Genetic Algorithm for Evolving an Objective Function of a Program Synthesizer

Nikolaos Efthymiou
Supervisor: Sebastijan Dumančić
EEMCS, Delft University of Technology, The Netherlands
22-6-2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

Program Synthesis is a challenging problem in Artificial Intelligence. An important element of a program synthesizer is the objective function that guides the combinatorial search for a program that satisfies a given user intent. Given multiple I/O example transformations that correspond to the intended behavior of the program, this function evaluates the performance of a generated program based on the distance of its output to the correct output. In this study, we consider the possibility of using a Genetic algorithm for the evolution of such a function as a means to partially automate the design process. In particular, we propose the *GeneticObjective* algorithm that evolves domain-specific objective functions by combining user-defined *local distance functions* in an algebraic expression. Using the Brute synthesizer, we conducted experiments in the Robot Planning and the String transformation domains, the results of which showed that such an approach evolves informative functions. The best evolved function reached the effectiveness of a manually-designed function in the Robot domain, while it outperformed the effectiveness of a hand-crafted function in the String domain.

1 Introduction

The problem of Program Synthesis (PS) is a long-standing challenge in Artificial Intelligence [1]. It essentially concerns combinatorial search over a space of programs to find one that satisfies a given user specification [2, 3]. The importance of this problem is evident, especially concerning the automation of repetitive manual tasks [4–6]. Building strong AI seems to require the ability to automatically synthesize universal computation [7]. Apart from the evident importance of PS in Theorem Proving and Software Development [8], there are also applications in other fields, including Robotics and Biology [5]. Some of the methods developed for solving the PS problem have also been applied in *Information Extraction* for the problem of *Rule Learning* [9]. Related problems to PS have also emerged throughout the years. Relaxing the PS problem by only requiring the synthesis of the output instead of the mapping/program itself leads to the *Program Induction* problem, which has gained a lot of attention [10, 11]. *Algorithm synthesis* is a new branch of PS, where a class of predefined algorithms is applied to a program given by the user. A synthesizer of divide and conquer algorithms is described in [12].

We can identify three integral elements of PS: the *specification*, which describes the intended behavior of the program, the *program space* that is defined via a programming language and the *search technique* [13]. In this study, we consider the case in which the specification is given in the form of input/output examples and the program space is implicitly defined by a Domain-Specific Language.

One key aspect of a program synthesizer is the objective function that is used to guide the search. Using appropriate such functions enhances the effectiveness of the search techniques. The objective function can be thought of as a distance metric between the output of a given program and the intended output, although more aspects might be considered (e.g. the complexity of the generated program). Typically, such functions are handcrafted and domain knowledge is essential. This manual process is not only time-consuming but also error-prone, since designing informative domain-specific distance functions is a challenging open problem [5]. Identifying the important "features" that can define a distance metric is far from trivial.

In this work, we suggest a method that utilizes a Genetic Algorithm, which we call *GeneticObjective*, to partly automate the design of such an objective function. Genetic Algorithm (GA) is a well-known population-based metaheuristic algorithm proposed by John Holland [14]. The algorithm mimics in some way the biological evolution process of natural selection (Darwinism). Solutions are encoded as chromosomes that are evolved over generations by means of various genetic operators. Each chromosome is comprised of indivisible units, called genes [14]. GA belongs to a broad class of algorithms that are studied in the sub-field of Artificial Intelligence known as Evolutionary Computation. GAs have found applications in many fields including networking (e.g. a distributed version of the *Bandwidth allocation* algorithm), operations research (e.g. *Single Row Facility Layout*) and medical imaging (e.g. MRI-based *brain tumour segmentation*) [15–17].

In our study, the Genetic Algorithm we suggest takes as input several user-defined domain-specific *local distance functions* (genes), which correspond to distance metrics between the output of two programs for specific features of the domain. Formulating several such functions is less complicated than designing a complete distance function that leads to an informed search of the program space. The aim of this study is to show that implementing a Genetic Algorithm that evolves an objective function of a synthesizer is feasible and also to measure the quality of the evolved function by testing the efficacy of the synthesizer when using the latter. More specifically, we pose the following research question:

Question 1. *How effective is a program synthesizer using an objective function that is evolved by means of a Genetic Algorithm?*

In order to answer question 1, we will run the *Brute* synthesizer [18] on two domains: *Robot Planning* and *String transformation* [19, 20]. The use of Genetic Algorithms for such a goal is a novel approach that is expected to have an essential contribution to solving the Program Synthesis problem.

2 Background

2.1 Program Synthesis

Program Synthesis (PS) is a problem lying in the intersection of Artificial Intelligence, Programming Languages and Formal Methods [21–23]. In its most general form, the PS problem can be described as follows:

Problem 1 (PS problem)**.** *Given a programming language that defines the program space and a user intent/specification, find a program that satisfies this specification.*

Given the description of the PS problem, three key components can be identified: the *user intent*, the *search space* and the *search technique* [13]. Changing the way in which any of these components is expressed or defined leads to different formulations of the problem and therefore to different ways of tackling it.

User intent

The user intent captures the behavior of the desired program and can be expressed in various ways. Natural language, logical specification and input-output examples are three realizable mechanisms for a user to describe their specification [4]. The last one is one of the most common frameworks for the PS problem and it is usually referred to as the *Programming By Example* (PBE) paradigm [24, 25]. The given input/output examples correspond to the intended behavior (output) of a program. For instance, in the *Robot planning* domain, the user provides a set of pairs consisting of the initial and the target configurations of the grid (initial and target positions of the robot and the ball) [19]. The facileness of expressing the user intent in PBE is one of the main reasons for its wide adoption [26, 27].

Search space

The search space realizes the trade-off between efficiency and expressiveness. Considering typical high-level languages (e.g. Java) increases the size of the program space and enables solving more problems, but makes the search computationally expensive. So, a *Domain Specific Language* (DSL) is often designed for the domain at hand to provide an adequate level of expressiveness, while bypassing the intractability of the combinatorial search. An example of a DSL for transforming strings on a syntactic level can be found in [28]. In the case of the *Robot planning* domain, the tokens that are used for the construction of programs include tokens for moving the robot (*moveUp*, *moveDown*, *moveLeft*, *moveRight*) and for dropping and picking up the ball. As for the *String transformation* domain, there are tokens for moving the cursor, for converting a symbol to upper/lower case and for dropping a symbol [18].

