Citation: T. Helmuth and L. Spector. General Program Synthesis Benchmark Suite. In GECCO '15: Proceedings of the 17th annual conference on Genetic and evolutionary computation, pp. 1039-1046. July 2015. ACM.

General Program Synthesis Benchmark Suite

Thomas Helmuth
Computer Science
University of Massachusetts
Amherst, MA 01003
thelmuth@cs.umass.edu

Lee Spector
Cognitive Science
Hampshire College
Amherst, MA 01002
lspector@hampshire.edu

ABSTRACT

Recent interest in the development and use of non-trivial benchmark problems for genetic programming research has highlighted the scarcity of general program synthesis (also called "traditional programming") benchmark problems. We present a suite of 29 general program synthesis benchmark problems systematically selected from sources of introductory computer science programming problems. This suite is suitable for experiments with any program synthesis system driven by input/output examples. We present results from illustrative experiments using our reference implementation of the problems in the PushGP genetic programming system. The results show that the problems in the suite vary in difficulty and can be useful for assessing the capabilities of a program synthesis system.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program synthesis*

Keywords

program synthesis; genetic programming; benchmarks

1. INTRODUCTION

Several genetic programming (GP) researchers have highlighted the need for better benchmark problems to guide research in the field [9, 17, 18]. While benchmarks have been proposed, few are for general programming problems (also called "traditional" or "algorithmic" programming problems) even though this category received the second highest level of interest in a recent community survey about the need for benchmarks [17].

Automating human programming has long been a goal of GP, as articulated for example in Koza's first book [7]. The purpose of a general program synthesis benchmark is to help researchers assess the ability of a system to automate human programming. Such problems should require a range of programming techniques including the use of control flow, modularity, and large, diverse instruction sets covering multiple data types and data structures. Also, minimal sizes for solution programs should cover a range beyond what could be found using brute-force search. This contrasts with most existing benchmark problems used in GP and other program synthesis fields [6], which prescribe small, domain-specific instruction sets and assess a system's abilities only on a narrow range of programming techniques.

In this paper we present a suite of 29 general program synthesis benchmark problems, systematically selected from sources of introductory computer science programming problems. We present each problem's specifications in the form of input/output examples, making them suitable to a wide range of program synthesis techniques, including GP. While the problems are not particularly challenging for skilled human programmers, they are reasonably challenging for beginners and many are arguably too difficult for existing program synthesis systems, including GP. As textbook problems, they are not likely representative of real general program synthesis applications, yet they should prove useful for assessing progress toward this goal.

2. BENCHMARK-BASED COMPARISONS

We designate the solution of a general program synthesis problem as a program that perfectly maps inputs to the correct outputs. While one might argue that human-written software is often useful even if it has known bugs, the goal here is to pass all input/output tests. Therefore, we are not interested in programs that are only approximately correct, as might be appropriate in the context of other problems for which GP is used, such as symbolic regression. We recommend measuring performance on the problems presented here primarily in terms of success rates, quantifying how often a stochastic algorithm finds a successful program across a set of runs[1]. A more thorough argument for assessment in terms of success rates can be found in [3]. Furthermore, in order to be considered successful, a program must not only achieve zero error on all of the example data used to train the program (the "training set"), but also on a set of withheld generalization data (the "test set").

When using this benchmark suite to compare different settings within one system, we recommend limiting computation with a budget based on the maximum number of program evaluations allowed in a run. This ensures that the

[1]For deterministic synthesis algorithms other measures must be used, such as whether a correct program is found within a set period of time.

GECCO '15, July 11 - 15, 2015, Madrid, Spain

DOI: http://dx.doi.org/10.1145/2739480.2754769

methods perform similar computational work. That said, it may nonetheless be difficult to justify fine-grained numerical comparisons among diverse techniques on these problems, as they may involve qualitatively different kinds of costs and each may be parameterized in radically different ways. In many cases, the most interesting question to ask vis-a-vis a particular system on a particular problem may just be whether the system can solve the problem at all, and if so, whether it can solve it reasonably reliably. Nevertheless, we aim here to describe specifications that will allow for as much cross-system comparability as possible.

3. PROBLEM SELECTION CRITERIA

In this section we describe the criteria we used when selecting problems for the benchmark suite. Several of our criteria overlap with those described in the GP benchmarks papers [9, 17], such as being varied, relevant, realistically difficult, representation-independent, and precisely defined.

