Detailed Problem Descriptions for General Program Synthesis Benchmark Suite

Technical Report UM-CS-2015-006

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. This technical report provides sufficient detail of the problems and our reference implementation for researchers to implement and attempt to solve these problems in other synthesis systems. 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.

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 [11, 25, 26]. 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 [25].

Automating human programming has long been a goal of GP, as articulated for example in Koza's first book [9]. 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. 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 [7], which prescribe small, domain-specific instruction sets and assess a system's abilities only on a narrow range of programming techniques.

In this technical report 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.

This technical report expands on the initial publication of these benchmark problems [5] by providing additional details of the implementation and experimental results. These additions include a list of instructions used in the evolving programs, detailed descriptions of the inputs and fitness functions used for training and testing of each problem, details of the GP parameters used in our experiments, and an expanded statistical analysis of our results. This expansion should provide sufficient detail to replicate our experiments or implement the problems in other systems.

2. BENCHMARK-BASED COMPARISONS

In the context of general program synthesis, we call a program a "solution" only if it perfectly maps all inputs to 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 [4]. 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

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

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 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 [11, 25], 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

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

We used two sources for problems: iJava [14, 13], 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.

4.1 iJava

iJava is an interactive introductory computer science textbook that contains a number of automatically graded programming problems [14, 13]. 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 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 cases, 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 [18, 24, 22]. 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.

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

6.1 Training and Test Data

When generating training and test 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" [4], 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.

In order to facilitate the creation of training and test data, we designed a general system for automatic data generation based on data domains. A "data domain" D is a set of program inputs described by either a list of inputs or a random input generator function. The list (`"hi"`, `"hello"`, `"howdy"`, `"hey"`) and a function that returns `"zoo"` followed by 0 to 17 random lowercase letters are examples of data domains, where the former is an enumerated list of four inputs and the latter is random input generator function of strings at most 20 characters long that start with the substring `"zoo"`. Along with each data domain D, the user must provide the integers $train(D)$ and $test(D)$ that indicate the number of training and test cases respectively to generate from D.

To generate training and test data from a set of data domains $\{D_1, D_2, ..., D_n\}$, we simply take each domain and create the required number of cases. If the domain D_i is an enumerated list of inputs, we select $train(D_i)$ and $test(D_i)$ of them at random, without replacement within the training cases or test cases. If the domain is described by a random input generator, we run it $train(D_i)$ and $test(D_i)$ times (with replacement) to create the data. This automatic data generation system allows for the generation of training and test cases for a wide range of problems.

Tables 2, 3, and 4 present detailed descriptions of the data domains we used to generate training and test data for each benchmark problem. The table has two types of data domains: hard coded lists of inputs (HC) and random input generators (RNGs). For HC data domains, we give the list of inputs; for RNGs, we describe the generator. Unless stated otherwise, RNGs have the following properties: ranges for inputs are given in Table 1. For integer and float RNGs, inputs are sampled uniformly across the given range; for string RNGs, lengths are sampled uniformly between 1 and the max length given in Table 1, and characters are distributed uniformly across visible ASCII characters along with space, newline, and tab. If a HC domain is specified by a range such as [40, 50], it includes every integer in the range inclusive. For HC string inputs, we use `"␣"` for the space character, `"\t"` for tab, and `"\n"` for newline.

6.2 Fitness Functions

When using this benchmark suite with GP, we not only need the training and test 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 possible. Some problems produce boolean values, or are best measured by a simple binary

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

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 Table 5.

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 after adding additional fitness functions calculating the number of newline characters and summed errors of differences in number of words on each line. 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.

Since these problems aim to test how well a system would perform on real program synthesis applications, we try to keep fitness functions simple to resemble those that might be used by practitioners. For the majority of the problems in this suite (19 out of 29), we use a basic fitness function based on the type of the output. The basic fitness function for integers and floats is absolute numeric distance; for booleans it is right/wrong; for printed strings it is either right/wrong or Levenshtein distance. Most of the problems for which we use problem-specific fitness functions require a printed string as output, and attempt to parse that string to provide extra information based on the problem's expected output. We try to not put too much knowledge about a problem into the problem-specific fitness functions, but in-

Table 2: Data domains for each benchmark problem (part 1).

