Human-Driven Genetic Programming for Program Synthesis: A Prototype

Thomas Helmuth
Hamilton College
Clinton, New York, USA
thelmuth@hamilton.edu

James Gunder Frazier
Hamilton College
Clinton, New York, USA
jgfrazie@hamilton.edu

Yuhan Shi
Hamilton College
Clinton, New York, USA
yxshi@hamilton.edu

Ahmed Farghali Abdelrehim
Hamilton College
Clinton, New York, USA
aabdelre@hamilton.edu

ABSTRACT

End users can benefit from automatic program synthesis in a variety of applications, many of which require the user to specify the program they would like to generate. Recent advances in genetic programming allow it to generate general purpose programs similar to those humans write, but require specifications in the form of extensive, labeled training data, a barrier to using it for user-driven synthesis. Here we describe the prototype of a human-driven genetic programming system that can be used to synthesize programs from scratch. In order to address the issue of extensive training data, we draw inspiration from counterexample-driven genetic programming, allowing the user to initially provide only a few training cases and asking the user to verify the correctness of potential solutions on automatically generated potential counterexample cases. We present anecdotal experiments showing that our prototype can solve a variety of easy program synthesis problems entirely based on user input.

CCS CONCEPTS

• **Software and its engineering → Genetic programming**.

KEYWORDS

automatic programming, genetic programming, interactive evolution

ACM Reference Format:
Thomas Helmuth, James Gunder Frazier, Yuhan Shi, and Ahmed Farghali Abdelrehim. 2023. Human-Driven Genetic Programming for Program Synthesis: A Prototype. In *Genetic and Evolutionary Computation Conference Companion (GECCO '23 Companion), July 15–19, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3583133.3596373

1 INTRODUCTION

Genetic programming (GP) has long targeted the automatic synthesis of general-purpose programs [19], and in recent years has made progress on synthesizing functions similar to those that humans often write [11, 29]. General program synthesis systems need to create programs that manipulate multiple data types and use control flow such as loops or recursion. There are many potential applications of automatic program synthesis, ranging from programmer assistants to computer science education. Many (though not all) of these applications have a human user synthesizing a program for some purpose.

However, specifying new problems for GP from scratch typically requires extensive effort from a human user. In particular, systems require problem descriptions that include the data types and instructions that might be used in the program, the number of inputs/outputs and their data types of the training cases, and the set of training cases themselves. In particular, GP systems are usually given hundreds of labeled training cases used for evaluating the evolving programs, an unreasonable number for a human user to provide.

In this paper, we consider what changes need to be made to a GP program synthesis system to allow a non-GP-expert human user to provide full specifications of a problem to solve. We devise, implement, and experiment with a prototype of a human-driven GP (HDGP) system. In doing so, we tackle the following research questions:

RQ1. Interactions: What would a human-driven GP system for program synthesis look like? What are the interactions necessary to provide specifications and start a GP run?
RQ2. Training Cases: Typically, GP needs hundreds of training cases in order to evolve effectively. Can we reduce the number of training cases to an amount reasonable for a human to provide manually?
RQ3. Limitations: What are the limitations of the our system in terms of types of problems it can tackle?

To address RQ1, we consider which pieces of a GP system need to be specified for a new problem, and which can be automatically parameterized. We find that the user must specify two key aspects of a problem: what are the initial training cases that are used to evaluate evolving programs, and which instructions and literals should be present in programs. On the other hand, we stick with constant or automatically defined choices for other hyperparameters, such as population size, maximum number of generations, and error function. With error functions in particular, we select an error function based on the data type of the output, as recommended by benchmarks of program synthesis problems [10].

Based on our answers to RQ1, we have created a prototype HDGP system that can fully specify new problems using user input and find solutions to those problems. In this paper, we demonstrate this HDGP system, showing the places where it receives input from the user.

RQ2 presents a larger challenge, since GP systems have typically been tested in research on large numbers of training cases, and rely on them to guide evolution when determining parents for the next generation. We would ideally prefer that the user only has to enter a few training cases, perhaps 5 or 10, to reduce their burden in generating these examples of the desired program's behavior. No program can be fully specified by training examples alone, since it may give the wrong answer on some input not in the training set. However, a synthesized program that passes a larger number of training cases (such as hundreds) may give us more confidence that it implements the desired behavior than a program that passes a few cases, simply because it exhibits the correct behavior across a larger set of inputs. This results in a trade-off between specifying fewer cases, which is easier for the user, and specifying more cases, which provide more information for guiding evolution and gives more confidence in the generalization of synthesized programs.

In this work, we propose a compromise, where we ask the user for only a few training cases, but do not assume that an evolved program that passes those cases actually implements the desired functionality. Instead, we borrow ideas from *counterexample-driven genetic programming* (CDGP), which tries to find *counterexample cases* that a potential solution program does not pass, and if so, adds those counterexamples to the set of training cases as evolution continues [7]. Since CDGP requires some method of determining whether a synthesized program passes potential counterexample cases, we rely on the human user to verify whether the program produces the correct outputs for a set of inputs.

Now, verifying the correctness of input/output behavior on potential counterexample cases is likely not as taxing as providing inputs and outputs from scratch. However, we would still like to limit how much work we ask the user to do in this process; for example, asking the user to verify correctness on hundreds of examples is infeasible, and we would prefer providing a number of cases the user could check in a minute or two. With this limited number of counterexamples that we can ask the user to check, it is important to choose cases that are more likely to be actual counterexamples and, if the program does not fully solve the problem, reveal its flaws. If we instead pick cases poorly, it is more likely that the evolved program passes those cases even if it would fail at others. With this in mind, we consider four different potential counterexample generation methods aimed at generating more useful cases.