This benchmark suite is designed for systems that use example inputs and their corresponding outputs as the specifications for desired programs. In the context of GP, we call the input/output pairs *test cases* for the problem. Thus, a problem must be defined on a range of inputs that have known correct outputs; it cannot simply specify the calculation of a single value. For example, a problem that requires the program to calculate the number of prime numbers less than 1000 would not qualify, since it only has one answer; but, a similar problem that requires the program to calculate the number of prime numbers less than an input integer n would meet this requirement, since we could then provide example inputs for n and their corresponding outputs. This requirement also ensures that test cases can be generated to fill the training and test set, as required to test generalization of successful programs.

Problems in the suite should present challenges typical of real programming tasks. This criterion leads us to choose problems that call for a range of programming constructs and data types. The problems should require a variety of sizes and shapes for solution programs, not just artificially small programs.

The benchmarks should not be biased toward a particular method of synthesis; it should be possible to attempt to solve them using various GP systems as well as analytic and search-based program synthesis systems. Since systems generate programs in a variety of languages, we avoid problems that require a specific language feature or non-standard data type (such as Java objects).

We take our problems from pre-existing sources of introductory programming problems. From each source, we include all problems that meet the criteria described above, aiming to avoid biasing the selection of problems. We rejected problems from other sources that did not meet our criteria, such as the inductive programming benchmark repository[2], other program synthesis and inductive programming papers, and programming competitions.

4. PROBLEM DESCRIPTIONS

We used two sources for problems: iJava [11], an interactive textbook for introductory computer science, and IntroClass [2, 1], a set of problems originally used as benchmarks for automatic program repair. Below we describe each of these sources in further detail and present our natural language description of each problem, summarized from the original source. All problems use functional arguments as inputs besides one that requires reading input from a file. Some problems require programs to return functional outputs, where others require the program to print results.

[2]http://www.inductive-programming.org/repository.html

4.1 iJava

iJava is an interactive introductory computer science textbook that contains a number of automatically graded programming problems [11]. Many of its problems are graded by testing programs against a range of inputs, making them easy to convert into benchmark problems.

Some sets of problems in iJava meet our criteria but test similar programming techniques; for these sets, we chose one representative problem from the group, ensuring a reasonable distribution of problem requirements. Along with each problem name and description, we provide the question or project number associated with the problem in iJava 3.1.

1. **Number IO (Q 3.5.1)** Given an integer and a float, print their sum.

2. **Small or Large (Q 4.6.3)** Given an integer n, print "small" if $n < 1000$ and "large" if $n \geq 2000$ (and nothing if $1000 \leq n < 2000$).

3. **For Loop Index (Q 4.11.7)** Given 3 integer inputs $start, end$, and $step$, print the integers in the sequence

$$n_0 = start$$
$$n_i = n_{i-1} + step$$

   for each $n_i < end$, each on their own line.

4. **Compare String Lengths (Q 4.11.13)** Given three strings $n1$, $n2$, and $n3$, return true if $length(n1) < length(n2) < length(n3)$, and false otherwise.

5. **Double Letters (P 4.1)** Given a string, print the string, doubling every letter character, and tripling every exclamation point. All other non-alphabetic and non-exclamation characters should be printed a single time each.

6. **Collatz Numbers (P 4.2)** Given an integer, find the number of terms in the Collatz (hailstone) sequence starting from that integer.

7. **Replace Space with Newline (P 4.3)** Given a string input, print the string, replacing spaces with newlines. Also, return the integer count of the non-whitespace characters. The input string will not have tabs or newlines.

8. **String Differences (P 4.4)** Given 2 strings (without whitespace) as input, find the indices at which the strings have different characters, stopping at the end of the shorter one. For each such index, print a line containing the index as well as the character in each string. For example, if the strings are "dealer" and "dollars", the program should print:

```
1 e o
2 a l
4 e a
```

9. **Even Squares (Q 5.4.1)** Given an integer n, print all of the positive even perfect squares less than n on separate lines.

10. **Wallis Pi (P 6.4))** John Wallis gave the following infinite product that converges to $\pi/4$:

$$\frac{2}{3} \times \frac{4}{3} \times \frac{4}{5} \times \frac{6}{5} \times \frac{6}{7} \times \frac{8}{7} \times \frac{8}{9} \times \frac{10}{9} \times ...$$

Given an integer input n, compute an approximation of this product out to n terms. Results are rounded to 5 decimal places.