Name	Type	Domain	Train	Test
Number IO	RNG	integer, float	25	1000
Small Or Large	HC	-10000, 0, 980, 1020, 1980, 2020, 10000, [995, 1004], [1995, 2004]	27	0
	HC	integers in ranges [980, 1019] and [1980, 2019]	0	80
	RNG	integer	73	920
For Loop Index	RNG	integers: $\texttt{start} < 0 < \texttt{end}$, $\texttt{start} + (20 \times \texttt{step}) + 1 > \texttt{end}$	10	100
	RNG	integers: $\texttt{start} < \texttt{end}$, $\texttt{start} + (20 \times \texttt{step}) + 1 > \texttt{end}$	90	900
Compare String Lengths	HC	triplet ("", "", "")	1	0
	HC	all permutations of ("", "a", "bc")	6	0
	RNG	(repeated twice) all permutations of 2 empty strings and a string	6	0
	RNG	(repeated 3 times) all permutations of 2 copies of a string and another string	9	0
	RNG	random string repeated 3 times	3	100
	RNG	3 strings in sorted length order	25	200
	RNG	3 strings	50	700
Double Letters	HC	"", "A", "!", "␣", "*", "\t", "\n", "B\n", "\n\n", "CD", "ef", "!!", "q!", "!R", "!#", "@!", "!F!", "T$L", "4ps", "q\t ", "!!!", "i:!i:!i:!i:!i", "88888888888888888888", "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣", "ssssssssssssssssssss", "!!!!!!!!!!!!!!!!!!!!", "Ha␣Ha␣Ha␣Ha␣Ha␣Ha␣Ha", "x\ny!x\ny!x\ny!x\ny!x\ny!", "1!1!1!1!1!1!1!1!1!1!", "G5G5G5G5G5G5G5G5G5G5", ">_=]>_=]>_=]>_=]>_=]", "k!!k!!k!!k!!k!!k!!k!"	32	0
	RNG	string	68	1000
Collatz Numbers	HC	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 6171, 6943, 7963, 9257, 9999, 10000	16	0
	RNG	integer	184	2000
Replace Space With Newline	HC	"", "A", "*", "␣", "s", "B␣", "␣␣", "␣D", "ef", "!!", "␣F␣", "T␣L", "4ps", "q␣␣", "␣␣␣", "␣␣e", "hi␣", "␣␣$␣␣", "␣␣␣␣␣␣9", "i␣!i␣!i␣!i␣!i", "88888888888888888888", "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣", "ssssssssssssssssssss", "1␣1␣1␣1␣1␣1␣1␣1␣1␣1␣", "␣v␣v␣v␣v␣v␣v␣v␣v␣v␣v", "Ha␣Ha␣Ha␣Ha␣Ha␣Ha␣Ha", "x␣y!x␣y!x␣y!x␣y!x␣y!", "G5G5G5G5G5G5G5G5G5G5", ">_=]>_=]>_=]>_=]>_=]", "^_^␣^_^␣^_^␣^_^␣^_^␣"	30	0
	RNG	string (with ␣ as 20% of characters)	70	1000
String Differences	HC	pairs of strings: ("" ""), ("" "hi"), ("ThereWorld" ""), ("A" "A"), ("B" "C"), ("&" "#"), ("4" "456789"), ("rat" "hat"), ("new" "net"), ("big" "bag"), ("STOP" "SIGN"), ("abcde" "a"), ("abcde" "abcde"), ("abcde" "edcba"), ("2n" "nn"), ("hi" "zipper"), ("dealer" "dollars"), ("nacho" "cheese"), ("loud" "louder"), ("qwertyuiop" "asdfghjkl;"), ("LALALALALA" "LLLLLLLLLL"), ("!!!!!!" ".?."), ("9r2334" "9223d4r"), ("WellWell" "wellwell"), ("TakeThat!" "TAKETHAT!!"), ("CHOCOLATE^" "CHOCOLATE^"), ("sssssssss" "~~~~~~~~~~~"), (">_=]>_=]>_" "q_q_q_q_q_"), ("()()()()()" "pp)pp)pp)p"), ("HaHaHaHaHa" "HiHiHiHiHi")	30	0
	RNG	pair of strings, length > 1	170	0
	RNG	pair of strings	0	2000
Even Squares	HC	1, 2, 3, 4, 5, 6, 15, 16, 17, 18, 36, 37, 64, 65, 9600, 9700, 9999	17	0
	RNG	integer	83	1000
Wallis Pi	HC	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 198, 199, 200	15	0
	RNG	integer	135	50
String Lengths Backwards	HC	vector of strings: [], [""], ["" ""], ["" "" ""], ["" "" "" "" "" "" "" "" "" ""], ["abcde"], ["1"], ["abc" "hi␣there"], ["!@#" "\n\n\t\t" "5552\na␣r"], ["tt" "333" "1" "ccc"]	10	0
	RNG	vector of strings	90	1000

Table 3: Data domains for each benchmark problem (part 2).