To give a full view of what our prototype can and cannot do, we address RQ3 by describing some types of problem specification that the system cannot (yet) handle. In particular, we cannot specify problems that have constraints on the inputs beyond data type and range. These limitations provide avenues of future research.

In the next section we describe related work, including examining different methods of problem specification as well as describing CDGP in detail. We give a high-level overview of the HDGP algorithm in Section 3. We address RQ1 in Section 4 by describing the information necessary to specify a new problem in HDGP. In Section 5, we address RQ2 by describing how the user verifies evolved potential solution programs. We then describe experiments using our HDGP prototype, and discuss its limitations.

2 RELATED WORK

In this work, we specifically address the problem of automatic program synthesis where the specifications of the desired program are given as examples of inputs and their corresponding outputs, which we call training cases. This type of synthesis is usually called *programming by example* or *inductive program synthesis*, and has been tackled by a variety of techniques [4, 9, 11, 22].

Other types of program synthesis use other kinds of specifications. Synthesis of code using large language models has recently become feasible; such systems typically specify the desired program using natural language descriptions and sometimes starter code such as function headers [3, 8, 28]. On the other hand, the models are not given examples of correct behavior, so it is possible that generated code does not produce correct results on any inputs.

Other program synthesis systems require formal specifications, and are able to generate code that is guaranteed to meet those specifications [1, 2, 5, 30]. While formal specifications make more guarantees about what the generated program can do, they also require the user to be trained in formal specifications and have more knowledge about the problem at hand, often to the point where the specification is more detailed than the generated program. These restrictions eliminate some of the potential applications of human-driven program synthesis.

This work focuses on human-driven GP for program synthesis using the programming by example framework. Recent work has advanced the state of the art in GP's application to program synthesis, without considering the implications of having a human specify the problem. The PSB1 [14] and PSB2 [10, 11] benchmark suites have provided sets of problem definitions and training data, allowing different GP systems to be tested and benchmarked on a consistent set of problems [13, 15, 18, 24, 25, 29]. However, all of these studies assume that hundreds of training cases are available to evaluate the evolving programs, which is not the case if a human user is providing the training data.

Recently, the problem of human-interactive program synthesis motivated the development of Metamorphic Testing Genetic Programming (MTGP) [27], which takes a different approach than ours to evolving programs with limited training data. MTGP requires the user to create *metamorphic tests* that define relations between the output of a program on a random input and how that output should relate to the output of the program when run on a slightly altered input. By providing a reduced number of hand-labeled training cases and a few metamorphic relations, MTGP automatically generates a larger number of metamorphic cases, giving GP more information than with the hand-labeled data alone. This system showed moderate improvements in success rate and generalization when compared to the same system with the training cases alone.

Algorithm 1 Human-Driven Genetic Programming

```
initialization : problem specified by the user
training is initial set of training cases specified by user
$g \leftarrow 0$                                    ▷ $g$ is the generation counter
while $g <$ max generations do
    run 1 generation of GP
    let $p$ be the best program in $g$ evaluated on training
    if $p$ passes all cases in training then
        generate potential counterexamples
        user verifies correctness of $p$ on potential counterexamples
        if $p$ passes all potential counterexamples then
            return $p$
        else
            add failed counterexamples to training
    $g \leftarrow g + 1$
return "no solution found"
```

Our approach to human-driven GP is inspired by counterexample-driven genetic programming (CDGP) [6, 7, 20, 21]. CDGP uses specifications provided as formal constraints in order to generate training cases [7, 20, 21]. CDGP was extended to use both formal constraints and user-provided training data to solve symbolic regression problems [6]. CDGP evaluates individuals in the population against both a set of automatically generated training cases and the provided formal constraints. In particular, the training cases are used for evaluating individuals for parent selection. When a program passes all of the training cases, it may or may not generalize to work on inputs not in the training set. Such a program is tested using a satisfiability modulo theories (SMT) solver to evaluate it on the problem's formal constraints. If the program passes the formal specifications, it solves the problem and is returned by the system. Otherwise, the SMT solver creates a *counterexample case* that this program gives the wrong answer to, adds the case to the set of training cases, and continues evolution.

Informal CDGP adopts the ideas of CDGP to solve problems that do not have formal constraints defined for them [17]. Instead, it assume that a large set of labeled training data is available for the problem, with 100 or more training cases. It then starts evolution using a small random subsample of those training cases, and evolves until a program passes those training cases. Then, since there are no formal constraints, it tests the program on all training cases not in the current subsample of the training set, to see if any are counterexamples to the program. If so, it adds those cases to the training set and continues evolution, like in CDGP. Informal CDGP forms the basis for how our human-driven GP system creates and evaluates *potential counterexamples*, with the human evaluating whether the program passes each case.

3 HDGP ALGORITHM

We present a high-level overview of the HDGP algorithm in Alg. 1. This algorithm outlines the key stages involved in completing a GP run using our HDGP system. In Sections 4 and 5, we will explore the mechanisms underlying the algorithm that facilitate human interaction. These mechanisms include problem specification, generation of counterexamples, and human verification.