11. **String Lengths Backwards (Q 7.2.5)** Given a vector of strings, print the length of each string in the vector starting with the last and ending with the first.

12. **Last Index of Zero (Q 7.7.8)** Given a vector of integers, at least one of which is 0, return the index of the last occurrence of 0 in the vector.

13. **Vector Average (Q 7.7.11)** Given a vector of floats, return the average of those floats. Results are rounded to 4 decimal places.

14. **Count Odds (Q 7.7.12)** Given a vector of integers, return the number of integers that are odd, without use of a specific `even` or `odd` instruction (but allowing instructions such as `mod` and `quotient`).

15. **Mirror Image (Q 7.7.15)** Given two vectors of integers, return `true` if one vector is the reverse of the other, and `false` otherwise.

16. **Super Anagrams (P 7.3)** Given strings x and y of lowercase letters, return true if y is a super anagram of x, which is the case if every character in x is in y. To be true, y may contain extra characters, but must have at least as many copies of each character as x does.

17. **Sum of Squares (Q 8.5.4)** Given integer n, return the sum of squaring each integer in the range $[1, n]$.

18. **Vectors Summed (Q 8.7.6)** Given two equal-sized vectors of integers, return a vector of integers that contains the sum of the input vectors at each index.

19. **X-Word Lines (P 8.1)** Given an integer X and a string that can contain spaces and newlines, print the string with exactly X words per line. The last line may have fewer than X words.

20. **Pig Latin (P 8.2)** Given a string containing lowercase words separated by single spaces, print the string with each word translated to pig Latin. Specifically, if a word starts with a vowel, it should have "ay" added to its end; otherwise, the first letter is moved to the end of the word, followed by "ay".

21. **Negative To Zero (Q 9.6.8)** Given a vector of integers, return the vector where all negative integers have been replaced by 0.

22. **Scrabble Score (P 10.1)** Given a string of visible ASCII characters, return the Scrabble score for that string. Each letter has a corresponding value according to normal Scrabble rules, and non-letter characters are worth zero.

23. **Word Stats (P 10.5)** Given a file, print the number of words containing n characters for n from 1 to the length of the longest word, in the format:

```
words of length 1:  12
words of length 2:  3
words of length 3:  0
words of length 4:  5
...
```

At the end of the output, print a line that gives the number of sentences and line that gives the average sentence length using the form:

```
number of sentences:  4
average sentence length:  7.452423455
```

A word is any string of consecutive non-whitespace characters (including sentence terminators). Every file will contain at least one sentence terminator (period, exclamation point, or question mark). The average sentence length is the number of words in the file divided by the number of sentence terminator characters.

4.2 IntroClass

The set of 6 problems in the IntroClass dataset [2, 1] was designed for the purpose of benchmarking automatic software defect repair systems. As such, the authors of this dataset provide a number of buggy programs written by students trying to solve each problem, taken from students in an introductory computer science class. For the purposes of general program synthesis from scratch, we will use the problems themselves but not the accompanying buggy programs.

24. **Checksum** Given a string, convert each character in the string into its integer ASCII value, sum them, take the sum modulo 64, add the integer value of the space character, and then convert that integer back into its corresponding character (the checksum character). The program must print `Check sum is X`, where X is replaced by the correct checksum character.

25. **Digits** Given an integer, print that integer's digits each on their own line starting with the least significant digit. A negative integer should have the negative sign printed before the most significant digit.

26. **Grade** Given 5 integers, the first four represent the lower numeric thresholds for achieving an A, B, C, and D, and will be distinct and in descending order. The fifth represents the student's numeric grade. The program must print `Student has a X grade.`, where X is A, B, C, D, or F depending on the thresholds and the numeric grade.

27. **Median** Given 3 integers, print their median.

28. **Smallest** Given 4 integers, print the smallest of them.

29. **Syllables** Given a string containing symbols, spaces, digits, and lowercase letters, count the number of occurrences of vowels (a, e, i, o, u, y) in the string and print that number as X in `The number of syllables is X`.

5. SYNTHESIS SPECIFICATIONS

The natural language descriptions of the problems in Section 4 do not provide all of the information needed to apply

Table 1: For each problem, the types of the inputs and outputs, and the limits imposed on the inputs. Any printed outputs should be printed by the program to standard output. The columns Train and Test indicate the recommended sizes of the training set and test set respectively.