Name	Type	Domain	Train	Test
Last Index of Zero	HC	vector of integers: [0 1], [1 0], [7 0], [0 8], [0 -1], [-1 0], [-7 0], [0 -8]	8	0
	HC	every vector of zeros of length between 1 and 50	30	20
	HC	all permutations of vector [0 5 -8 9]	20	4
	HC	all permutations of vector [0 0 -8 9]	10	2
	HC	all permutations of vector [0 0 0 9]	4	0
	RNG	vector of integers with at least one 0	78	974
Vector Average	HC	vector of floats: [0.0], [100.0], [-100.0], [2.0 129.0], [0.12345 -4.678], [999.99 74.113]	6	0
	RNG	length 50 vector of floats	4	50
	RNG	vector of floats	90	950
Count Odds	HC	vector of integers: [], [-10], [-9], [-8], [-7], [-6], [-5], [-4], [-3], [-2], [-1], [0], [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [-947], [-450], [303], [886], [0 0], [0 1], [7 1], [-9 -1], [-11 40], [944 77]	32	0
	RNG	vector of integers, all odd	9	100
	RNG	vector of integers, all even	9	100
	RNG	vector of integers, random probability of odd per vector	150	1800
Mirror Image	HC	pair of vectors of integers: ([] []), ([1] [1]), ([0] [1]), ([1] [0]), ([-44] [16]), ([-13] [-12]), ([2 1] [1 2]), ([0 1] [1 1]), ([0 7] [7 0]), ([5 8] [5 8]), ([34 12] [34 12]), ([456 456] [456 456]), ([40 831] [-431 -680]), ([1 2 1] [1 2 1]), ([1 2 3 4 5 4 3 2 1] [1 2 3 4 5 4 3 2 1]), ([45 99 0 12 44 7 7 44 12 0 99 45] [45 99 0 12 44 7 7 44 12 0 99 45]), ([24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24] [24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]), ([33 45 -941] [33 45 -941]), ([33 -941 45] [33 45 -941]), ([45 33 -941] [33 45 -941]), ([45 -941 33] [33 45 -941]), ([-941 33 45] [33 45 -941]), ([-941 45 33] [33 45 -941])	23	0
	RNG	pair of vectors of integers that are mirror image	37	500
	RNG	pair of equal vectors of integers	10	100
	RNG	pair of vectors of integers that are close to mirror image, but have a few elements changed	20	200
	RNG	pair of vectors of integers	10	200
Super Anagrams	HC	pair of strings: ("" ""), ("h" ""), ("" "i"), ("a" "a"), ("c" "b"), ("nn" "n"), ("c" "abcde"), ("abcde" "c"), ("mnbvccxz" "r"), ("aabc" "abc"), ("abcde" "aabc"), ("edcba" "abcde"), ("moo" "mo"), ("mo" "moo"), ("though" "tree"), ("zipper" "rip"), ("rip" "flipper"), ("zipper" "hi"), ("dollars" "dealer"), ("louder" "loud"), ("ccccc" "ccccccccc"), ("oldwestaction" "clinteastwood"), ("ldwestaction" "clinteastwood"), ("verificationcomplete" "verificationcomplete"), ("hhhhhhhhhhaaaaaaaaaa" "hahahahahahahahahaha"), ("aahhhh" "hahahahahahahahahaha"), ("qwqeqrqtqyquqiqoqpqs" ""), ("qazwsxedcrfvtgbyhnuj" "wxyz"), ("gggffggfefeededddd" "dddeeefffgggg"), ("dddeeefffgggg" "gggffggfefeededddd")	30	0
	RNG	pair of strings, chosen to be close to (or actually) super anagrams	170	2000
Sum of Squares	HC	1, 2, 3, 4, 5, 100	6	0
	RNG	integer	44	50
Vectors Summed	HC	pair of vectors of integers: ([] []), ([0] [0]), ([10] [0]), ([5] [3]), ([-9] [7]), ([0 0] [0 0]), ([-4 2] [0 1]), ([-3 0] [-1 0]), ([-323 49] [-90 -6])	10	0
	RNG	pair of length 1 vectors of integers	5	0
	RNG	pair of length 50 vectors of integers	10	100
	RNG	pairs of vectors of integers	125	1400
X-Word Lines	HC	pair of strings and integers (too long to print, see reference implementation for details)	46	0
	RNG	pair of strings and integers	104	2000

Table 4: Data domains for each benchmark problem (part 3).