4 PROBLEM SPECIFICATION

In this section, we address RQ1 by describing how the user provides problem-specific hyperparameters to initialize our HDGP system. Our prototype system uses a text-based user interface to receive information from the user; a more fully-featured GUI interface could be implemented to gather the same information in a fleshed-out system.

To acquire the necessary specifications for an HDGP run, our prototype presents a series of questions requesting either text or multiple-choice inputs. The system processes and formats the direct user input into data structures which will be utilized to parameterize a GP run and generate test cases when necessary. We separate the types of prompts into two categories: problem training data and instruction sets. Additionally, there are GP hyperparameters that would normally be set per problem that we automatically parameterize.

4.1 Problem Training Data

Our prototype HDGP system creates a problem-specific input/output interface based on interactions with the user. This interface is used both when the user enters the initial set of training data and when the system generates potential counterexamples to an evolved program. Since potential counterexample generation requires more extensive specification of the shape of inputs than outputs, the system requests more user interaction to define the inputs. Finally, our prototype asks the user to enter the actual inputs and outputs for the set of initial training cases. This process is summarized in Figure 1 and detailed below.

4.1.1 Case Interface. In order to define the input/output interface for the problem, we begin by requesting the number of input parameters the problem requires for each case. For each input parameter, the system provides a series of questions requesting the input parameter's data type and behavior. Our prototype supports the following data types, which are listed along with the other information required from the use to generate potential counterexamples:

- **Integer**: the lower-bound and upper-bound of possible inputs.
- **Float**: the lower-bound and upper-bound of possible inputs.
- **String**: the lower-bound and upper-bound of possible string lengths along with the set of characters that can appear in the strings.
- **Vector**: the lower-bound and upper-bound of possible vector lengths along with the data type of the elements, which may be one of the above. The data type of the elements must also be characterized by the above specifications.

Next, the system requests the number of outputs and the data types of each output. Each output can be either an integer, float, string, Boolean, or a vector of the previous types. Often there is only one output, but some problems may require multiple outputs.

4.1.2 Initial Training Cases. The user must define the initial set of training cases that HDGP uses to evaluate evolving programs. Our prototype first requests the number of initial training cases. For each case, the system requests the input parameters and the correct output(s) for those inputs. The system uses the case interface provided previously by the user to specify which input/output it is

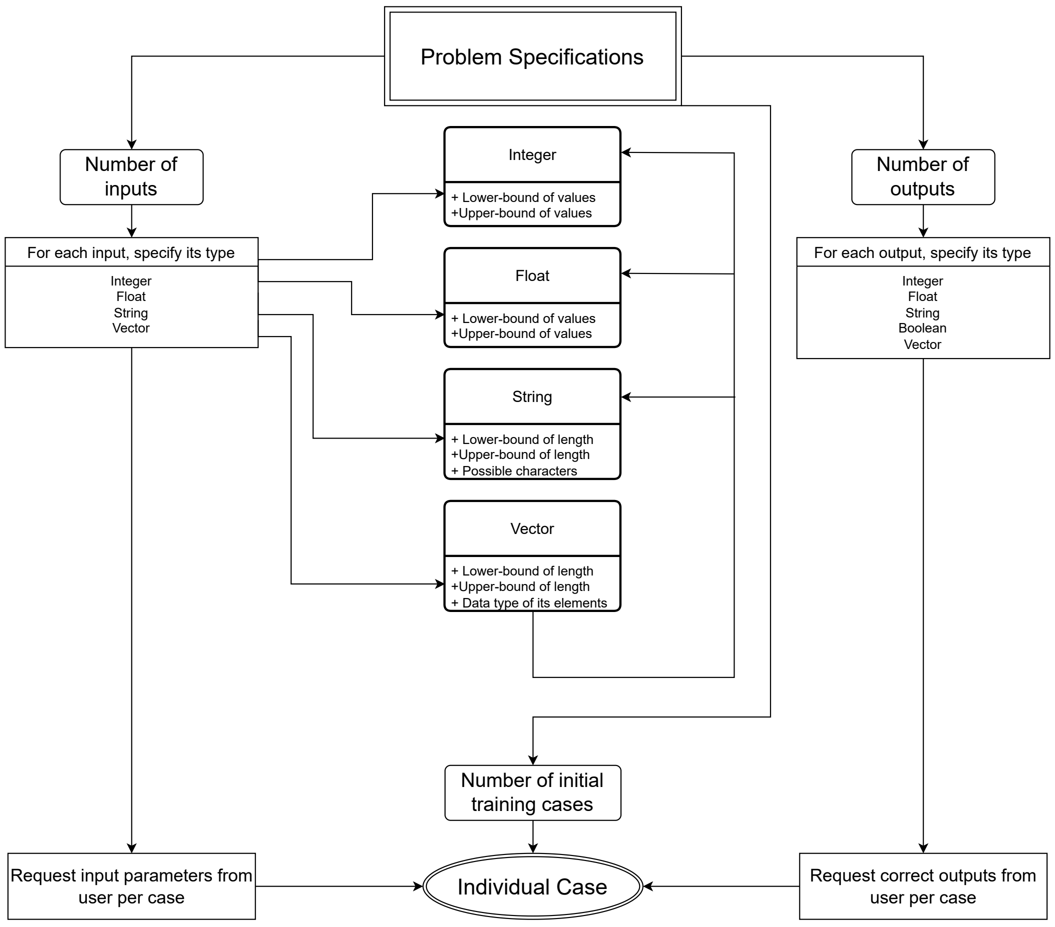

Figure 1: Diagram of the information required from the user to define the training data for a new problem.

requesting and furthermore the data type of such to reduce possible confusion. However, the system does not require the inputs to conform to the ranges given in the case interface. This allows the user to specify edge cases that are outliers to the general range of cases a program may encounter. For example, the user may want to include an initial training case whose input is an empty string, and then require all other cases' inputs to contain at least one character.