Name	Inputs	Outputs	Train	Test
Number IO	integer in $[-100, 100]$, float in $[-100.0, 100.0]$	printed float	25	1000
Small Or Large	integer in $[-10000, 10000]$	printed string	100	1000
For Loop Index	integers `start` and `end` in $[-500, 500]$, `step` in $[1, 10]$	printed integers	100	1000
Compare String Lengths	3 strings of length $[0, 49]$	boolean	100	1000
Double Letters	string of length $[0, 20]$	printed string	100	1000
Collatz Numbers	integer in $[1, 10000]$	integer	200	2000
Replace Space with Newline	string of length $[0, 20]$	printed string, integer	100	1000
String Differences	2 strings of length $[0, 10]$	printed string	200	2000
Even Squares	integer in $[1, 9999]$	printed string	100	1000
Wallis Pi	integer in $[1, 200]$	float	150	50
String Lengths Backwards	vector of length $[0, 50]$ of strings of length $[0, 50]$	printed string	100	1000
Last Index of Zero	vector of integers of length $[1, 50]$ with each integer in $[-50, 50]$	integer	150	1000
Vector Average	vector of floats of length $[1, 50]$ with each float in $[-1000.0, 1000.0]$	float	100	1000
Count Odds	vector of integers of length $[0, 50]$ with each integer in $[-1000, 1000]$	integer	200	2000
Mirror Image	2 vectors of integers of length $[0, 50]$ with each integer in $[-1000, 1000]$	boolean	100	1000
Super Anagrams	2 strings of length $[0, 20]$	boolean	200	2000
Sum of Squares	integer in $[1, 100]$	integer	50	50
Vectors Summed	2 vectors of integers of length $[0, 50]$ with each integer in $[-1000, 1000]$	vector of integers	150	1500
X-Word Lines	integer in $[1, 10]$, string of length $[0, 100]$	printed string	150	2000
Pig Latin	string of length $[0, 50]$	printed string	200	1000
Negative To Zero	vector of integers of length $[0, 50]$ with each integer in $[-1000, 1000]$	vector of integers	200	2000
Scrabble Score	string of length $[0, 20]$	integer	200	1000
Word Stats	file containing $[1, 100]$ chars	printed string	100	1000
Checksum	string of length $[0, 50]$	printed string	100	1000
Digits	integer in $[-9999999999, 9999999999]$	printed integers	100	1000
Grade	5 integers in $[0, 100]$	printed string	200	2000
Median	3 integers in $[-100, 100]$	printed integer	100	1000
Smallest	4 integers in $[-100, 100]$	printed integer	100	1000
Syllables	string of length $[0, 20]$	printed string	100	1000

program synthesis systems to the problems. Here we provide the needed additional information, aiming to do so in a technique-independent and system-independent way.

Table 1 presents recommendations regarding training and test data for each problem. While these are merely guidelines, and there may be good reasons to diverge from them when using different techniques or systems, adhering to these guidelines will clarify comparisons among techniques and systems. The table describes the data types of the inputs and outputs and gives reasonable ranges for program inputs.

We also provide recommendations for numbers of cases to use in the training and test sets in Table 1. For most problems, we recommend between 100 and 200 training cases, depending on the difficulty of the problem as well as the dimensionality of the input space. A few problems use fewer, either because they have limited input spaces or are simple enough to solve with fewer cases. We usually recommend using a test set ten times as large as the training set; again, there are exceptions for problems with limited input spaces. The method of producing the training and test cases is system-specific; we recommend a combination of hand-chosen edge cases with randomly generated cases, and will describe our method in more detail in Section 6.

The question of which instructions to make available for a synthesis system to use for each problem is a complex one. It is important to not cherry pick a small set of instructions that are known to be sufficient to solve a problem; such a selection may be difficult for a real-world problem, where it might not be clear which instructions will be useful. On the other hand, using all available instructions for every problem expands the search space and may make problems more difficult than necessary. We recommend a compromise between these approaches in which one first determines which data types are likely to be useful for solving the problem and then uses all instructions that operate on those data types. For example, an instruction that compares the equality of

two integers and returns a boolean would be included if the problem could potentially make use of integers and booleans. By specifying only the data type requirements for a problem, we can limit the number of instructions without cherry picking.

