, CEC, IEEE, 2020, pp. 1–8.
[52] A.E. Brownlee, J. Petke, B. Alexander, E.T. Barr, M. Wagner, D.R. White, Gin: genetic improvement research made easy, in: Proceedings of the Genetic and Evolutionary Computation Conference, 2019, pp. 985–993.
[53] G. An, A. Blot, J. Petke, S. Yoo, Pyggi 2.0: language independent genetic improvement framework, in: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 1100–1104.
[54] H.R. Lourenço, O.C. Martin, T. Stützle, Iterated local search: framework and applications, in: Handbook of Metaheuristics, Springer, 2019, pp. 129–168.
[55] M.F. Brameier, W. Banzhaf, Linear Genetic Programming, Springer Science & Business Media, 2007.
[56] P. Liskowski, I. Bladek, K. Krawiec, Neuro-guided genetic programming: prioritizing evolutionary search with neural networks, in: Proceedings of the Genetic and Evolutionary Computation Conference, 2018, pp. 1143–1150.
[57] Y. Chen, C. Wang, O. Bastani, I. Dillig, Y. Feng, Program synthesis using deduction-guided reinforcement learning, in: International Conference on Computer Aided Verification, Springer, 2020, pp. 587–610.
[58] Syntax-guided synthesis, https://sygus.org, 2014. (Accessed 1 January 2022).
[59] E. Schkufza, R. Sharma, A. Aiken, Stochastic superoptimization, ACM SIGARCH Comput. Archit. News 41 (2013) 305–316.
[60] R. Poli, W.B. Langdon, N.F. McPhee, J.R. Koza, A Field Guide to Genetic Programming, 2008.
[61] W. Lee, Combining the top-down propagation and bottom-up enumeration for inductive program synthesis, Proc. ACM Program. Lang. 5 (2021) 1–28.
[62] W. Banzhaf, F.D. Francone, P. Nordin, The effect of extensive use of the mutation operator on generalization in genetic programming using sparse data sets, in: International Conference on Parallel Problem Solving from Nature, Springer, 1996, pp. 300–309.
[63] E. Pantridge, L. Spector, Pyshgp: Pushgp in python, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, 2017, pp. 1255–1262.
[64] D.J. Montana, Strongly typed genetic programming, Evol. Comput. 3 (1995) 199–230.
[65] L. Spector, A. Robinson, Genetic programming and autoconstructive evolution with the push programming language, Genet. Program. Evol. Mach. 3 (2002) 7–40.
[66] M. Fenton, J. McDermott, D. Fagan, S. Forstenlechner, E. Hemberg, M. O'Neill, PonyGE2: grammatical evolution in python, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, 2017, pp. 1194–1201.
[67] S. Kelly, R.J. Smith, M.I. Heywood, Emergent policy discovery for visual reinforcement learning through tangled program graphs: a tutorial, in: Genetic Programming Theory and Practice XVI, 2019, pp. 37–57.
[68] T. Helmuth, L. Spector, General program synthesis benchmark suite, in: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, 2015, pp. 1039–1046.
[69] M. O'Neill, C. Ryan, Grammatical evolution, IEEE Trans. Evol. Comput. 5 (2001) 349–358.
[70] S. Forstenlechner, D. Fagan, M. Nicolau, M. O'Neill, A grammar design pattern for arbitrary program synthesis problems in genetic programming, in: European Conference on Genetic Programming, Springer, 2017, pp. 262–277.
[71] T. Helmuth, N.F. McPhee, L. Spector, Lexicase selection for program synthesis: a diversity analysis, in: Genetic Programming Theory and Practice XIII, Springer, 2016, pp. 151–167.
[72] E. Parisotto, A.-r. Mohamed, R. Singh, L. Li, D. Zhou, P. Kohli, Neuro-symbolic program synthesis, in: International Conference on Learning Representations, 2017.