Search technique

The search procedure specifies the way in which the search (program) space is explored in order to find a desired program; *Enumerative* and *Deductive* search are two classical approaches. The former is essentially a bottom-up search procedure that enumerates programs, while pruning to make the search efficient [29]. The latter works in a top-down fashion: a chain of deductive rules symbolically reduces the user intent to a correct program [30]. Applying transformation rules within a deductive system is fundamentally a *Theorem Proving* approach [31]. During the past years, techniques from the advancing field of *Machine Learning* have been utilized to ameliorate the classical approaches [32], with Neural architectures being recurrently used in the context of PS [11, 33–36]. The Brute program synthesizer, which we use in our experiments, performs a best-first search using a specific objective function [18].

2.2 Objective function of a program synthesizer

One aspect that is common to most of the search techniques discussed above is the objective function that is used to measure the performance of a generated program. This function guides the search and thus it is a key factor in the efficiency and the effectiveness of a synthesizer. A common terminology that has been used in the literature is the so-called *ranking* function, which approximates the likelihood of a given program to be the desired program [5, 13]. It is important to note that there may exist multiple programs that satisfy a given specification and in this case the ranking function can be used in the *disambiguation* process [5]. Machine Learning approaches have been used in the variant of PS in which a vast amount of synthesized programs are provided as input and the ranking function shall predict a correct program [37].

It is a hard open problem to define appropriate functions, since domain-specific knowledge and substantial manual effort are usually required [38–40]. Identifying the key features of a domain that would lead to an informative objective (distance) function is a laborious task. Furthermore, it is error-prone and in some cases these functions may lack robustness to modifications in the underlying DSL. As a general rule, the performance on the given examples and some complexity metric of the generated program can be considered when designing the objective function of a synthesizer [41–45]. Manually-designed objective functions for the Robot and the String domains are presented in [46] and [47] respectively.

2.3 Genetic Algorithms for the evolution of objective functions

The elements of GA in its simplest form are the following: *chromosome representation*, *fitness function*, *selection*, *mutation* and *crossover*. Each chromosome encodes a feasible solution (assuming a meaningful and accurate encoding) to the problem that GA is trying to solve. The fitness function is used as a measure of the quality of each individual chromosome. Usually, this is the objective function of the underlying optimization problem. Selection is the phase in which individuals are chosen for later breeding. Mutation concerns a random perturbation (with probability p_m) of the individuals in the current iteration (population). Crossover (with probability p_c) is analogous to the biological process of sexual reproduction, which involves the recombination of the genetic information of a pair of parents that leads to the creation of new offspring. In essence, the manipulation of chromosomes using the biologically-inspired mechanisms mentioned above increases on average the fitness of the individuals (chromosomes) over generations.

A generic version of GA is given in algorithm 1. $\mathcal{S}$ denotes the solution space and $F : \mathcal{S} \to \mathbb{R}$, $M : \mathcal{S} \to \mathcal{S}$, $C : \mathcal{S}^2 \to \mathcal{S}^2$ and $P : 2^{\mathcal{S}} \to 2^{\mathcal{S}^2}$ denote the fitness, mutation, crossover and selection functions respectively.

GA has been shown to be a powerful optimizer in many occasions. It has been used for solving various *NP-hard* (intractable) problems such as *Modular Exam Scheduling*, *Channel routing* in VLSI design and *Graphical Steiner Tree* in Graph Theory [48–50]. It is important to note that in the context of Metaheuristics, such as GA, "solving" means

Algorithm 1 GenericGA

1: Construct initial population $X \subseteq \mathcal{S}$
2: best $\leftarrow$ NULL
3: **while** termination condition is not met **do**
4:     Selected $\leftarrow$ P(X)
5:     X $\leftarrow \emptyset$
6:     **for** $(x, y) \in$ Selected **do**
7:         $(x', y') \leftarrow \begin{cases} C((x,y)), & \text{if } U(0,1) \leq p_c \\ (x, y), & \text{otherwise} \end{cases}$
8:         $x'' \leftarrow \begin{cases} M(x'), & \text{if } U(0,1) \leq p_m \\ x', & \text{otherwise} \end{cases}$
9:         $y'' \leftarrow \begin{cases} M(y'), & \text{if } U(0,1) \leq p_m \\ y', & \text{otherwise} \end{cases}$
10:        Add x'', y'' to X.
11:    **end for**
12:    best $\leftarrow$ argmax(F(best), $\max_{x \in X} F(x)$)
13: **end while**
14: **return** best

finding near-optimal solutions. Genetic Algorithms have been used to evolve functions, such as mating selection fitness functions for the *Circle Packing in Squares* problem and similarity functions for clustering [51, 52]. The special case of GA in which each chromosome represents a computer program is known as Genetic Programming (GP). GP is a natural way to solve the PS problem and in fact this approach was presented in the seminal work of Koza [53].

The literature review above highlights that the objective function used by a program synthesizer highly affects the efficacy of the synthesis. In addition, coming up with an insightful and informative such function is a challenging endeavor and domain experts, even after some contemplation, can be fallible in this matter. Given the power of GA, we suggest using it for evolving an objective function of a program synthesizer.

3 Methodology

Our method is based on the idea that objective functions (represented as algebraic expressions) involving some domain-specific "local" distance functions are evolved with a GA that we call *GeneticObjective*. The fitness of an individual function f is computed as a function of the efficiency and the effectiveness of *Brute* when the latter is using f as an objective function.

3.1 Mathematical notation

For a set A, 2^A denotes the power set of A. For a set of symbols S, S^* denotes the Kleene star on S, i.e. $S^* = \bigcup_{i \geq 0} S^i$. For $n \in \mathbb{N}, [n] = \{1, ..., n\}$. For two sets A, B, $A \setminus B$ denotes the set difference and $A^B = \{f | f : B \to A\}$. From asymptotic analysis, we are using $\sim$ as the asymptotic equality symbol. $\binom{n}{k}$ denotes the binomial coefficient. For a rooted tree T, $h(T)$ denotes the height of the tree, which is defined as the length of a longest path from the root of T to a leaf. For a subtree S of T, $d(S, T)$ denotes the depth of S in T,

which is defined as the distance between the roots of the two trees. In the context of probability theory, $X \sim Y$ denotes equality in law. $U(a, b)$ denotes the uniform distribution in $[a, b]$ and $U\{a, b\}$ denotes the discrete uniform distribution in $\{a, a + 1, ..., b\}$. $Ber(p)$ denotes a Bernoulli random variable with parameter p. $x \in_R X$ denotes a uniform random selection from a countable set X. We denote the Manhattan distance with L^1. abs denotes the absolute value function, $|s|$ denotes the length of a string (sequence of characters) s and $|s \cap s'|$ denotes the number of distinct matching characters between s and s'.