6. SYSTEM-SPECIFC PARAMETERS

Whereas Section 5 gave technique-independent recommendations for specifying the benchmark problems for a synthesis system, this section will give more detail about the system-specific parameters and decisions that must be made in order to implement these problems in a given program synthesis system. Here we will focus on our implementation in the PushGP genetic programming system, but we emphasize that this is just one possible approach and one possible implementation, and that the problems here could be used in any system that meets the requirements in Section 3.

PushGP evolves programs in Push, a stack-based programming language designed specifically for GP [16]. The reference implementation of our problems in PushGP can be found on GitHub[3]. In the rest of this section, we will describe some of the major decisions necessary for implementing these benchmark problems in this environment. As we do not have space to discuss every parameter and implementation choice, the technical report accompanying this paper supplements the information presented here [4].

When generating training and testing data, we use a combination of hand-picked edge cases that remain constant across runs and randomly generated inputs that vary across runs. For each problem, we specify one or more "data domains" [3], which consist of either a set of hard-coded inputs or a random input generator, as well as the number of training and test cases that should come from each domain. Details on using data domains can be found in [3], and our technical report gives our choices for data domains for each problem [4].

When using this benchmark suite with GP, we not only need the input/output cases, but also a method of measuring how well a particular program performs on each case—the *fitness function*. Many of the problems in this suite print results to standard output, and we generally treat these outputs as strings and use Levenshtein distance (a measure of string edit distance) as the fitness function. Other problems produce numeric outputs, either returned or printed; for these problems we use absolute error for fitness, parsing printed numbers when necessary and possible. Some problems produce boolean values, or are best measured by a simple binary right or wrong; here, we use a fitness of 0 for right and 1 for wrong. Finally, some problems require problem-tailored fitness functions, such as vector edit distance or string formatting requirements. We give the details of each fitness function in [4].

For some problems we found it appropriate to use multiple fitness functions per test case. For example, the Replace Space With Newline problem requires both a printed string and a returned integer. For problems like this, we produce multiple fitness values for a single case. Additionally, we find that PushGP performs better on some problems when we use more than one fitness value per case, even where not strictly necessary. For example, we found no solutions to the X-Word Lines problem when using Levenshtein distance as the only fitness function, but found solutions when adding additional fitness functions for the number of newline characters and summed errors of differences in number of words compared. When using multiple fitness values for a single training case, we treat each fitness value separately when the parent selection method requires it; in tournament selection, we simply sum all fitness values.

As discussed in Section 5, we have chosen to specify the data types relevant to each problem, and then include all instructions that use those data types in each problem's instruction set. Table 2 presents the Push data types we chose for each problem. The "exec" column signifies instructions that use Push's exec stack, which typically perform control flow manipulations such as conditionals, iteration, and subfunctions defined through tagging [15]. The "print" column includes instructions that print data to standard output, and "file input" includes a small set of file reading instructions.

Table 2 also gives the terminals used for each problem, which encompass constants and ephemeral random constants (ERCs). ERCs allow for the creation of random constants in randomly generated code during initialization and mutation. We used problem-specific ERC ranges, which can be found in the technical report [4].

We keep most of our PushGP system parameters constant across all problems, with specific details in [4]. The only significant parameters that we vary per problem are the maximum program size, the maximum number of instruction evaluations that a program may use per execution, and the maximum number of generations per run. We used a maximum of 300 generations for every problem besides Number IO, Median, and Smallest, for which we used 200 generations. Maximum program sizes varied from 200 to 1000 instructions depending on anticipated problem difficulty. We limited instruction executions to approximately 2 to 5 times the maximum program size, though for some problems that require many loops we increased this limit. By specifying the maximum generations, the population size (1000 for all of our runs), and the size of the training set (see Table 1), we also specify the program evaluation budget, which is the product of those values. In our implementation, this budget falls between 5 and 60 million program evaluations per run for every problem.

7. EXPERIMENTAL RESULTS

Whereas the relevance of a benchmark suite is determined by how well its problems reflect potential applications of the test systems, its utility is based on how well it differentiates between different approaches. We aim to include problems with a large range of difficulties, from those that can be solved reliably to those that extend beyond the abilities of current program synthesis systems. More importantly, we hope to include problems that are solved more often with some systems or settings than others, allowing us to compare their performances on these problems. In this section we present a simple experiment showing the utility of the benchmark suite presented here. This experiment compares three parent selection algorithms: tournament selection, implicit fitness sharing, and lexicase selection.