[73] J. Hong, D. Dohan, R. Singh, C. Sutton, M. Zaheer, Latent programmer: discrete latent codes for program synthesis, in: International Conference on Machine Learning, PMLR, 2021, pp. 4308–4318.
[74] M. Chen, J. Tworek, H. Jun, Q. Yuan, H.P.d.O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., Evaluating large language models trained on code, arXiv preprint, arXiv:2107.03374, 2021.
[75] R. Bunel, A. Desmaison, M.P. Kumar, P.H. Torr, P. Kohli, Learning to superoptimize programs, in: International Conference on Learning Representations, 2016.
[76] J.R. Koenig, O. Padon, A. Aiken, Adaptive restarts for stochastic synthesis, in: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2021, pp. 696–709.
[77] P. Larrañaga, J.A. Lozano, Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation, vol. 2, Springer Science & Business Media, 2001.
[78] R. Salustowicz, J. Schmidhuber, Probabilistic incremental program evolution, Evol. Comput. 5 (1997) 123–141.
[79] K. Sastry, D.E. Goldberg, Probabilistic model building and competent genetic programming, in: Genetic Programming Theory and Practice, Springer, 2003, pp. 205–220.
[80] K. Yanai, H. Iba, Estimation of distribution programming based on Bayesian network, in: IEEE Congress on Evolutionary Computation, vol. 3, IEEE, 2003, pp. 1618–1625.
[81] R.I. McKay, N.X. Hoai, P.A. Whigham, Y. Shan, M. O'neill, Grammar-based genetic programming: a survey, Genet. Program. Evol. Mach. 11 (2010) 365–396.

[82] T. Helmuth, N.F. McPhee, L. Spector, Program synthesis using uniform mutation by addition and deletion, in: Proceedings of the Genetic and Evolutionary Computation Conference, 2018, pp. 1127–1134.
[83] T. Helmuth, A. Abdelhady, Benchmarking parent selection for program synthesis by genetic programming, in: Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, 2020, pp. 237–238.
[84] P.A. Whigham, et al., Grammatically-based genetic programming, in: Proceedings of the Workshop on Genetic Programming: from Theory to Real-World Applications, vol. 16, Citeseer, 1995, pp. 33–41.
[85] D. Sobania, On the generalizability of programs synthesized by grammar-guided genetic programming, in: European Conference on Genetic Programming (Part of EvoStar), Springer, 2021, pp. 130–145.
[86] D. Sobania, F. Rothlauf, Teaching gp to program like a human software developer: using perplexity pressure to guide program synthesis approaches, in: Proceedings of the Genetic and Evolutionary Computation Conference, 2019, pp. 1065–1074.
[87] E. Hemberg, J. Kelly, U.-M. O'Reilly, On domain knowledge and novelty to improve program synthesis performance with grammatical evolution, in: Proceedings of the Genetic and Evolutionary Computation Conference, 2019, pp. 1039–1046.
[88] S. Nguyen, M. Zhang, M. Johnston, K.C. Tan, Automatic programming via iterated local search for dynamic job shop scheduling, IEEE Trans. Cybern. 45 (2014) 1–14.
[89] D. Kantor, F.J. Von Zuben, F.O. de Franca, Simulated annealing for symbolic regression, in: Proceedings of the Genetic and Evolutionary Computation Conference, 2021, pp. 592–599.
[90] J.R. Koza, Human-competitive results produced by genetic programming, Genet. Program. Evol. Mach. 11 (2010) 251–284.
[91] A.E. Eiben, J. Smith, From evolutionary computation to the evolution of things, Nature 521 (2015) 476–482.
[92] P. Orzechowski, W. La Cava, J.H. Moore, Where are we now? A large benchmark study of recent symbolic regression methods, in: Proceedings of the Genetic and Evolutionary Computation Conference, 2018, pp. 1183–1190.
[93] S. Chaudhuri, K. Ellis, O. Polozov, R. Singh, A. Solar-Lezama, Y. Yue, et al., Neurosymbolic programming, Found. Trends® Program. Lang. 7 (2021) 158–243.