4.2 Instruction Sets and Literals

In addition to the training data, we have the user specify the set of instructions and literals that may appear in evolving programs. We adopt the practice of general program synthesis benchmarking where the problem specifies only the data types that must be manipulated by the evolving programs, and all instructions that use those data types are automatically included in the instruction set [11]. Thus our prototype system simply asks the user which data types should be included and automatically populates the instruction set from those.

We provide two methods of specifying literals to include in the instruction set. First, the system displays sets of suggested literals for each data type in the instruction set, which the user may include or not. These sets include common literals depending on the data type, such as $\{-1, 0, 1\}$ for integers and the empty string for strings. Then, the system will query for other literals to include for each instruction data type.

4.3 Automatic Hyperparameters

We have chosen to hold constant or automatically set some GP hyperparameters, to minimize the effort of the user to specify a problem. For many common GP parameters, such as population size and parent selection method, we have chosen common values from other program synthesis work, which are detailed in Section 6.

For the GP error function, which determines how close the program's output is to the correct output on each training case, we automatically choose an appropriate error function depending on the output's data type, using those described for PSB2 [11]. For example, integers and floats use the absolute difference between expected and actual outputs, while strings use the Levenshtein edit distance.

5 CORRECTNESS VERIFICATION

In this section, we describe how the user interacts with the system during the evolutionary process once it starts. We will focus on the verification of potential solution programs synthesized by the system, which involves generating potential counterexample cases and having the user check the program's outputs for correctness.

When a program is found that passes all current training cases, it may not generalize to other inputs, especially since the initial training set is likely very small. Once such a potential solution program has been found, HDGP generates a set of potential counterexample cases to verify the correctness of this program to the desired task. Then, the user verifies whether the potential solution program passes the potential counterexamples; if the program gives the wrong output on any cases, those cases are added to the training set while evolution continues. This verification will allow the user to expand the training set and avoid solution programs that do not generalize to unseen data, without the undue burden of verifying the correctness of the program by hand.

HDGP performs user-driven verification whenever it finds a program that passes all the examples in the current training set. This can happen any number of times during the evolutionary process and not just when the training set only contains the initial training cases. If the human verifies that the potential solution program passes all potential counterexamples, the run is terminated and that program is returned as a solution.

Since we do not want to burden the user with a very large set of potential counterexample cases, we would like to generate cases that are more likely to demonstrate that a potential solution program is incorrect, if it indeed is so. Thus HDGP is designed to generate different types of potential counterexamples, created to try to elicit incorrect behavior. We provide more detail about counterexample generation and the user verification process below.

5.1 Potential Counterexample Case Generation

The specifications for the GP run are used as a schema to generate potential counterexamples on which a potential solution program is verified. We created four types of potential counterexamples: fully random cases, edge cases, cases that aim to have dissimilar outputs from those in the training set, and cases that have distinct program traces from those in the training set.

5.1.1 *Random Cases.*

Random inputs are generated uniformly from within the boundaries given by the input parameters. These inputs are therefore the least engineered, but may in some cases hit upon issues not discovered by our other generators.

5.1.2 *Edge Cases.*

Edge cases often illuminate bugs in software; thus, we aimed to generate cases that fall in the extremes of the user-specified input boundaries. Based on the given boundaries, the edge case generator creates two sets of inputs, one with all input values of the minimum values/sizes and the other with all maximum values/sizes. When a problem has more than one input parameter, the edge case generator creates all possible combinations of the minimum and maximum values. For example, if the problem specifies two integer inputs each in the range $[0, 100]$, four edge cases will be generated for $(input1, input2) : (0, 0)$, $(0, 100)$, $(100, 0)$, and $(100, 100)$.

5.1.3 *Cases with Dissimilar Outputs.*

In order to detect cases on which the potential solution program gives the wrong answer, we attempt to generate cases on which the program produces outputs that are dissimilar from its outputs in the current training set. To begin, we randomly generate a large number of inputs and store them in an array. We then run the potential solution program on each random input, resulting in an output array. Those outputs are compared to every output in the training set. After the comparison, each input/output pair, regardless of data type, is assigned a *dissimilarity score* indicating how different its output is from the outputs of the current training set cases. Larger dissimilarity scores indicate more dissimilar outputs. Thus we select the most dissimilar input/output pairs as potential counterexamples.

For different data types, we use different dissimilarity metrics to measure the differences between the outputs. For integers and floats, the absolute value of difference between the numbers is recorded. For Booleans, the dissimilarity scores are measured by whether they are identical. For strings, the dissimilarity scores are determined by the Levenshtein distances. For vectors, to account for both the lengths and the values inside the vectors, both the dissimilarity among the values inside the vectors and the dissimilarity in vectors' lengths are calculated and summed. Note that if the program produces an error when run on an input, the dissimilarity score is set to an arbitrary large number to indicate the significance of an unexpected output.