Name	Type	Domain	Train	Test
Pig Latin	HC	"", "a", "b", "c", "d", "e", "i", "m", "o", "u", "y", "z", "hello", "there", "world", "eat", "apple", "yellow", "orange", "umbrella", "ouch", "in", "hello␣there␣world", "out␣at␣the␣plate", "nap␣time␣on␣planets", "supercalifragilistic", "expialidocious", "uuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuu", "ssssssssssssssssssssssssssssssssssssssssssssssssss", "w␣w␣w␣w␣w␣w␣w␣w␣w␣w␣w␣w␣w␣w␣w␣w␣w␣w␣w␣w␣w␣w␣w␣w␣w", "e␣e␣e␣e␣e␣e␣e␣e␣e␣e␣e␣e␣e␣e␣e␣e␣e␣e␣e␣e␣e␣e␣e␣e␣e", "ha␣ha␣ha␣ha␣ha␣ha␣ha␣ha␣ha␣ha␣ha␣ha␣ha␣ha␣ha␣ha␣ha", "x␣y␣x␣y␣x␣y␣x␣y␣x␣y␣x␣y␣x␣y␣x␣y␣x␣y␣x␣y␣x␣y␣x␣y␣x"	33	0
	RNG	string	167	1000
Negative To Zero	HC	vector of integers: [], [-10], [-1], [0], [1], [10], [0 0], [0 1], [-1 0], [-90 -6], [-16 33], [412 111]	12	0
	RNG	length 1 vector of integers	5	0
	RNG	vector of negative integers	9	100
	RNG	vector of positive integers	9	100
	RNG	vector of integers	165	1800
Scrabble Score	HC	each single lowercase letter	26	0
	HC	each single uppercase letter	0	26
	HC	"", "*", "␣", "Q␣", "zx", "␣Dw", "ef", "!!", "␣F@", "ydp", "4ps", "abcdefghijklmnopqrst", "ghijklmnopqrstuvwxyz", "zxyzxyqQQZXYqqjjawp", "h␣w␣h␣j##r##r\␣n+JJL", "i␣!i␣!i␣!i␣!i", "QQQQQQQQQQQQQQQQQQQQ", "$$$$$$$$$$$$$$$$$$$$", "wwwwwwwwwwwwwwwwwwww", "1␣1␣1␣1␣1␣1␣1␣1␣1␣1␣", "␣v␣v␣v␣v␣v␣v␣v␣v␣v␣v", "Ha␣Ha␣Ha␣Ha␣Ha␣Ha␣Ha", "x␣y!x␣y!x␣y!x␣y!x␣y!", "G5G5G5G5G5G5G5G5G5G5"	24	0
	RNG	string with at least 2 characters	150	974
Word Stats	HC	string (too long to print, see reference implementation for details)	36	0
	RNG	string containing at least one sentence terminator	64	1000
Checksum	HC	string (too long to print, see reference implementation for details)	12	0
	RNG	string	88	1000
Digits	HC	-9495969798, -20008000, -777777, -9876, -482, -97, -20, 0, 19, 620, 24068, 512000, 8313227, 30000000, 9998887776	15	0
	RNG	integer taken from logarithmic distribution	85	1000
Grade	HC	quintuplet of integers: (80 70 60 50 85), (80 70 60 50 80), (80 70 60 50 79), (80 70 60 50 75), (80 70 60 50 70), (80 70 60 50 69), (80 70 60 50 65), (80 70 60 50 60), (80 70 60 50 59), (80 70 60 50 55), (80 70 60 50 50), (80 70 60 50 49), (80 70 60 50 45), (90 80 70 60 100), (90 80 70 60 0), (4 3 2 1 5), (4 3 2 1 4), (4 3 2 1 3), (4 3 2 1 2), (4 3 2 1 1), (4 3 2 1 0), (100 99 98 97 100), (100 99 98 97 99), (100 99 98 97 98), (100 99 98 97 97), (100 99 98 97 96), (98 48 27 3 55), (98 48 27 3 14), (98 48 27 3 1), (45 30 27 0 1), (45 30 27 0 0), (48 46 44 42 40), (48 46 44 42 41), (48 46 44 42 42), (48 46 44 42 43), (48 46 44 42 44), (48 46 44 42 45), (48 46 44 42 46), (48 46 44 42 47), (48 46 44 42 48), (48 46 44 42 49)	41	0
	RNG	quintuplet of integers, with the first four distinct and decreasing	159	2000
Median	RNG	triplet of integers, all equal	10	100
	RNG	triplet of integers, two of three equal	30	300
	RNG	triplet of integers	60	600
Smallest	HC	quadruplet of integers: (0 0 0 0), (-44 -44 -7 -13), (0 4 -99 -33), (-22 -22 -22 -22), (99 100 99 100)	5	0
	RNG	quadruplet of integers, all equal	5	100
	RNG	quadruplet of integers, three of four equal	10	100
	RNG	quadruplet of integers in range [0, 100]	20	200
	RNG	quadruplet of integers	60	600
Syllables	HC	"", "a", "v", "4", "o", "␣", "aei", "ouy", "chf", "quite", "a␣r␣e9j>", "you␣are␣many␣yay␣yea", "ssssssssssssssssssss", "oooooooooooooooooooo", "wi␣wi␣wi␣wi␣wi␣wi␣wi", "x␣y␣x␣y␣x␣y␣x␣y␣x␣y␣", "eioyeioyeioyeioyeioy"	17	0
	RNG	string (with each char having 20% chance of being a vowel)	83	1000

Table 5: The fitness functions used for each problem. For problems that require the program to print, we usually use Levenshtein distance on the printed string and the correct output. Additionally, we add a second fitness function to many problems by parsing part or all of a printed string as a different data type and comparing to the correct output. For example, for the Number IO problem, if the printed output can be parsed as a float, it is done so and used as a float error. For such problems, an output that cannot be parsed correctly receives a penalty error.

Problem	Fitness Function
Number IO	printed string Levenshtein distance; printed float error
Small Or Large	printed string Levenshtein distance
For Loop Index	printed string Levenshtein distance
Compare String Lengths	boolean error
Double Letters	printed string Levenshtein distance
Collatz Numbers	integer error
Replace Space with Newline	printed string Levenshtein distance; integer error
String Differences	printed string Levenshtein distance; numeric difference in number of lines with correct format
Even Squares	printed string Levenshtein distance; numeric difference in number of lines with correct format; printed integer error on each line
Wallis Pi	float error; Levenshtein distance of string version of float
String Lengths Backwards	printed string Levenshtein distance
Last Index of Zero	integer error
Vector Average	float error
Count Odds	integer error
Mirror Image	boolean error
Super Anagrams	boolean error
Sum of Squares	integer error
Vectors Summed	integer error at each position in vector
X-Word Lines	printed string Levenshtein distance; integer error for number of newlines; numeric difference in correct words on each line summed over lines
Pig Latin	printed string Levenshtein distance
Negative To Zero	integer vector Levenshtein distance
Scrabble Score	integer error
Word Stats	printed string Levenshtein distance; integer error for printed number of sentences; float error for printed average sentence length
Checksum	printed string Levenshtein distance; for last printed char in string, ASCII value error
Digits	printed string Levenshtein distance
Grade	printed string Levenshtein distance; printed char error for grade char
Median	printed string right/wrong
Smallest	printed string right/wrong
Syllables	printed string Levenshtein distance; printed integer error

stead choose functions that are fairly obvious based on the problem descriptions.