Implicit fitness sharing (IFS) is a modification of tournament selection designed to encourage diversity preservation in the population [10]. IFS selection greatly rewards individuals for solving training cases that are solved by a small

[3]http://thelmuth.github.io/GECCO_2015_Benchmarks_Materials/

Table 2: Instructions and data types used in our PushGP implementation of each problem. The column "# Instructions" reports the number of instructions, terminals, and ERCs used for each problem. The middle columns show which data types were used for each problem. For example, the Number IO problem used all instructions relevant to integers, floats, and printing. The last column lists the constants and ERCs used for the problem. Here, char constants are represented in the Clojure style, starting with a backslash, and strings are surrounded by double quotation marks. The "Problems" row simply counts how many problems use each data type. The "Instructions" row shows the number of Push instructions that primarily use each data type; some use multiple types but are only counted once.

Problem	# Instructions	exec	integer	float	boolean	char	string	vector of integers	vector of floats	vector of strings	print	file input	Terminals (besides inputs)
Number IO	50		x	x							x		integer ERC, float ERC
Small Or Large	103	x	x		x		x				x		"small", "large", integer ERC
For Loop Index	74	x	x		x						x		
Compare String Lengths	98	x	x		x		x						boolean ERC
Double Letters	132	x	x		x	x	x				x		\!
Collatz Numbers	102	x	x	x	x								0, 1, integer ERC
Replace Space with Newline	135	x	x		x	x	x				x		\space, \newline, string ERC, char ERC
String Differences	135	x	x		x	x	x				x		\space, \newline, integer ERC
Even Squares	72	x	x		x						x		
Wallis Pi	103	x	x	x	x								2 integer ERCs, 2 float ERCs
String Lengths Backwards	134	x	x		x		x			x	x		integer ERC
Last Index of Zero	101	x	x		x			x					0
Vector Average	88	x	x	x					x				
Count Odds	104	x	x		x			x					0, 1, 2, integer ERC
Mirror Image	102	x	x		x			x					boolean ERC
Super Anagrams	129	x	x		x	x	x						boolean ERC, char ERC, integer ERC
Sum of Squares	71	x	x		x								0, 1, integer ERC
Vectors Summed	68	x	x					x					[], integer ERC
X-Word Lines	134	x	x		x	x	x				x		\newline, \space
Pig Latin	141	x	x		x	x	x				x		"ay", \space, \a, \e, \i, \o, \u, "aeiou", string ERC, char ERC
Negative To Zero	102	x	x		x			x					0, []
Scrabble Score	158	x	x		x	x	x	x					vector containing Scrabble values (indexed by ASCII values)
Word Stats	281	x	x	x	x	x	x	x	x	x	x	x	\., \?, \!, \space, \tab, \newline, [], "words of length ", ": ", "number of sentences: ", "average sentence length: ", integer ERC
Checksum	136	x	x		x	x	x				x		"Check sum is ", \space, 64, integer ERC, char ERC
Digits	133	x	x		x	x	x				x		\newline, integer ERC [-10, 10]
Grade	112	x	x		x		x				x		"Student has a ", " grade.", "A", "B", "C", "D", "F", integer ERC
Median	75	x	x		x						x		integer ERC
Smallest	76	x	x		x						x		integer ERC
Syllables	141	x	x		x	x	x				x		"The number of syllables is ", "aeiouy", \a, \e, \i, \o, \u, \y, char ERC, string ERC
Problems		28	29	5	26	11	15	7	2	2	17	1	
Instructions		28	28	31	19	17	39	31	31	31	10	4	

Table 3: Number of successful runs out of 100 for each setting, where "Tourn" is size 7 tournament selection, "IFS" is implicit fitness sharing with size 7 tournaments, and "Lex" is lexicase selection. For each problem, underline indicates significant improvement over the other two selection methods at $p < 0.05$ based on a pairwise chi-square test with Holm correction [13], or a pairwise Fisher's exact test with Holm correction if any number of successes is below 5 [12]. The "Size" column indicates the smallest size of any simplified solution program.