5.1.4 *Stochastic White Box Testing Cases.*

Inspired by white-box testing [23], we attempt to generate cases that produce different evaluation paths through the potential solution program. Like with cases with dissimilar outputs, we evaluate the best program on randomly generated inputs and record the evaluation trace. We compare the random input traces to the training set traces and select those inputs which have the greatest dissimilarity to the training inputs as potential counterexamples. Since our HDGP system evolves Push programs that execute by manipulating data on typed stacks, the program traces look at the states of the stacks as the program runs. We'll discuss our use of Push more in Section 6.

More specifically, an array of randomly generated inputs is collected in preparation of the selection. Each random input is evaluated by the current best performing program. During the evaluation, the stack size after each instruction evaluation is recorded in an array. The same process is used on the inputs in the training set, resulting in a training set stack trace array. Each random case stack trace is then compared with the every training set stack trace, creating a Boolean array for each of the random inputs. Each Boolean array's length is the number of cases in the training set. Each element in the Boolean array indicates whether the current random case program trace is identical to one of the training set program traces. Each Boolean array is then simplified to an integer representing the number of identical stack traces in the array. At the end of the selection, every random input is paired with an integer indicating how many training set inputs share the same program trace as the random input. The potential counterexamples are selected based on the integer values. The smaller the number is, the more different the random input's program trace is, meaning such input might use a different path of the program than most training set inputs.

Hyperparameter	Value
Population Size	1000
Max Generations	300
Parent Selection	Lexicase Selection [16]
Variation	UMAD [13]
Mutation Rate	0.1
Initial Genome Size Range	[1, 250]

Table 1: The evolutionary hyperparameters used for all HDGP runs.

5.2 Human Interactive Verification

HDGP generates a set of 10-20 potential counterexample cases using the methods above to present to the user. The cases are presented to the user as inputs and outputs, which the user checks for correctness. Then, the user selects the potential counterexamples on which the best program gives the wrong answer, if any. The behavior of the system diverges at this point based on the response received from the user.

If the program passes all of the generated potential counterexamples, the GP run terminates since a program passes both the current training set and the generated cases. At this point, the program is returned as the solution program.

If the program fails at least one generated test case, as identified by the user, the system prompts the user to give the actual correct output for each case that the program did not pass. The correct outputs entered by the user are then paired with their respective inputs in the specified case format during the initialization step. These cases are then added to the training set as counterexamples. Afterward, the GP run continues with the updated training set until either a new potential solution program is found on the updated training set or the maximum number of generations is reached.

6 METHODS

In this section, we will discuss the PushGP, the GP system we used as the basis for our HDGP prototype, and system-specific hyperparameters for it. PushGP evolves programs in Push, a stack-based programming language where every data type has its own stack and program instructions are executed on an `exec` stack [31]. Each instruction pushes and pops data from the typed stacks. The output(s) of an evolved program are obtained by observing the top value(s) on the stack(s) specified by the problem. Our system is built on top of Clojush, the PushGP implementation in Clojure, a dialect of Lisp. The code used in our experiments is available on GitHub[1]. Independently from human specifications for a problem, the system-specific hyperparameters are fixed. A list of important hyperparameters is given in Table 1.

For our HDGP system, the user specifies which data types/stacks are relevant to the problem. Once these stacks have been identified, their instructions are then included in the instructions set used in the evolution process. As an example, in the **Substring** problem, the program is given a string and an integer as an index, and must return the substring that ends at this index exclusively. For this problem, the human should provide two types of inputs, which are string and integer, and one output of type string. This tells the system to work with the `string` and `integer` stacks in addition to the `exec` stack, which is required in every problem.

[1]https://github.com/thelmuth/Clojush/releases/tag/HDGP-Prototype

7 EXPERIMENTS AND RESULTS

In this section we exhibit our HDGP prototype's ability to take user specifications for new problems, generate potential counterexamples to those problems, conduct correctness verification with user input, and synthesize solution programs. These proof-of-concept experiments provide anecdotal evidence showing that our HDGP prototype is capable of solving new problems.

We devised 5 new easy program synthesis problems in the vein of the PSB benchmark problems [11, 14]. **Subtract Two Integers, Spaces,** and **Substring** are intended to be trivial problems for the prototype in order to allow us to observe the complete HDGP algorithm for various specifications. **Element-Rank** is designed illustrate the user input of cases whose inputs are vectors, and is moderately more difficult. **Fizz** was designed expressly to demonstrate the addition of counterexamples. Below we describe HDGP runs on each of these problems.

7.1 Subtract Two Integers Problem

In order to demonstrate HDGP's ability to preserve the ordering of its parameters, we devised the **Subtract Two Ints** problem. In this problem, the program is given $x, y \in \mathbb{N}$ where $-1000 \leq x, y \leq 1000$ and the program must return the difference $x - y$. In one run of this problem, HDGP found a solution in 12 generations when given 5 initial training cases. None of the potential counterexamples we generated were actual counterexamples, since the program solved the underlying problem.

7.2 Spaces Problem

In the Spaces problem, the program is given a string of characters. If any character in the string is a space, a correct program will return the Boolean *True*; otherwise, it will return *False*. HDGP found a solution in 37 generations when given 5 initial training cases. At total of four counterexamples were added to the training set after generations 26 and 28, with the inputs: `""` (empty string), `" "` (string containing one space), `"ab8"`, and `"7"`.