6.3 Instruction Sets

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 6 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 [23]. The "print" column includes instructions that print data to standard output, and "file input" includes a small set of file reading instructions.

Table 6 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 Table 7. These ranges were selected as seemed appropriate for each problem; we do not anticipate that varying from these ranges would have significant impact on results.

Tables 8 and 9 show every Push instruction used in our experiments, and the data types that they require. For example, the `string_containschar` instruction requires that the boolean, char, and string data types be used for a problem in order to be included; this is because it must use a string and a char as inputs, and returns a boolean of whether the input string contains the input char. These tables are intended to give an idea of the scope and complexity of the instructions used in our experiments. Attempting the problems in another system would obviously require a different set of instructions specific to the programming language of the search. While we would expect such a system to use different instructions, we would also expect similar numbers of instructions that are not cherry-picked for the individual problems.

Table 6: 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; ERC ranges are given in Table 7. 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 7: ERC ranges used in our problems. For char and string ERCs, "visible chars" indicates all visible ASCII characters plus space, newline, and tab.

Problem	ERC Ranges
Number IO	integer ERC $[-100, 100]$, float ERC $[-100.0, 100.0)$
Small Or Large	integer ERC $[-10000, 10000]$
For Loop Index	
Compare String Lengths	boolean ERC $[true, false]$
Double Letters	
Collatz Numbers	integer ERC $[-100, 100]$
Replace Space with Newline	char ERC (visible chars), string ERC (lowercase letters and spaces, with space having 20% chance at each character)
String Differences	integer ERC $[-10, 10]$
Even Squares	
Wallis Pi	integer ERC $[-10, 10]$, integer ERC $[-500, 500]$, float ERC $[-500.0, 500.0)$
String Lengths Backwards	integer ERC $[-100, 100]$
Last Index of Zero	integer ERC $[-50, 50]$
Vector Average	
Count Odds	integer ERC $[-1000, 1000]$
Mirror Image	boolean ERC $[true, false]$
Super Anagrams	boolean ERC $[true, false]$, integer ERC $[-1000, 1000]$, char ERC (visible chars)
Sum of Squares	integer ERC $[-100, 100]$
Vectors Summed	integer ERC $[-1000, 1000]$
X-Word Lines	
Pig Latin	char ERC (visible chars), string ERC (lowercase letters and spaces, with space having 20% chance at each character)
Negative To Zero	
Scrabble Score	
Word Stats	integer ERC $[-100, 100]$
Checksum	integer ERC $[-128, 128]$, char ERC (visible chars)
Digits	integer ERC $[-10, 10]$
Grade	integer ERC $[0, 100]$
Median	integer ERC $[-100, 100]$
Smallest	integer ERC $[-100, 100]$
Syllables	char ERC (visible chars), string ERC (lowercase letters, spaces, digits, and symbols, with vowels having 20% chance at each character)

6.4 GP Parameters

In the most recent version of PushGP, genomes are represented by flat sequences of instructions that may have one or more epigenetic markers attached to each instruction. In this work, we use the default epigenetic markers, which only include a marker that tells how many pairs of parentheses to close after each instruction when translating the genome into a Push program. We initialize genomes by selecting a genome size uniformly between 0 and the maximum initial genome size, which for these runs we set to half of the maximum genome size. Each gene is composed of an instruction taken uniformly from the instruction set, as well as an epigenetic marker for parentheses ranging from 0 to 3, weighted toward 0.

In our experiment, we keep most of our PushGP system parameters constant across all problems, with specific details in Table 10. The genetic operators in our system work on the linear Push genomes as described in Table 10. The only significant PushGP 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 varied these parameters based on expected problem difficulty and expected program size necessary to solve each problem; the exact values are given in Table 11. 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.

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 tourna-

Table 8: Push data types and instructions used in our experiments. For each combination of data types listed in the first column, we list all of the Push instructions that are included in the instruction set when those data types are present for the problem. Continued in Table 9.