Problem	Tourn	IFS	Lex	Size
Number IO	68	72	98	5
Small Or Large	3	3	5	27
For Loop Index	0	0	1	21
Compare String Lengths	3	6	7	11
Double Letters	0	0	6	20
Collatz Numbers	0	0	0	
Replace Space with Newline	8	16	51	9
String Differences	0	0	0	
Even Squares	0	0	2	37
Wallis Pi	0	0	0	
String Lengths Backwards	7	10	66	9
Last Index of Zero	8	4	21	5
Vector Average	14	13	16	7
Count Odds	0	0	8	7
Mirror Image	46	64	78	4
Super Anagrams	0	0	0	
Sum of Squares	2	0	6	7
Vectors Summed	0	0	1	11
X-Word Lines	0	0	8	15
Pig Latin	0	0	0	
Negative To Zero	10	8	45	8
Scrabble Score	0	0	2	14
Word Stats	0	0	0	
Checksum	0	0	0	
Digits	0	1	7	20
Grade	0	0	4	52
Median	7	43	45	10
Smallest	75	98	81	8
Syllables	1	7	18	14
Problems Solved	13	13	22	

fraction of the population, and gives less reward for solving cases that are solved by more of the population. Most of the problems here produce non-binary error values, for which we use the non-binary adaptation of IFS found in [8]. As required by this method, we normalize error values to $[0, 1]$ by dividing each error by a maximum allowed error value, which differs per problem based on the fitness function.

Lexicase selection [5], unlike tournament selection and IFS, does not base selection on a single fitness value. Instead, it uses a random ordering of the training set to select individuals that perform as well as possible on a subset of the cases even if they exhibit poor performance on other cases. Lexicase selection has been shown to improve the performance of a GP system on a variety of problems [5, 3].

Table 3 gives the results of our parent selection experiment. Over the 29 problems, PushGP with lexicase selection produced at least one successful run on nine more problems than either tournament selection or IFS. Additionally, there were 8 problems where lexicase selection achieved a significantly higher number of successful runs than the other two, where IFS showed significant improvement on just one problem and tournament selection none. These results strongly indicate the utility of lexicase selection for general program synthesis problems.

The data in Table 3 only reflect solutions that generalize by achieving zero error on the unseen test set. Some problems seem to lend themselves to generalization more than others; for example, PushGP using lexicase selection found 14 programs with zero error on the training set for the Super Anagrams problem, none of which generalized to the test set. For lexicase selection, five problems resulted in 20 or more runs that passed the training set that did not generalize (Small Or Large, Compare String Lengths, Last Index of Zero, Negative To Zero, and Median), and five problems had between 10 and 20 runs that did not generalize (String Lengths Backwards, Mirror Image, Digits, Smallest, and Super Anagrams). These 10 problems show an important area for future study: how to evolve programs that generalize to unseen data for general program synthesis problems. Among these problems are the only five in the suite that give a correct/incorrect binary error as fitness in our implementation: Compare String Lengths, Mirror Image, Super Anagrams, Median, and Smallest. This shows the difficulty of evolving general programs based entirely on correctness of output, and suggests that these problems might be better tackled if they can be transformed into problems with more informative error functions.

With regards to the problems themselves, this experiment illustrates the ability of this benchmark suite to provide useful comparisons between multiple systems or parameter settings. By looking at the number of problems solved by each technique, and how often each technique showed significant improvements over the others, we can clearly see that lexicase selection increases PushGP's ability to solve general program synthesis problems compared to tournament selection and IFS. The main goal of a benchmark suite is to support this type of experiment. Additionally, some problems in the suite were solved frequently by each system, whereas others were solved infrequently or not at all. This range of difficulties permits the suite to be useful for a variety of experiments, and allows it to remain relevant as program synthesis systems improve.

Of the seven problems on which PushGP found no generalizing solution, most are not surprising in that they involve extensive use of multiple programming constructs, the linking of many distinct steps, or a deceptive fitness space where fitness improvements do not lead toward perfect programs. We have written solutions to each of the unsolved problems by hand to ensure that each problem is solvable within the constraints we put on the system and instruction set.

The last column in Table 3 gives the size (in instructions) of the smallest simplified solution program. Here, we've used post-run simplification to automatically reduce the sizes of solution programs without changing their semantics on the training data [14]. While this hill-climbing simplification is not guaranteed to find the smallest semantically equivalent program, it reliably removes excess code, leaving the core functionality of the program [14]. The simplified program sizes present a reasonable proxy for the smallest solution program for each problem (using our instruction sets). While

some problems can be solved with programs containing fewer than 10 instructions, none would likely be found using brute-force search over our instruction sets within the number of program evaluations allowed here. Searching over size 5 programs using the Number IO instruction set would require evaluating over 7 billion programs, much more than the 5 million we used in our GP runs. Other problems have smallest known solutions of over 20 instructions using instruction sets with more than 100 instructions, to our knowledge beyond the reach of all other program synthesis systems.