3.2 Problem formulation

Program Synthesis domain and examples

Let $\mathcal{D}$ denote the set of all possible domains. For our purposes, we can think of a 1-to-1 correspondence between domains and DSLs. The Robot domain is defined uniquely by some DSL that contains tokens for manipulating/processing input grids. This domain involves a robot and a ball on a grid. The intended program transforms an initial configuration of the grid to a target configuration by moving the robot in order to place the ball on a target cell and then reach a target position [19]. In the String transformation domain, a string is given together with a cursor pointing to a specific position of the string and the goal is to transform the string to a target string, by moving the cursor, converting characters to lower/upper case and dropping characters [18]. For each domain D, let I_D and O_D denote the input and output domains of D respectively, i.e. a program written in the DSL of D maps an element of I_D to an element of O_D through a sequence of transformations. For the Robot domain, we have that $I_D = O_D = \mathbb{N}^5 \times \{0, 1\}$: grid's size s, robot's position (coordinates) p_r, ball's position (coordinates) p_b, holding flag h. We denote with A the set of all alphanumeric symbols. For the string domain, $I_D = O_D = A^* \times \mathbb{N}$ (string s, cursor's position i).

We denote a set of examples for a domain $D \in \mathcal{D}$ with:

$$\mathcal{E}_D = \{e_{ij}\} \subseteq I_D \times O_D, i \in I \subseteq [t_D], j \in J \subseteq [n_{D_i}] \quad (1)$$

, where t_D denotes the number of tasks of domain D and n_{D_i} denotes the number of examples of task i in domain D. Each $e = (x, y) \in \mathcal{E}$ is a pair of an input x and the correct output y for x (PBE paradigm).

Local distance functions

GeneticObjective constructs a domain-specific objective (distance) function, which is composed of finitely many manually-crafted *local distance functions* $\mathcal{F}_D \subseteq \mathbb{R}_{\geq 0}^{O_D^2}$. We think of each function in this set as a distinct distance metric between a given program's output y and the correct output y^* for selected features of the domain. Each function in $\mathcal{F}_{\text{robot}}$ takes as input two 5-tuples $y = (s, r_x, r_y, b_x, b_y, h)$ and $y^* = (s^*, r_x^*, r_y^*, b_x^*, b_y^*, h^*)$, where $(r_x, r_y) = p_r, (b_x, b_y) = p_b, (r_x^*, r_y^*) = p_r^*$ and $(b_x^*, b_y^*) = p_b^*$. Taking into account the components that are used in the manually-designed objective function in the Robot domain [46], we suggest the subsequent

local distance functions to form $\mathcal{F}_{\text{robot}}$:

$$\mathcal{F}_{\text{robot}} = \begin{cases} L^1(p_r, p_b^*), \\ L^1(p_r, p_r^*), \\ L^1(p_b^*, p_r^*), \\ L^1(p_r, p_b), \\ L^1(p_b, p_b^*) \end{cases} \tag{2}$$

Each function in $\mathcal{F}_{\text{string}}$ takes as input two 2-tuples $y = (s, i)$ and $y^* = (s^*, i^*)$. $\mathcal{F}_{\text{string}}$ is defined as follows:

$$\mathcal{F}_{\text{string}} = \begin{cases} abs(|s| - |s^*|), \\ min(|s|, |s'|) - |s \cap s^*|, \\ abs(i - i^*), \\ abs(cU(s) - cU(s^*)), \\ abs(cL(s) - cL(s^*)) \end{cases} \tag{3}$$

, where $cU(s)$ and $cL(s)$ compute the number of upper and lower case symbols respectively. *GeneticObjective* requires the existence of $\mathcal{F}_D$ and this is the sole task that has to be performed manually. *GeneticObjective* combines the "important" functions of $\mathcal{F}_D$ in an algebraic expression. Let $\mathcal{T}_C^O$ denote the set of all correctly parenthesized algebraic expressions over the set of operators O and the set of terms C. We set $O = \{+, -, \cdot, /\}$. In the context of a domain $D \in \mathcal{D}$, we think of these algebraic expressions as domain-specific objective functions, which take as input a program's output $y \in O_D$ and the correct output $y^* \in O_D$ and evaluate a distance between the two (the absolute value is used to ensure non-negativity, a 0 distance is considered when $y = y^*$ and an infinite distance is considered when dividing by 0). For the Robot domain, we set $\mathcal{T}_C^O = \mathcal{T}_{\mathcal{F}_{\text{robot}}}^O$ and analogously for the string domain.

Evaluating *Brute* for a given objective function

The evaluation criteria for assessing the quality of Brute are: its running time, the percentage of solved tasks and the percentage of solved examples (a task is called solved if Brute managed to successfully solve all of its examples). Consider a domain $D \in \mathcal{D}$. We define a function $\mathcal{V} : \mathcal{T}_{\mathcal{F}_D}^O \times 2^{I_D \times O_D} \to [0, 1]$, which evaluates the efficiency and the effectiveness of Brute on a given set of examples $\mathcal{E}_D$ (using the notation from 1) while using a given objective function to guide its search. $\mathcal{V}$ is used as the fitness function of *GeneticObjective*. Thus, as a measure of quality of an objective function f we use the efficacy of Brute when employing f during the search. Let S denote the percentage of solved tasks and U the average percentage of unsolved examples over the unsolved tasks. Finally, R denotes the average normalized runtime of Brute over the examples (ratio between the runtime of Brute and the given timeout). $\mathcal{V}$ is defined (equation 4) as a convex combination of the three components described above, adjusted such that it can be used as the objective of a maximization problem.

$$\mathcal{V}(T, \mathcal{E}_D) = w_1 S + w_2(1 - U) + w_3(1 - R) \tag{4}$$

We note that for the Robot domain, there is only a single task and so the fitness function only has a component cor-responding to the percentage of examples solved and a component corresponding to the average normalized running time of Brute.