Data Types	Instructions
boolean	boolean_empty, boolean_swap, boolean_eq, boolean_invert_first_then_and, boolean_flush, boolean_rot, boolean_and, boolean_invert_second_then_and, boolean_xor, boolean_not, boolean_or, boolean_dup, boolean_pop
boolean, char	char_iswhitespace, char_empty, char_isletter, char_eq, char_isdigit
boolean, char, string	string_containschar
boolean, exec	exec_eq, exec_when, exec_if, exec_do*while, exec_while, exec_empty
boolean, float	float_lt, boolean_fromfloat, float_empty, float_lte, float_gte, float_fromboolean, float_gt, float_eq
boolean, float, vector_float	vector_float_contains
boolean, integer	integer_eq, boolean_yank, integer_gte, integer_lt, integer_lte, boolean_shove, integer_empty, integer_gt, integer_fromboolean, boolean_frominteger, boolean_stackdepth, boolean_yankdup
boolean, integer, vector_integer	vector_integer_contains
boolean, string	string_eq, string_emptystring, string_fromboolean, string_contains, string_empty
boolean, string, vector_string	vector_string_contains
boolean, vector_float	vector_float_emptyvector, vector_float_empty, vector_float_eq
boolean, vector_integer	vector_integer_eq, vector_integer_empty, vector_integer_emptyvector
boolean, vector_string	vector_string_empty, vector_string_emptyvector, vector_string_eq
char	char_dup, char_swap, char_flush, char_rot, char_pop
char, exec, string	exec_string_iterate
char, float	char_fromfloat, float_fromchar
char, integer	char_shove, char_stackdepth, integer_fromchar, char_yank, char_yankdup, char_frominteger
char, integer, string	string_occurrencesofchar, string_setchar, string_nth, string_indexofchar
char, string	string_removechar, char_allfromstring, string_replacefirstchar, string_replacechar, string_conjchar, string_fromchar, string_first, string_last
exec	exec_y, exec_pop, exec_rot, exec_s, exec_k, exec_flush, exec_swap, exec_dup, exec_noop, tag, tagged
exec, float, vector_float	exec_do*vector_float
exec, integer	exec_stackdepth, exec_do*times, exec_do*count, exec_do*range, exec_yank, exec_yankdup, exec_shove
exec, integer, vector_integer	exec_do*vector_integer
exec, string, vector_string	exec_do*vector_string
file	file_readline, file_readchar, file_EOF, file_begin
float	float_rot, float_sin, float_cos, float_swap, float_div, float_inc, float_sub, float_flush, float_add, float_tan, float_mult, float_max, float_pop, float_min, float_dup, float_dec, float_mod
float, integer	float_yank, float_frominteger, float_stackdepth, float_shove, float_yankdup, integer_fromfloat
float, integer, vector_float	vector_float_indexof, vector_float_occurrencesof, vector_float_nth, vector_float_set
float, string	float_fromstring, string_fromfloat
float, vector_float	vector_float_conj, vector_float_remove, vector_float_last, vector_float_first, vector_float_replacefirst, vector_float_pushall, vector_float_replace
integer	integer_add, integer_swap, integer_yank, integer_dup, integer_yankdup, integer_flush, integer_shove, integer_mult, integer_stackdepth, integer_div, integer_inc, integer_max, integer_sub, integer_mod, integer_rot, integer_dec, integer_min, integer_pop
integer, string	string_substring, string_take, string_frominteger, string_stackdepth, integer_fromstring, string_yank, string_yankdup, string_length, string_shove
integer, string, vector_string	vector_string_indexof, vector_string_set, vector_string_nth, vector_string_occurrencesof
integer, vector_float	vector_float_shove, vector_float_length, vector_float_stackdepth, vector_float_subvec, vector_float_yank, vector_float_take, vector_float_yankdup
integer, vector_integer	vector_integer_remove, vector_integer_pushall, vector_integer_yank, vector_integer_subvec, vector_integer_last, vector_integer_first, vector_integer_shove, vector_integer_indexof, vector_integer_occurrencesof, vector_integer_replace, vector_integer_replacefirst, vector_integer_take, vector_integer_stackdepth, vector_integer_nth, vector_integer_set, vector_integer_length, vector_integer_yankdup, vector_integer_conj
integer, vector_string	vector_string_stackdepth, vector_string_subvec, vector_string_take, vector_string_shove, vector_string_yank, vector_string_length, vector_string_yankdup

Table 9: **Continuation of Table 8.**

Data Types	Instructions
print	print_newline
print, boolean	print_boolean
print, char	print_char
print, exec	print_exec
print, float	print_float
print, integer	print_integer
print, string	print_string
print, vector_float	print_vector_float
print, vector_integer	print_vector_integer
print, vector_string	print_vector_string
string	string_pop, string_rot, string_rest, string_parse_to_chars, string_reverse, string_swap, string_split, string_flush, string_replacefirst, string_butlast, string_concat, string_replace, string_dup
string, vector_string	vector_string_remove, vector_string_conj, vector_string_first, vector_string_pushall, vector_string_last, vector_string_replacefirst, vector_string_replace
vector_float	vector_float_dup, vector_float_pop, vector_float_rot, vector_float_swap, vector_float_flush, vector_float_reverse, vector_float_rest, vector_float_concat, vector_float_butlast
vector_integer	vector_integer_swap, vector_integer_butlast, vector_integer_flush, vector_integer_rest, vector_integer_concat, vector_integer_rot, vector_integer_reverse, vector_integer_pop, vector_integer_dup
vector_string	vector_string_dup, vector_string_rot, vector_string_rest, vector_string_reverse, vector_string_butlast, vector_string_concat, vector_string_pop, vector_string_flush, vector_string_swap

Table 10: The PushGP parameters that were held constant across the problems. Alternation is a uniform crossover operator similar to ULTRA [20]. Uniform mutation has a constant probability of replacing each instruction with a random one. Uniform close mutation increases or decreases the number of closing parentheses after each instruction probabilistically. Alignment deviation is the standard deviation of index changes during alternation, and for four problems was set to 5 (Number IO, Small Or Large, Median, and Smallest).