7.3 Substring Problem

In the Substring problem, the program is given a string S with length $0 \leq l \leq 20$ and an integer v where $0 \leq v \leq 20$. The program must return a substring of S, call it T, of the first v characters of S. If $v \geq l$, then $T = S$. We found HDGP was able to find a solution in 13 generations when given 5 initial training cases. Counterexamples were added at generations 2 and 11 for a total of 8 counterexamples added to the training case set, which included `("<li6HS>N9,f)6nJP&0K7", 0)`, for which the best program returned `"<"`, and `("2QVWGfxOa._a1", 1)`, for which the best program returned `"2QVWGfxOa._a"`.

7.4 Element-Rank Problem

In the Element-Rank problem, the program is given an unsorted vector V of non-negative integers. The program must return a vector W where the element at index i of W corresponds to the rank of the element at index i in V: the rank of the element at index i is determined by the lowest index at which the element would appear in a sorted vector of distinct values from V. For example, if $V = [16, 101, 2, 16]$, the solution vector is $W = [1, 2, 0, 1]$.

We will use the Element-Rank problem to illustrate the case interface initialization of the prototype system. A run of the problem is initialized as follows:

```
How many input parameters are given?:
1

What is the data type for parameter 1?
    (1) Integer
    (2) Float
    (3) String
    (4) VectorOf
Please choose a number from the options above.
4
For a VectorOf, please provide the following information.
Lower-bound of element-count:
1
Upper-bound of element-count:
100
Now specify the data type of each element of the vector:

What is the data type for parameter of the vector?
    (1) Integer
    (2) Float
    (3) String
    (4) VectorOf
Please choose a number from the options above.
1
For an Integer, please provide the following information.
Lower-bound of integer range:
0
Upper-bound of integer range:
1000000

How many outputs there are:
1
What is the data type for output 1 of the program?
    (1) Integer
    (2) Float
    (3) String
    (4) Boolean
    (5) VectorOf
Please choose a number from the options above
5
What is the data type of each element?
    (1) Integer
    (2) Float
    (3) String
    (4) Boolean
Please choose a number from the options above
1
```

After providing HDGP with the instruction sets and literals necessary for the run, the system prompts us to provide initial training cases based on our selections above. One of the initial training cases we provided is the input vector $[6, 5, 8, 8]$ with its corresponding output $[1, 0, 2, 2]$, which is entered into the system as follows:

```
Case # 1 :
 Inputs
 Vector:
      Element Count: 4
      Integer 1: 6
      Integer 2: 5
      Integer 3: 8
      Integer 4: 8

 Outputs
 Vector:
      Element Count: 4
      Integer 1: 1
      Integer 2: 0
      Integer 3: 2
      Integer 4: 2
```

Out of 4 separate GP runs, the HDGP prototype was not able to find a solution before reaching the generational maximum limit when provided 5 initial training cases. In fact, no program was found that passed all 5 of the initial cases. However, 3 of the 4 runs came close, with total errors off by 2.

7.5 Fizz Problem

In order to illustrate a situation in which an evolved program that passes the current training set does not solve the problem, we devised the **Fizz** problem, designed to be a simpler version of the FizzBuzz problem from PSB2 [10]. In the Fizz problem, the program is given an integer, and must return the string `"Fizz"` if the integer is evenly divisible by 3, and return a string version of the integer otherwise. When running this problem by hand, we initially do not provide any training cases that are divisible by 3, so that none of the correct answers are `"Fizz"`.

In a run of the Fizz problem, a program in the initial population solved each of the 5 initial training cases. The HDGP system generated the following 17 potential counterexamples and presented them to the user:

```
Case 0 : Input: [0] ; Program's output: ["0"]
Case 1 : Input: [10000] ; Program's output: ["10000"]
Case 2 : Input: [8143] ; Program's output: ["8143"]
Case 3 : Input: [5581] ; Program's output: ["5581"]
Case 4 : Input: [4580] ; Program's output: ["4580"]
Case 5 : Input: [2797] ; Program's output: ["2797"]
Case 6 : Input: [90] ; Program's output: ["90"]
Case 7 : Input: [6589] ; Program's output: ["6589"]
Case 8 : Input: [7369] ; Program's output: ["7369"]
Case 9 : Input: [3652] ; Program's output: ["3652"]
Case 10 : Input: [6988] ; Program's output: ["6988"]
Case 11 : Input: [6876] ; Program's output: ["6876"]
Case 12 : Input: [2490] ; Program's output: ["2490"]
Case 13 : Input: [3183] ; Program's output: ["3183"]
Case 14 : Input: [4481] ; Program's output: ["4481"]
Case 15 : Input: [6079] ; Program's output: ["6079"]
Case 16 : Input: [514] ; Program's output: ["514"]
```

With a brief inspection, it was clear to the user that cases 0, 6, and 13 all contained inputs that are evenly divisible by 3, and selected them to be added to the training set, all with the correct output of "fizz" (the user missed that cases 11 and 12 are also divisible by 3, so those cases were not added). HDGP continued for 7 generations, at which point it found a program that passed the original and the added cases. HDGP presented 17 new potential counterexamples, and this time the program passed all of them, so evolution was terminated. The simplified Push program that solved this problem is:

```
(false integer_empty exec_rot 3 (integer_mod
  exec_dup_times boolean_eq) in1 "fizz" in1
  string_frominteger integer_fromboolean string_yank)
```

These results provide evidence that the HDGP prototype can, given only user input, (A) create counterexample cases that a potential solution fails on, allowing the user to add them, and (B) find general solutions that pass the training cases and the generated potential counterexamples.