***Domain-Specific Objective* problem**

We can now define the *Domain-Specific Objective* (DSO) problem that *GeneticObjective* is trying to solve as an optimization (maximization) problem. More specifically, we think of $\mathcal{V}$ as a black-box function (it runs the Brute procedure) and thus the problem is an example of *Unconstrained Discrete Black-Box Optimization*:

Problem 2 (DSO). *Given a set of examples $\mathcal{E}_D$, the black-box function $\mathcal{V}$ and $\mathcal{T}_{\mathcal{F}_D}^O$, find $\mathrm{argmax}_{T \in \mathcal{T}_{\mathcal{F}_D}^O} \mathcal{V}(T, \mathcal{E}_D)$.*

It is clear that $\mathcal{T}_{\mathcal{F}_D}^O$ is a countably infinite set and thus we cannot solve problem 2 exhaustively. For our purposes, we are not interested in expressions containing an arbitrarily large number of operators, since these are computationally impractical to work with. We now argue that even if the number of operators in an expression is upper-bounded by B, we cannot hope for a brute-force approach, since the search space is prohibitively large. We show the asymptotic behaviour of $|\mathcal{T}_{\mathcal{F}_D}^O|$, in the case in which every expression contains at most B operators. It is known from Combinatorics that the number of parenthesizations of an arithmetic expression with n operators is given by the *Catalan number* $C_n = \frac{1}{n+1}\binom{2n}{n}$ [54]. With a straightforward combinatorial argument, we can observe that:

$$|\mathcal{T}_{\mathcal{F}_D}^O| = \sum_{n=0}^{B} C_n |O|^n |\mathcal{F}_D|^{n+1} \tag{5}$$

We note that even though the associativity of addition implies the equivalence of the expressions $(\square + \square) + \square$ and $\square + (\square + \square)$, these are considered two distinct expressions (parenthesization is different). Using Stirling's approximation formula for $n!$, we observe that $C_B \sim 4^B B^{-\frac{3}{2}} \pi^{-\frac{1}{2}}$. By only considering the last term of the sum in Equation 5, it easily follows that $|\mathcal{T}_{\mathcal{F}_D}^O| \sim (4|O||\mathcal{F}_D|)^B B^{-\frac{3}{2}} \pi^{-\frac{1}{2}}$.

3.3 *GeneticObjective*

Following the notation of Algorithm 1, we set $\mathcal{S} = \mathcal{T}_{\mathcal{F}_D}^O$ and in the following subsections we define: a) the chromosome encoding, b) the construction of the initial population and c) functions M, C, P. The termination condition is met simply by reaching a given number of generations (iterations). We set $F = \mathcal{V}$.

Chromosome encoding

Each chromosome represents an element of $\mathcal{T}_{\mathcal{F}_D}^O$. The representation has the form of a *binary expression tree* (this representation is unique). A binary expression tree is a full binary tree (we consider only binary operators in the set O), in which the internal nodes store an operator and the leaves store the terms (elements of $\mathcal{F}_D$). In other words, the symbol of each node v (we denote this by v.symbol) is a function symbol. A tree T is evaluated recursively starting from the root (for a similar approach, see [52]). When the tree is used by Brute as a distance function, the absolute value of the evaluation of the tree is used.

Initial population

For creating the initial population, we generate random binary expression trees of a given maximum height H. H implies an upper bound of $2^H - 1$ on the number of operators that are present in a tree. Let $Tree(s, t_{\text{left}}, t_{\text{right}})$ denote the expression that constructs a binary expression tree with symbol s, left child t_{left} and right child t_{right}. A random tree is generated by evaluating $T_{rand}(0, H, p)$ (algorithm 2), where $p \in [0, 1]$ is the probability of placing an operator node.

Algorithm 2 $T_{rand}(d_{cur}, d_{max}, p)$

1: **if** $d_{cur} = d_{max}$ **then**
2:     **return** $Tree(c \in_R \mathcal{F}_D, \emptyset, \emptyset)$
3: **else**
4:     $p_{\text{op}} \leftarrow \begin{cases} 1, & \text{if } d_{cur} = 0 \\ p, & \text{otherwise} \end{cases}$
5:     **if** $Ber(p_{\text{op}}) = 1$ **then**
6:         **return** $Tree(\text{op} \in_R O,$
                                 $T_{rand}(d_{cur} + 1, d_{max}, p),$
                                 $T_{rand}(d_{cur} + 1, d_{max}, p))$
7:     **else**
8:         **return** $Tree(c \in_R \mathcal{F}_D, \emptyset, \emptyset)$
9:     **end if**
10: **end if**

The given construction ensures that $1 \leq h(T_{\text{rand}}(0, H, p)) \leq H$. At every step of the algorithm, except when starting at the root, an operator node is placed with probability p and a term node (a leaf) is placed with probability $1 - p$.

Selection

We implement *deterministic k-tournament selection without replacement* [55, 56]. This procedure samples k individuals from the population without replacement (k is the tournament size) and then the chromosome with the highest fitness value is deterministically chosen as the "winner" of the tournament and thus as a selected parent for reproduction. Note that 1-tournament is equivalent to random selection. We run this procedure in pairs to produce the desired number of parent pairs, by excluding from the second iteration the chromosome chosen as the first parent. This selection method exhibits advantages over other alternatives (e.g. stochastic *Roulette Wheel*). For instance, it does not require the fitness values to be scaled [57]. k-tournament, among other selection methods, has linear time complexity, which is preferable over the linearithmic or even quadratic complexity of other commonly used schemes [55]. A common notion used in many selection schemes is *selection pressure*. Loosely speaking, the higher the selection pressure, the more favored are the individuals with high fitness [58, 59]. Tournament selection is also preferred due to the easy adjustment of the selection pressure, by changing the tournament size.

Random Walk

For describing the mutation and crossover functions, we explain the $RandWalk$ algorithm, which takes as input a tree T and returns a random subtree of T (algorithm 3).