Parameter	Value
population size	1000
alternation rate	0.01
alignment deviation	10
uniform mutation rate	0.01
uniform close mutation rate	0.1
Genetic Operator	**Prob**
alternation	0.2
uniform mutation	0.2
uniform close mutation	0.1
alternation followed by uniform mutation	0.5

ment selection designed to encourage diversity preservation in the population [12, 17]. IFS selection greatly rewards individuals for solving training cases that are solved by a small 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 [10]. 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 [6, 19], 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 [6, 4, 3].

Table 12 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. Similarly, the confidence intervals of the difference in success rate between lexicase and tournament or IFS generally show neutral to positive effects of using lexicase.

To examine aggregate performance of each selection method, we calculate the average rank for each method across the 29 problems, with 1 being best and 3 being worst:

Lexicase	IFS	Tournament
1.28	2.26	2.47

Lexicase achieves the lowest average rank, as it has the most or tied for the most successes on every problem except for

Table 12: The first three columns give the number of successful runs out of 100 for each setting, where "Lex" is lexicase selection, "Tourn" is size 7 tournament selection, and "IFS" is implicit fitness sharing with size 7 tournaments. 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 [16], or a pairwise Fisher's exact test with Holm correction if any number of successes is below 5 [15]. The columns "Lex−Tourn" and "Lex−IFS" give the differences in success rate (successful runs divided by total runs) between lexicase and the other two settings. The columns "Lex−Tourn CI" and "Lex−IFS CI" give 95% confidence intervals of the differences in success rate. The "Size" column indicates the smallest size of any simplified solution program.

Problem	Lex	Tourn	IFS	Lex−Tourn	Lex−Tourn CI	Lex−IFS	Lex−IFS CI	Size
Number IO	<u>98</u>	68	72	0.30	$[0.19, 0.41]$	0.26	$[0.16, 0.36]$	5
Small Or Large	5	3	3	0.02	$[-0.04, 0.08]$	0.02	$[-0.04, 0.08]$	27
For Loop Index	1	0	0	0.01	$[-0.02, 0.04]$	0.01	$[-0.02, 0.04]$	21
Compare String Lengths	7	3	6	0.04	$[-0.03, 0.11]$	0.01	$[-0.07, 0.09]$	11
Double Letters	6	0	0	0.06	$[0.00, 0.12]$	0.06	$[0.00, 0.12]$	20
Collatz Numbers	0	0	0	0	–	0	–	
Replace Space with Newline	<u>51</u>	8	16	0.43	$[0.31, 0.55]$	0.35	$[0.22, 0.48]$	9
String Differences	0	0	0	0	–	0	–	
Even Squares	2	0	0	0.02	$[-0.02, 0.06]$	0.02	$[-0.02, 0.06]$	37
Wallis Pi	0	0	0	0	–	0	–	
String Lengths Backwards	<u>66</u>	7	10	0.59	$[0.47, 0.71]$	0.56	$[0.44, 0.68]$	9
Last Index of Zero	<u>21</u>	8	4	0.13	$[0.02, 0.24]$	0.17	$[0.07, 0.27]$	5
Vector Average	16	14	13	0.02	$[-0.09, 0.13]$	0.03	$[-0.08, 0.14]$	7
Count Odds	<u>8</u>	0	0	0.08	$[0.02, 0.14]$	0.08	$[0.02, 0.14]$	7
Mirror Image	<u>78</u>	46	64	0.32	$[0.18, 0.46]$	0.14	$[0.01, 0.27]$	4
Super Anagrams	0	0	0	0	–	0	–	
Sum of Squares	6	2	0	0.04	$[-0.02, 0.10]$	0.06	$[0.00, 0.12]$	7
Vectors Summed	1	0	0	0.01	$[-0.02, 0.04]$	0.01	$[-0.02, 0.04]$	11
X-Word Lines	<u>8</u>	0	0	0.08	$[0.02, 0.14]$	0.08	$[0.02, 0.14]$	15
Pig Latin	0	0	0	0	–	0	–	
Negative To Zero	<u>45</u>	10	8	0.35	$[0.23, 0.47]$	0.37	$[0.25, 0.49]$	8
Scrabble Score	2	0	0	0.02	$[-0.02, 0.06]$	0.02	$[-0.02, 0.06]$	14
Word Stats	0	0	0	0	–	0	–	
Checksum	0	0	0	0	–	0	–	
Digits	7	0	1	0.07	$[0.01, 0.13]$	0.06	$[0.00, 0.12]$	20
Grade	4	0	0	0.04	$[-0.01, 0.09]$	0.04	$[-0.01, 0.09]$	52
Median	45	7	43	0.38	$[0.26, 0.50]$	0.02	$[-0.13, 0.17]$	10
Smallest	81	75	<u>98</u>	0.06	$[-0.06, 0.18]$	−0.17	$[-0.26, -0.08]$	8
Syllables	18	1	7	0.17	$[0.08, 0.26]$	0.11	$[0.01, 0.21]$	14
Problems Solved	22	13	13					