8 LIMITATIONS

While we designed the HDGP prototype to be able to support a variety of program synthesis tasks, there are certain problem characteristics and GP details that are not yet possible to account for. One simple example of this is ephemeral random constants (ERCs) [19], which cannot be provided to the prototype instead of fixed literals. While we could have added the ability to specify ERCs, each ERC would require multiple user interactions to parameterize the random constant generator for a specific type, such as ranges and distributions. Thus ERCs are not currently available.

A more difficult issue to tackle is how to specify the training case interface for problems whose inputs have constraints beyond their types and ranges. When this happens, the random inputs generated by our potential counterexample generators might not even be legal inputs for the problem.

For example, consider if the user wanted to solve the Last Index of Zero problem from PSB1 [14], in which the program must return the last index at which a 0 appears in a given vector of integers. While our HDGP prototype can generate random vectors of integers to use as inputs, there is no way to specify that at least one of the integers must be 0. Thus many or most of the potential counterexamples will not be legal inputs for the problem. While many problems can be defined without constraints on their inputs, many cannot; here are a few examples from PSB1 [14] and PSB2 [11]:

- **Find Pair**: The input is a vector of integers and a target, and two of the integers in the vector must sum to the target. Also, there must be only one unique pair that sums to the target.
- **Camel Case**: The input is a string written in `kebab-case`, with dashes between words and spaces between phrases. We cannot specify that there must be dashes in the input, and more of them than other types of characters.
- **Mirror Image**: The input is two vectors of integers of the same length. Since the problem is to determine whether one vector is the reverse of the other, about half of the cases should actually be true and the other half not. However, there is no way to specify that many cases should have one input be the reverse of the other.

Since we currently have no way of addressing this issue, problems with constraints on their inputs cannot currently be run in the HDGP prototype.

9 CONCLUSIONS AND FUTURE WORK

We have presented a prototype human-driven genetic programming system that is capable of synthesizing general-purpose programs given only user input. This system requires user interaction in two places: when initially specifying the problem to solve, and when verifying potential solution programs against potential counterexamples. We devised a set of four different potential counterexample case generators in an attempt to effectively identify inputs that the program gives the wrong answer to, if it is indeed incorrect.

While this paper shows that HDGP is able to solve moderately simple program synthesis problems, it does not collect quantitative data on its efficacy. This raises a variety of questions to be answered in future work. To start, can HDGP solve more challenging problems, such as those in the PSB benchmark suites [11, 14]?

There are many trade offs we had to consider in creating HDGP. For example, how many training cases should the user provide? More training data better specifies the problem to be solved, but requires more user time to give the specification. Similarly, how many potential counterexamples should be presented to the user in order to be fairly confident that real counterexamples are given when necessary? These and other questions should be investigated to best handle the trade off between user time and problem-solving performance.

While anecdotal evidence suggests that the correctness verification step of HDGP is important to catch potential solution programs that do not actually solve the problem, this may prove less valuable with further tests. For example, if many or most HDGP runs produce a solution that passes all potential counterexamples on the first try, then producing counterexamples may not be important to the system. However, given the difficulty of evolving programs that generalize to unseen data when using hundreds of training cases [12, 26], we expect many potential solution programs will not pass the counterexample cases. Then, we can ask further: what types of potential counterexamples are most likely to be actual counterexamples, and therefore most useful to present to the user? Gathering data on this question could both improve the quality of the potential counterexamples and reduce the number of them that need to be presented to the user for verification.

ACKNOWLEDGMENTS

We would like to thank Lee Spector, Anil Kumar Saini, and Ryan Boldi for discussions that helped develop this work.

REFERENCES

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

[2] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-Based Program Synthesis. *Commun. ACM* 61, 12 (nov 2018), 84–93. https://doi.org/10.1145/3208071

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *arXiv* (Aug. 2021). http://arxiv.org/abs/2108.07732 arXiv: 2108.07732.

[4] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to write programs. In *ICLR*.

[5] Iwo Bladek and Krzysztof Krawiec. 2017. Evolutionary Program Sketching. In *EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming (LNCS, Vol. 10196)*, Mauro Castelli, James McDermott, and Lukas Sekanina (Eds.). Springer Verlag, Amsterdam, 3–18. https://doi.org/doi:10.1007/978-3-319-55696-3_1

[6] Iwo Błądek and Krzysztof Krawiec. 2019. Solving symbolic regression problems with formal constraints. In *GECCO ’19: Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, Prague, Czech Republic, 977–984.

[7] Iwo Błądek, Krzysztof Krawiec, and Jerry Swan. 2018. Counterexample-Driven Genetic Programming: Heuristic Program Synthesis from Formal Specifications. *Evolutionary Computation* 26, 3 (Fall 2018), 441–469. https://doi.org/doi:10.1162/evco_a_00228

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, Will Guss, Alex Nichol, Igor Babuschkin, Suchir Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *arXiv* (2021). http://arxiv.org/abs/2107.03374

[9] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL ’11)*. ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

[10] Thomas Helmuth and Peter Kelly. 2021. PSB2: The Second Program Synthesis Benchmark Suite. In *2021 Genetic and Evolutionary Computation Conference (GECCO ’21)*. ACM, Lille, France. https://doi.org/10.1145/3449639.3459285

[11] Thomas Helmuth and Peter Kelly. 2022. Applying Genetic Programming to PSB2: The Next Generation Program Synthesis Benchmark Suite. *Genetic Programming and Evolvable Machines* (June 2022), 375–404. https://doi.org/10.1007/s10710-022-09434-y