Algorithm 3 $RandWalk(T)$

1: Pick $x \in_R \{0, ..., h(T)\}$
2: Pick $\boldsymbol{v} \in_R \{0, 1\}^x$
3: $S \leftarrow T$, $i \leftarrow 0$
4: **while** $(i < x) \land (deg(S.\text{root}) \neq 1)$ **do**
5:     $S \leftarrow \begin{cases} S.\text{left}, & \text{if } \boldsymbol{v}[i] = 0 \\ S.\text{right}, & \text{otherwise} \end{cases}$
6:     $i \leftarrow i + 1$
7: **end while**
8: **return** S

Line 4 of algorithm 3 reflects the fact that the random walk terminates either at a node (internal or leaf) of depth x or at a leaf of depth smaller than x. The algorithm ensures that $d(RandomWalk(T), T) \sim U\{0, h(T)\}$. In the context of the mutation and crossover operations, by random node or subtree of T we mean $RandWalk(T)$.

Mutation

We describe the mutation function $M : \mathcal{T}^O_{\mathcal{F}_D} \rightarrow \mathcal{T}^O_{\mathcal{F}_D}$ in algorithm 4. This function replaces the symbol of a random node in the given tree with a different random symbol of the same type (operator or term symbol).

Algorithm 4 $M(T)$

1: $S \leftarrow RandWalk(T)$
2: **if** $deg(S.\text{root}) = 1$ **then** $\triangleright$ Leaf node
3:     $S.\text{root.symbol} \leftarrow c \in_R \mathcal{F}_D \setminus \{S.\text{root.symbol}\}$
4: **else**
5:     $S.\text{root.symbol} \leftarrow o \in_R O \setminus \{S.\text{root.symbol}\}$
6: **end if**
7: **return** T

Crossover

The crossover function $C : \mathcal{T}^O_{\mathcal{F}_D}{}^2 \rightarrow \mathcal{T}^O_{\mathcal{F}_D}{}^2$ produces two children from a given pair of trees (T_1, T_2) by exchanging a random subtree of T_1 with a random subtree of T_2.

Elitism

In order to ensure that the sequence of maximum fitness values over generations is non-decreasing, we used *elitism* as part of *GeneticObjective*. This mechanism transfers (copies) some of the fittest individuals of a generation to the next and it has a significant impact on the performance of GA [60, 61]. The number of individuals copied, i.e. the *elite* size, is another way of adjusting the selection pressure. Tuning the *elite* size is highly important as a means to avoid premature convergence [62].

3.4 Evaluation of *GeneticObjective*

The *GeneticObjective* GA attempts to find an objective function T, from a given set of functions, which maximizes the effectiveness/efficiency of Brute, when the latter is using T to guide its search. We expect that a random function T results to a low value of $\mathcal{V}$ with high probability. In other words, we expect that random objectives do not contain task-specific information and lead to an ineffective search procedure. Using

GeneticObjective, a Metaheuristic algorithm, we cannot provide any theoretical guarantee of global optimality. Instead, we analyze the algorithm's performance experimentally, by trying multiple configurations of the parameters of the algorithm. The fitness value of the best function found by the GA is compared to the fitness of a manually-designed objective function for each domain (robot and string).

4 Results and Discussion

A (train) set of examples is used for the fitness function during the execution of the GA (while functions are being evolved) and a different unseen (test) set of examples is used for the fitness evaluation of the best function (individual) found to get an unbiased estimate. This final value is then compared to the fitness value of the manually-designed objective functions of each domain, as suggested by [46] and [47], on the same test set. The size of the train/test sets and the value of parameters such as the number of generations, the population size and Brute's timeout were chosen such that GeneticObjective runs in a reasonable amount of time. We ran the experiments on a machine running Ubuntu 20.04.4 LTS with *Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz*, 8 cores, 32GB of RAM.

The parameters of the GA are the following: the number of generations G, the population size N, the crossover probability p_c, the mutation probability p_m, the tournament size s, the elite size e, the maximum height H and the probability p of placing an operator symbol for the construction of a random tree, the weights $\{w_i\}_{i=1}^3$ in the fitness function $\mathcal{V}$ and the timeout t given to Brute in the fitness evaluations.

We experimented with various values of p and H to test the influence of the initial population on the performance of the GA. From the construction of a random expression (Algorithm 2), it is clear that on expectation a $(1-p)^2$ fraction of the trees in the initial population has height 1. Thus, we mostly used values of p satisfying $p > 1 - \sqrt{0.1}$ to ensure that less than 10% of the initial population has a height value of 1 on expectation. This leads to a more diverse initial population in terms of tree structure/shape. To ensure a reasonable running time, we did not exceed the value of 5 for the maximum height H in the initial population. We explored the effect of selection pressure, by adjusting the values of s (we mostly tested binary and ternary tournaments) and e (mostly in the range $[2, 16]$). Finally, we tested whether providing more time (increasing G) or examining more candidate solutions in each generation (increasing N) leads to an improvement of *GeneticObjective*. We present the configurations of selected experiments for the Robot and the String domains. For each configuration, we present: a) the maximum fitness value f_train obtained by *GeneticObjective* on the train set, b) the fitness value f_test of the best function found on the test set and we compare the latter to the fitness value f_manual of the manually-designed function on the test set. **In the case of the Robot domain, the best fitness value obtained by *GeneticObjective* (0.95) is equal to the fitness value obtained by the manually-designed function. As for the String domain, a fitness value of 0.29 was reached, which outperforms the one of the manually-designed function (0.24).**

4.1 Robot planning domain

A set of 150 examples of different complexity (grid size) were used for the train set and 350 examples were used for the test set. This sample was drawn from a larger set of examples that was used by [18]. In Table 1, we present selected configurations (with different values of the parameters) and the respective results.

Table 1: Robot experiments

| id	G	p_c	p_m	s	e	H	p	f_train	f_test
1	20	0.8	0.08	3	6	2	0.7	0.895	0.862
2	15	0.9	0.01	3	6	2	0.7	0.774	0.78
3	15	0.9	0.05	3	6	2	0.7	0.819	0.769
4	15	0.9	0.11	3	6	2	0.7	0.957	0.943
5	15	0.9	0.16	3	6	2	0.7	0.785	0.773
6	30	0.8	0.08	2	6	4	0.5	0.861	0.854
7	30	0.8	0.08	2	6	4	0.7	0.862	0.864
8	30	0.8	0.08	2	6	4	0.9	0.936	0.788
9	15	0.5	0.08	3	6	2	0.7	0.182	0.178
10	15	0.6	0.08	3	6	2	0.7	0.867	0.868
11	15	0.7	0.08	3	6	2	0.7	0.887	0.847
12	15	0.9	0.08	3	6	2	0.7	0.961	0.947
13	15	0.8	0.08	2	6	1	0.7	0.862	0.866
14	15	0.8	0.08	2	6	2	0.7	0.965	0.89
15	15	0.8	0.08	2	6	3	0.7	0.747	0.613
16	15	0.8	0.08	2	6	4	0.7	0.862	0.871
17	15	0.8	0.08	2	6	5	0.7	0.876	0.832
18	15	0.8	0.01	2	6	4	0.7	0.742	0.607
19	15	0.8	0.15	2	6	4	0.7	0.96	0.818
20	15	0.8	0.08	2	10	2	0.7	0.867	0.867
21	15	0.8	0.08	2	16	2	0.7	0.9	0.845
22	15	0.8	0.08	2	2	2	0.7	0.335	0.272
23	15	0.8	0.08	3	6	2	0.7	0.96	0.947
24	15	0.8	0.08	4	6	2	0.7	0.867	0.868