8. CONCLUSIONS

We have presented a suite of 29 general program synthesis benchmark problems, systematically selected from sources of introductory computer science programming problems. Through exposition and experimentation, we have demonstrated the potential utility of this suite to assess the capabilities of program synthesis systems. We expect that the application of this suite can help advance multiple fields of automatic program synthesis, including genetic programming, that have long employed simple benchmark problems not attuned to potential real-world applications.

9. ACKNOWLEDGMENTS

Thanks to the members of the Hampshire College Computational Intelligence Lab, Nicholas Freitag McPhee, Yuriy Brun, and David Jensen for discussions that helped to improve the work described in this paper, to Josiah Erikson for systems support, and to Hampshire College for support for the Hampshire College Institute for Computational Intelligence. This material is based upon work supported by the National Science Foundation under Grants No. 1017817, 1129139, and 1331283. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

10. REFERENCES

[1] Y. Brun, E. Barr, M. Xiao, C. Le Goues, and P. Devanbu. Evolution vs. intelligent design in program patching. Technical Report https://escholarship.org/uc/item/3z8926ks, UC Davis: College of Engineering, 2013.

[2] C. L. Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The ManyBugs and IntroClass benchmarks for automated program repair. *IEEE Transactions on Software Engineering*. Under Review.

[3] T. Helmuth and L. Spector. Word count as a traditional programming benchmark problem for genetic programming. In *GECCO '14: Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 919–926, Vancouver, BC, Canada, 12-16 July 2014. ACM.

[4] T. Helmuth and L. Spector. Detailed problem descriptions for general program synthesis benchmark suite. Technical Report UM-CS-2015-006, School of Computer Science, University of Massachusetts Amherst, 2015.

[5] T. Helmuth, L. Spector, and J. Matheson. Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation*, 2014.

[6] M. Hofmann, E. Kitzelmann, and U. Schmid. A unifying framework for analysis and evaluation of inductive programming systems. In *Proceedings of the Second Conference on Artificial General Intelligence*, pages 55–60. Citeseer, 2009.

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

[8] K. Krawiec and M. Nawrocki. Implicit fitness sharing for evolutionary synthesis of license plate detectors. In *Applications of Evolutionary Computing, EvoApplications 2012*, volume 7835 of *Lecture Notes in Computer Science*, pages 376–386, Vienna, Austria, 3-5 Apr. 2013. Springer.

[9] J. McDermott, D. R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong, and U.-M. O'Reilly. Genetic programming needs better benchmarks. In *GECCO '12: Proceedings of the Genetic and evolutionary computation conference*, pages 791–798, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[10] R. I. McKay. Fitness sharing in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 435–442, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[11] R. Moll. iJava. http://ijava.cs.umass.edu/index.html, 2014. Edition 3.1. Online; accessed September 2015.

[12] M. Nakazawa. *fmsb: Functions for medical statistics book with some demographic data*, 2014. R package.

[13] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. ISBN 3-900051-07-0.

[14] L. Spector and T. Helmuth. Effective simplification of evolved Push programs using a simple, stochastic hill-climber. In *GECCO Companion '14*, pages 147–148, Vancouver, BC, Canada, 12-16 July 2014. ACM.

[15] L. Spector, B. Martin, K. Harrington, and T. Helmuth. Tag-based modules in genetic programming. In *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1419–1426, Dublin, Ireland, 12-16 July 2011. ACM.

[16] L. Spector and A. Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, Mar. 2002.

[17] D. R. White, J. Mcdermott, M. Castelli, L. Manzoni, B. W. Goldman, G. Kronberger, W. Jaśkowski, U.-M. O'Reilly, and S. Luke. Better GP benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines*, 14(1):3–29, Mar. 2013.

[18] J. Woodward, S. Martin, and J. Swan. Benchmarks that matter for genetic programming. In *GECCO 2014 4th workshop on evolutionary computation for the automated design of algorithms*, pages 1397–1404, Vancouver, BC, Canada, 12-16 July 2014. ACM.