[12] Thomas Helmuth, Nicholas Freitag McPhee, Edward Pantridge, and Lee Spector. 2017. Improving Generalization of Evolved Programs through Automatic Simplification. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Berlin, Germany) *(GECCO ’17)*. Association for Computing Machinery, New York, NY, USA, 937–944. https://doi.org/10.1145/3071178.3071330

[13] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2018. Program Synthesis Using Uniform Mutation by Addition and Deletion. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Kyoto, Japan) *(GECCO ’18)*. ACM, New York, NY, USA, 1127–1134. https://doi.org/10.1145/3205455.3205603

[14] Thomas Helmuth and Lee Spector. 2015. General Program Synthesis Benchmark Suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation* (Madrid, Spain) *(GECCO ’15)*. Association for Computing Machinery, New York, NY, USA, 1039–1046. https://doi.org/10.1145/2739480.2754769

[15] Thomas Helmuth and Lee Spector. 2021. Problem-Solving Benefits of Down-Sampled Lexicase Selection. *Artificial Life* (09 2021), 1–21. https://doi.org/10.1162/artl_a_00341 arXiv:https://direct.mit.edu/artl/article-pdf/doi/10.1162/artl_a_00341/1960075/artl_a_00341.pdf

[16] Thomas Helmuth, Lee Spector, and James Matheson. 2015. Solving Uncompromising Problems with Lexicase Selection. *IEEE Transactions on Evolutionary Computation* 19, 5 (Oct. 2015), 630–643. https://doi.org/doi:10.1109/TEVC.2014.2362729

[17] Thomas Helmuth, Lee Spector, and Edward Pantridge. 2020. Counterexample-Driven Genetic Programming without Formal Specifications. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion (GECCO ’20)*. ACM, 239–240. https://doi.org/doi:10.1145/3377929.3389983

[18] Jose Guadalupe Hernandez, Alexander Lalejini, and Charles Ofria. 2022. *An Exploration of Exploration: Measuring the Ability of Lexicase Selection to Find Obscure Pathways to Optimality.* Springer Nature Singapore, Singapore, 83–107. https://doi.org/10.1007/978-981-16-8113-4_5

[19] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA.

[20] Krzysztof Krawiec, Iwo Błądek, and Jerry Swan. 2017. Counterexample-driven Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO ’17)*. ACM, Berlin, Germany, 953–960. https://doi.org/doi:10.1145/3071178.3071224 Best paper.

[21] Krzysztof Krawiec, Iwo Błądek, Jerry Swan, and John H. Drake. 2018. Counterexample-Driven Genetic Programming: Stochastic Synthesis of Provably Correct Programs. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)*, Jerome Lang (Ed.). International Joint Conferences on Artificial Intelligence, Stockholm, 5304–5308. https://www.ijcai.org/proceedings/2018/742

[22] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Tauman Kalai. 2013. A Machine Learning Framework for Programming by Example. *ICML* (2013), 9.

[23] Srinivas Nidhra and Jagruthi Dondeti. 2012. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)* 2, 2 (2012), 29–50.

[24] Edward Pantridge and Thomas Helmuth. 2023. Solving Novel Program Synthesis Problems with Genetic Programming using Parametric Polymorphism. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Lisbon, Portugal) *(GECCO ’23)*. Association for Computing Machinery, New York, NY, USA.

[25] Edward Pantridge, Thomas Helmuth, and Lee Spector. 2022. Functional Code Building Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Boston, Massachusetts) *(GECCO ’22)*. Association for Computing Machinery, New York, NY, USA, 1000–1008. https://doi.org/10.1145/3512290.3528866

[26] Dominik Sobania. 2021. On the Generalizability of Programs Synthesized by Grammar-Guided Genetic Programming. In *EuroGP 2021: Proceedings of the 24th European Conference on Genetic Programming (LNCS, Vol. 12691)*, Ting Hu, Nuno Lourenco, and Eric Medvet (Eds.). Springer Verlag, Virtual Event, 130–145. https://doi.org/doi:10.1007/978-3-030-72812-0_9

[27] Dominik Sobania, Martin Briesch, Philipp Röchner, and Franz Rothlauf. 2023. MTGP: Combining Metamorphic Testing and Genetic Programming. In *Genetic Programming*, Gisele Pappa, Mario Giacobini, and Zdenek Vasicek (Eds.). Springer Nature Switzerland, Cham, 324–338.

[28] Dominik Sobania, Martin Briesch, and Franz Rothlauf. 2022. Choose Your Programming Copilot: A Comparison of the Program Synthesis Performance of Github Copilot and Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Boston, Massachusetts) *(GECCO ’22)*. Association for Computing Machinery, New York, NY, USA, 1019–1027. https://doi.org/10.1145/3512290.3528700

[29] Dominik Sobania, Dirk Schweim, and Franz Rothlauf. 2023. A Comprehensive Survey on Program Synthesis With Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation* 27, 1 (2023), 82–97. https://doi.org/10.1109/TEVC.2022.3162324

[30] Armando Solar-Lezama. 2013. Program sketching. *International Journal on Software Tools for Technology Transfer* 15, 5 (2013), 475–495.

[31] Lee Spector, Jon Klein, and Maarten Keijzer. 2005. The Push3 execution stack and the evolution of control. 1689–1696. https://doi.org/10.1145/1068009.1068292