one. The Friedman test on this data gives us a p-value < 0.001, indicating that at least one method performs significantly differently from the others. A post-hoc Wilcoxon-Nemenyi-McDonald-Thompson test [8] indicates that lexicase outranks both IFS and tournament at the 0.05 significance level. These results strongly indicate the utility of lexicase selection for general program synthesis problems.

The data in Table 12 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, Super Anagrams, Digits, and Smallest). 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 fitness 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, how often each technique showed significant improvements over the others, and the average rank of each

Table 11: The PushGP parameters that we varied per problem. "Max Size" gives the maximum number of instructions that can appear in an individual's genome. "Eval Limit" is the number of steps of the Push interpreter that are executed before stopping a program's execution; programs halted in this way may still achieve good results if they print or return results before they are stopped. "Max Gens" gives the maximum number of generations in a single PushGP run. "Prog Eval Budget" is the maximum number of programs that will be evaluated before a run is terminated.

Problem	Max Size	Eval Limit	Max Gens	Prog Eval Budget
Number IO	200	200	200	5,000,000
Small Or Large	200	300	300	30,000,000
For Loop Index	300	600	300	30,000,000
Compare String Lengths	400	600	300	30,000,000
Double Letters	800	1600	300	30,000,000
Collatz Numbers	600	15000	300	60,000,000
Replace Space with Newline	800	1600	300	30,000,000
String Differences	1000	2000	300	60,000,000
Even Squares	400	2000	300	30,000,000
Wallis Pi	600	8000	300	45,000,000
String Lengths Backwards	300	600	300	30,000,000
Last Index of Zero	300	600	300	45,000,000
Vector Average	400	800	300	30,000,000
Count Odds	500	1500	300	60,000,000
Mirror Image	300	600	300	30,000,000
Super Anagrams	800	1600	300	60,000,000
Sum of Squares	400	4000	300	15,000,000
Vectors Summed	500	1500	300	45,000,000
X-Word Lines	800	1600	300	45,000,000
Pig Latin	1000	2000	300	60,000,000
Negative To Zero	500	1500	300	60,000,000
Scrabble Score	1000	2000	300	60,000,000
Word Stats	1000	6000	300	30,000,000
Checksum	800	1500	300	30,000,000
Digits	300	600	300	30,000,000
Grade	400	800	300	60,000,000
Median	200	200	200	20,000,000
Smallest	200	200	200	20,000,000
Syllables	800	1600	300	30,000,000

technique across the problems, 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 12 gives the size (in Push points, which includes instructions and nested parenthesis pairs) 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 [21]. 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 [21]. 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, few if any 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. This technical report expands on the original publication of this benchmark suite [5] by providing details of our implementation of the problems in PushGP. 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. Evolving a digital multiplier with the PushGP genetic programming system. In *GECCO '13 Companion*, pages 1627–1634, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[4] 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.

[5] T. Helmuth and L. Spector. General program synthesis benchmark suite. In *GECCO '15: Proceedings of the 2015 Conference on Genetic and Evolutionary Computation*, July 2015.

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

[7] 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.

[8] M. Hollander and D. Wolfe. *Nonparametric Statistical Methods*. Wiley Series in Probability and Statistics. Wiley, 1999.

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

[10] 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.

[11] 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.

[12] 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.

[13] R. Moll. iJava—an online interactive textbook for elementary Java instruction: Demonstration. *Journal of Computing Sciences in Colleges*, 26(6):55–57, June 2011.

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

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

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

[17] R. Smith, S. Forrest, and A. S. Perelson. Population diversity in an immune system model: Implications for genetic search. In *Foundations of Genetic Algorithms 2*, pages 153–166. Morgan Kaufmann, 1992.

[18] L. Spector. Autoconstructive evolution: Push, pushGP, and pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 137–146, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[19] L. Spector. Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report. In *Proceedings of the Genetic and evolutionary computation conference companion*, GECCO Companion '12, pages 401–408, New York, NY, USA, 2012. ACM.

[20] L. Spector and T. Helmuth. Uniform linear transformation with repair and alternation in genetic programming. In R. Riolo, J. H. Moore, and M. Kotanchek, editors, *Genetic Programming Theory and Practice XI*, Genetic and Evolutionary Computation, chapter 8, pages 137–153. Springer, Ann Arbor, USA, 9-11 May 2013.

[21] 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.

[22] L. Spector, J. Klein, and M. Keijzer. The Push3 execution stack and the evolution of control. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1689–1696, Washington DC, USA, 2005. ACM Press.

[23] 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.

[24] 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.

[25] 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.

[26] 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.