$$f_\text{manual} = 0.95$$

In the fitness function, we assigned a weight of 0.9 to the percentage of examples solved and a weight of 0.1 to the average normalized running time (solving more examples successfully is more important than running fast). We used $N = 60, t = 1$sec. Increasing G did not lead to any improvement and thus we mostly ran the GA with $G = 15$. The diversity of the population is affected by p_c and we observe that for a low value of $p_c = 0.5$ (configuration 9), the algorithm performs poorly. This is why we ran most of the experiments with $p_c \in \{0.8, 0.9\}$. A low value for the probability of mutation, such as $p_m = 0.01$ (e.g. configuration 18), also negatively affects the performance of *GeneticObjective*, since a low degree of randomness may lead to premature convergence. A value of $p_m = 0.16$ led to a suboptimal solution (configuration 5) as well; such a high value adds more stochastic noise to the procedure. The effect of p_m can be seen from configurations $2 - 5$, where it takes the values $p_m = 0.01, 0.05, 0.11, 0.16$, while all other parameters are fixed. As seen from the results of configuration 22, a low value of $e = 2$ led to poor results, highlighting the significance of *elitism* as suggested by [60, 61]. We hypothesize that this is due to lack of exploitation of the good solutions (individuals). It seems that H, p and G do not significantly

affect the efficacy of *GeneticObjective*.

The best results were obtained with configurations 12 and 23, for both of which $f_{test} = 0.947$. We ran the latter configuration 5 times and the curves of the best and the average fitness values over generations for each trial are shown in Figure 1. It is interesting to observe that the two trials in which the value of 0.94 was reached had a significant difference in the maximum fitness value of the initial population (0.55 vs 0.26). From the same Figure, we notice the convergence of the best fitness curve after 6-7 generations for all trials. The best objective function that was found (in 36 minutes) after simplifying the expression is the following (using the notation from 2):

$$T^*_{\text{robot}} = L^1(p_b, p_b^*)(L^1(p_r, p_b) + 2) + L^1(p_r, p_r^*) \qquad (6)$$
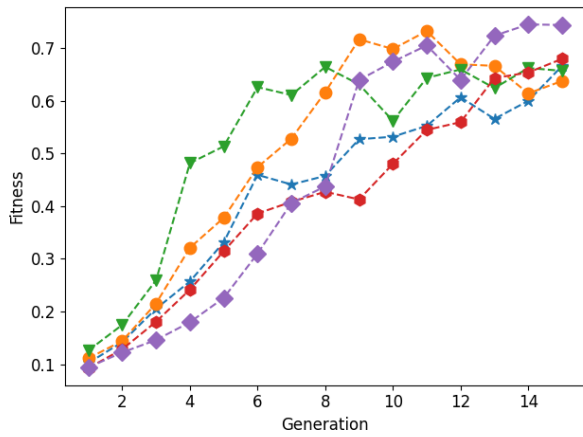
(a) Average fitness curve for each of the 5 runs
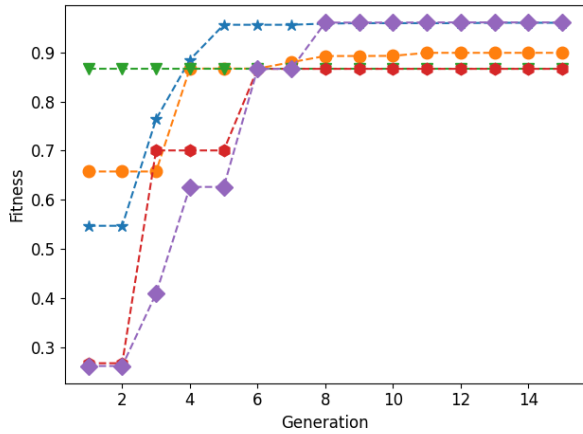
(b) Best fitness curve for each of the 5 runs

Figure 1: Best configuration in the Robot domain

4.2 String transformation domain

For the String domain, we used a train set of 150 examples and a test set of 225 different examples drawn from a larger

Table 2: String experiments

id	G	p_c	p_m	s	H	p	f_{train}	f_{test}
1	15	0.8	0.08	3	2	0.7	0.289	0.285
2	15	0.9	0.16	2	2	0.7	0.281	0.263
3	20	0.9	0.1	3	4	0.7	0.352	0.238
4	15	0.9	0.1	3	4	0.7	0.328	0.247
5	15	0.9	0.1	2	1	0.7	0.293	0.239
6	15	0.9	0.1	3	1	0.7	0.261	0.259
7	15	0.9	0.1	2	2	0.7	0.292	0.239
8	15	0.9	0.1	3	2	0.7	0.293	0.239
9	15	0.8	0.01	3	4	0.7	0.351	0.27
10	15	0.8	0.05	3	4	0.7	0.317	0.218
11	15	0.8	0.1	3	4	0.7	0.343	0.265
12	15	0.8	0.2	3	4	0.7	0.299	0.184
13	15	0.7	0.05	3	3	0.7	0.388	0.286
14	15	0.6	0.05	3	3	0.7	0.308	0.288
15	15	0.8	0.05	3	4	0.8	0.293	0.257
16	15	0.8	0.05	3	4	0.9	0.339	0.243
17	15	0.9	0.1	2	3	0.7	0.33	0.276

set used by [18]. The levels of difficulty (complexity) that were chosen for each set were the same. Selected configurations of the experiments in the Robot domain are shown in Table 2, together with the respective fitness values. Regarding the weights in the fitness function $\mathcal{V}$ (see Equation 4), we used $[w_1, w_2, w_3] = [0.6, 0.3, 0.1]$, i.e. solving many tasks of the domain is the most important goal. We used $e = 6$, which was shown to perform well in the Robot domain. Most configurations of the parameters led to good results compared to the manually-designed function, which had a fitness value of 0.24. An exception was configuration 12, which resulted in a poor performance on the test set. This was because of the high mutation probability $p_m = 0.2$ that makes the GA similar to random search. The objective function with the highest fitness value on the test set was found (in 98 minutes) with configuration 14 and the function evolved is the following (after simplifying):

$$T^*_{\text{string}} = 4d_{len} + \frac{d_m(d_p + d_{len})}{d_p} \qquad (7)$$

, where $d_{len}(s, s^*) = abs(|s| - |s^*|)$, $d_m(s, s^*) = min(|s|, |s'|) - |s \cap s^*|$ and $d_p(s, s^*) = abs(i - i^*)$ (see notation in 3). Configuration 13 led to a similar performance on the test set and a higher value on the train set. If we think of the difference between f_{test} and f_{train} as the generalization error, the latter configuration suffered in this aspect. This is likely due to the complexity of the function that was evolved, which contained 19 operators compared to the 7 operators of T^*_{string} in the expression tree representation before the simplification.

5 Conclusions

In this study, we examined an important aspect of a program synthesizer in the Programming By Example (PBE) paradigm, namely the objective (distance) function that is used to measure the performance of a generated program. In particular, we assessed the effect of using an objective function evolved by a Genetic Algorithm on the performance of a

program synthesizer. In order to answer Question 1, we proposed the *GeneticObjective* GA that evolves domain-specific objective functions by combining several user-defined *local distance functions* in an algebraic expression. Such an approach partly automates the laborious task of manually designing an informative distance function. We tested our proposed method using the *Brute* synthesizer on two well-studied program synthesis domains: the Robot planning and the String transformation domain. For each domain, we analyzed the performance of *GeneticObjective* and we compared the quality of the evolved function with an existing manually-crafted function. The experiments we conducted showed that our approach had a positive result on both domains, by reaching or even surpassing the manually-designed function. We hypothesize that the low scores for both the evolved and the existing function in the string domain are due to the inherent difficulty of the domain and thus a more effective search procedure would be required to obtain better results.

6 Limitations and Future Work

Due to the high amount of computational resources required for the experiments, this study had a limited scope regarding the number of configurations tested and the number of trials per configuration. Thus, future work could possibly focus on experimenting with more such configurations.

In the case of well-studied domains, such as the Robot and the String domains that were used in this study, there exist manually-designed objective (distance) functions that perform well in practice. An interesting direction would be to integrate such an existing objective function f in the fitness function of *GeneticObjective*. Specifically, the distance, as given by f, between a program's output and the correct output could be considered together with the percentage of successful examples, the percentage of successful tasks and the running time of Brute. Such an approach could lead to an improvement of the GA, even though we suspect that a small weight should be given to the distance component in order to avoid the GA attempting to "mimic" the given distance function.

Our suggested method for constructing the initial population does not ensure the absence of equivalent objective functions (due to associativity, distributivity and commutativity). Altering the way in which objective functions are represented and constructed could lead to an improvement of *GeneticObjective*. Although constructing equivalent algebraic expressions occurs with low probability as the number of local distance functions and the size of an expression tree increase, an alternative representation could be tested.

Lastly, the scope of our study could be broadened by considering more complex objective functions than algebraic expressions. For instance, introducing conditional expressions between the local distance functions could lead to more informative objective functions and thus to an enhancement of *GeneticObjective*.

7 Responsible Research

The most crucial aspect concerning the level of integrity of our research is *Reproducibility*. In order to ensure that other scientists in the research community can run experiments using our proposed method, the repository containing the code base has been made publicly available[1]. Specific instructions on how the experiments can be ran and more information concerning the implementation files can be found in the README file of the given repository. The results we obtained rely on the speed of the machine used. Thus, to make our experimentation phase even more transparent, we have included the specifications of the computer used for this study in the Results section. Finally, we revealed the sources of the data sets that were used in our experiments and the corresponding data files can also be found on our repository.

References

[1] Zohar Manna and Richard J Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

[2] Joey Hong, David Dohan, Rishabh Singh, Charles Sutton, and Manzil Zaheer. Latent programmer: Discrete latent codes for program synthesis. In *International Conference on Machine Learning*, pages 4308–4318. PMLR, 2021.

[3] Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis. In *Logic, algebra, and computation*, pages 41–107. Springer, 1991.

[4] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 13–24, 2010.

[5] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. OOPSLA 2015, page 107–126, New York, NY, USA, 2015. Association for Computing Machinery.

[6] Sumit Gulwani. Automating repetitive tasks for the masses. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–2, 2015.

[7] James Clift, Daniel Murfet, and James Wallbridge. Geometry of program synthesis. *arXiv preprint arXiv:2103.16080*, 2021.

[8] Neel Kant. Recent advances in neural program synthesis. *arXiv preprint arXiv:1802.02353*, 2018.

[9] Robert Vacareanu, Marco A Valenzuela-Escarcega, George CG Barbosa, Rebecca Sharp, and Mihai Surdeanu. From examples to rules: Neural guided rule synthesis for information extraction. *arXiv preprint arXiv:2202.00475*, 2022.

[10] Jacob Devlin, Rudy Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. Neural program meta-induction. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 2077–2085, Red Hook, NY, USA, 2017. Curran Associates Inc.

---

[1]https://github.com/FabianRadomski/
EvolvingProgramSynthesisers/tree/efthymiou_objective_function

[11] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 990–998. JMLR.org, 2017.

[12] Ruyi Ji, Yingfei Xiong, and Zhenjiang Hu. Black-box algorithm synthesis–divide-and-conquer and more. *arXiv preprint arXiv:2202.12193*, 2022.

[13] Sumit Gulwani, Alex Polozov, and Rishabh Singh. *Program Synthesis*, volume 4. NOW, August 2017.

[14] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[15] Hidehiro Kobayashi, Masaharu Munetomo, Kiyoshi Akama, and Yoshiharu Sato. Designing a distributed algorithm for bandwidth allocation with a genetic algorithm. *Systems and Computers in Japan*, 35(3):37–45, 2004.

[16] Dilip Datta, André RS Amaral, and José Rui Figueira. Single row facility layout problem using a permutation-based genetic algorithm. *European Journal of Operational Research*, 213(2):388–394, 2011.

[17] AR Kavitha and C Chellamuthu. Brain tumour segmentation from mri image using genetic algorithm with fuzzy initialisation and seeded modified region growing (gfsmrg) method. *The Imaging Science Journal*, 64(5):285–297, 2016.

[18] Andrew Cropper and Sebastijan Dumancic. Learning large logic programs by going beyond entailment. *ArXiv*, abs/2004.09855, 2020.

[19] Andrew Cropper. Playgol: Learning programs through play. *arXiv preprint arXiv:1904.08993*, 2019.

[20] Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua Tenenbaum, and Stephen Muggleton. Bias reformulation for one-shot function induction. In *Proceedings of the Twenty-First European Conference on Artificial Intelligence*, ECAI'14, page 525–530, NLD, 2014. IOS Press.

[21] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224. IEEE, 2010.

[22] Rishabh Singh and Pushmeet Kohli. Ap: Artificial programming. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, April 2017.

[23] Richard J. Waldinger and Richard C. T. Lee. Prow: A step toward automatic program writing. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI'69, page 241–252, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

[24] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *SIGPLAN Not.*, 46(1):317–330, jan 2011.

[25] Sumit Gulwani. Programming by examples: Applications, algorithms, and ambiguity resolution. In *Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706*, page 9–14, Berlin, Heidelberg, 2016. Springer-Verlag.

[26] Alan Leung, John Sarracino, and Sorin Lerner. Interactive parser synthesis by example. *ACM SIGPLAN Notices*, 50(6):565–574, 2015.

[27] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Learning to combine per-example solutions for neural program synthesis. *Advances in Neural Information Processing Systems*, 34, 2021.

[28] Sumit Gulwani, William R Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[29] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 319–336. Springer, 2017.

[30] Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. Reconciling enumerative and deductive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1159–1174, 2020.

[31] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121, 1980.

[32] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *International Conference on Machine Learning*, pages 187–195. PMLR, 2013.

[33] Kensen Shi, Hanjun Dai, Kevin Ellis, and Charles Sutton. Crossbeam: Learning to search in bottom-up program synthesis. *arXiv preprint arXiv:2203.10452*, 2022.

[34] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[35] Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis beyond domain-specific languages. *Advances in Neural Information Processing Systems*, 34:22196–22208, 2021.

[36] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

[37] Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. In *International Conference on Computer Aided Verification*, pages 398–414. Springer, 2015.

[38] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 27–38, New York, NY, USA, 2013. Association for Computing Machinery.

[39] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 997–1016, 2012.

[40] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 275–286, 2012.

[41] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186*, 2018.

[42] Colin G Johnson. Genetic programming with fitness based on model checking. In *European Conference on Genetic Programming*, pages 114–124. Springer, 2007.

[43] Hila Peleg and Nadia Polikarpova. Perfect is the enemy of good: Best-effort program synthesis. *Leibniz international proceedings in informatics*, 166, 2020.

[44] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 792–800. AAAI Press, 2015.

[45] Shivam Handa and Martin C Rinard. Inductive program synthesis over noisy data. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 87–98, 2020.

[46] Stef Rasing. Improving inductive program synthesis by using very large neighborhood search and variable-depth neighborhood search. https://repository.tudelft.nl/islandora/object/uuid:a24ed4f6-6abd-4661-86b8-c5a965d62e4e?collection=education, 2022.

[47] Bas Jenneboer. Program synthesis with a*. https://repository.tudelft.nl/islandora/object/uuid:873c3b33-2501-4438-a610-6dcb8ab8ad72?collection=education, 2022.

[48] D. Corne, H.-L. Fang, and C. Mellish. Solving the modular exam scheduling problem with genetic algorithms. In *Proceedings of the 6th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, IEA/AIE'93, page 370–373. Gordon Breach Science Publishers, 1993.

[49] Jens Lienig and Krishnaiyan Thulasiraman. A new genetic algorithm for the channel routing problem. In *Proceedings of 7th International Conference on VLSI Design*, pages 133–136. IEEE, 1994.

[50] A Kapsalis, VJ Raywad-Smith, and George D Smith. Solving the graphical steiner tree problem using genetic algorithms. *Journal of the Operational Research Society*, 44(4):397–406, 1993.

[51] Penousal Machado and António Leitao. Evolving fitness functions for mating selection. In *European Conference on Genetic Programming*, pages 227–238. Springer, 2011.

[52] Andrew Lensen, Bing Xue, and Mengjie Zhang. Genetic programming for evolving similarity functions for clustering: Representations and analysis. *Evolutionary computation*, 28(4):531–561, 2020.

[53] John R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4:87–112, 1994.

[54] Muzafer Saračević, Saša Adamović, and Enver Biševac. Application of catalan numbers and the lattice path combinatorial problem in cryptography. *Acta Polytechnica Hungarica*, 15(7):91–110, 2018.

[55] David E Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of genetic algorithms*, volume 1, pages 69–93. Elsevier, 1991.

[56] Tobias Blickle and Lothar Thiele. A mathematical analysis of tournament selection. In *ICGA*, volume 95, pages 9–15. Citeseer, 1995.

[57] Anupriya Shukla, Hari Mohan Pandey, and Deepti Mehrotra. Comparative review of selection techniques in genetic algorithm. In *2015 international conference on futuristic trends on computational analysis and knowledge management (ABLAZE)*, pages 515–519. IEEE, 2015.

[58] Brad L Miller, David E Goldberg, et al. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995.

[59] Gabriela Ochoa, Inman Harvey, and Hilary Buxton. Optimal mutation rates and selection pressure in genetic algorithms. In *Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation*, pages 315–322. Citeseer, 2000.

[60] Giorgio Guariso and Matteo Sangiorgio. Improving the performance of multiobjective genetic algorithms: An elitism-based approach. *Information*, 11(12):587, 2020.

[61] Robin C Purshouse and Peter J Fleming. Why use elitism and sharing in a multi-objective genetic algorithm? In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary computation*, pages 520–527, 2002.

[62] Chang Wook Ahn and Rudrapatna S Ramakrishna. Elitism-based compact genetic algorithms. *IEEE Trans-*

actions on Evolutionary Computation, 7(4):367–385, 2003.