On the Difficulty of Benchmarking Inductive Program Synthesis Methods

Edward Pantridge
MassMutual Financial Group
Amherst, Massachusetts, USA
epantridge@massmutal.com

Thomas Helmuth
Washington and Lee University
Lexington, Virginia
helmutht@wlu.edu

Nicholas Freitag McPhee
University of Minnesota, Morris
Morris, Minnesota 56267
mcphee@morris.umn.edu

Lee Spector
Hampshire College
Amherst, Massachusetts, USA
lspector@hampshire.edu

ABSTRACT

A variety of inductive program synthesis (IPS) techniques have recently been developed, emerging from different areas of computer science. However, these techniques have not been adequately compared on general program synthesis problems. In this paper we compare several methods on problems requiring solution programs to handle various data types, control structures, and numbers of outputs. The problem set also spans levels of abstraction; some would ordinarily be approached using machine code or assembly language, while others would ordinarily be approached using high-level languages. The presented comparisons are focused on the possibility of success; that is, on whether the system can produce a program that passes all tests, for all training and unseen testing inputs. The compared systems are Flash Fill, MagicHaskeller, TerpreT, and two forms of genetic programming. The two genetic programming methods chosen were PushGP and Grammar Guided Genetic Programming. The results suggest that PushGP and, to an extent, TerpreT and Grammar Guided Genetic Programming are more capable of finding solutions than the others, albeit at a higher computational cost. A more salient observation is the difficulty of comparing these methods due to drastically different intended applications, despite the common goal of program synthesis.

CCS CONCEPTS

•**Software and its engineering** → **Genetic programming;**
•**Theory of computation** → *Evolutionary algorithms;*

KEYWORDS

Genetic programming, Machine Learning, Inductive Program Synthesis, Benchmarking

GECCO '17 Companion, Berlin, Germany

978-1-4503-4939-0/17/07…$15.00
DOI: http://dx.doi.org/10.1145/3067695.3082533

ACM Reference format:
Edward Pantridge, Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2017. On the Difficulty of Benchmarking Inductive Program Synthesis Methods. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3067695.3082533

1 INTRODUCTION

Since the creation of Inductive Program Synthesis (IPS) in the 1970s [15], researchers have been striving to create systems capable of generating programs competitively with human intelligence. Modern IPS methods often trace their roots to the fields of machine learning, logic programming, evolutionary computation and others.

Some IPS systems, such as MagicHaskeller, were created to be used as a teaching tool for people learning to program in the Haskell language. Other IPS systems have been used to perform Automated Program Repair [19] to fix bugs in human written code.

As IPS systems become better at generating the same kinds of programs humans would generally write, there will likely be many more applications that have large impacts on the fields of machine learning, artificial intelligence, and software development.

The similarities and differences of IPS methods have been discussed [15], but their performance is rarely compared on problem sets that could provide concrete insight into the capabilities and limitations of each method.

This is partially due to how recently many of current methods have been introduced. TerpreT and the Grammar Guided Genetic Programming system proposed by Forstenlechner et al. have been published for less than a year. Flash Fill and Magic Haskeller were both introduced in the 2010s. PushGP, although first published in 2002, has improved greatly on IPS tasks in recent years [6].

A larger reason for the lack of IPS method comparisons is that these systems often cannot be applied to the same problems, due to various features that are not consistently supported.

Two demonstrative problem sets have been compiled that assess an IPS method's ability to work within a range of levels of abstraction [2], manipulate a variety of data types, utilize complex control structures and produce an arbitrary number of outputs of various forms [8].

This investigation is exclusively considering each method's ability to find solutions. Other measures, such as runtime or hardware models, are not thoroughly examined. In order to determine if a

method can find solutions to a problem, the problem must first be phrased in its entirety to the method. This is not always possible.

It should be noted that we have much more experience using genetic programming than Flash Fill, MagicHakseller, and TerpreT. We made our best effort to become as familiar as possible with all methods used in this comparison. There is still a chance that these systems can be used in ways we are unaware of that could change the results of this experiment.

New tests were run on the PushGP, MagicHaskeller and Flash Fill systems for this comparison. Results gathered for TerpreT and Grammar Guided Genetic Programming (G3P) were reported by the original authors in [2] and [1] respectively. The source code for G3P is available online, and more results could be gathered, but due to how recently the particular G3P technique in [1] was proposed, no results have been gathered beyond what the original authors presented. There is no open source implementation of TerpreT yet.

The conclusions drawn from this comparison will speak to the flexibility of each considered method as well as highlight the issues encountered when using benchmarks to compare IPS methods.

2 PROBLEMS

We take our problems from two sources of benchmark problems. The first set of problems exhibit low-level requirements at various levels of abstraction [2]. The second problem set requires the system to perform tasks similar to those we expect human programmers to perform[8]. These problem sets are both intended to benchmark an IPS system, but ultimately require systems to demonstrate different capabilities. Each is described in detail below.

2.1 Basic Execution Models Problems

The first set of 8 problems, taken from [2], was designed to demonstrate TerpreT's ability to synthesis programs in a variety of execution models that span multiple levels of abstraction. These execution models are: Turing Machine, Boolean Circuits, Basic Block, and Assembly Language. As stated by [2], the problems in this set progress from more abstract execution models towards models which resemble assembly languages. These problems demonstrate how a system performs across a variety of low-level domains.

The problems in this set are described below:

Invert:
Given a binary string (binary tape), invert all the bits.

Prepend Zero:
Insert a 0 in the first index of a binary string and shift all other bits to the right.

Binary Decrement:
Given an input binary string equal to a positive decimal number, return a binary string equal to the input number decremented by one.

2-bit Controlled Shift Register:
Given input bit (r_1, r_2, r_3), return the same bits, except swap the order of r_2 and r_1 if $r_1 == 1$.

Full Adder:
Given a carry bit, c_{in}, and two argument bits, a and b , output a sum bit, s, and carry bit, c_{out}, such that $s + 2c_{out} = c_{in} + a_1 + b_1$.

2-bit Adder:
Given input bits a_1, a_2, b_1, and b_2, output s_1, s_2, and c_{out} such that $s_1 + 2s_2 + 4c_{out} = a_1 + b_1 + 2(a_2 + b_2)$.

Access:
Given an input array, V, and a positive integer, i, return V_i. Assume $0 < i < |A|$.

Decrement:
Given an input array, V, return a new array, U, such that $U_i = V_i - 1$.

In Flash Fill, binary strings were implemented as strings of 1s and 0s in a single cell that was set to the type "Text". In all other systems, the binary strings were implemented using vectors of boolean values.

The TerpreT system was originally given between 5 and 16 input-output examples for each of these problems. PushGP, MagicHaskeller, Flash Fill were also given between 5 and 16 input-output examples for each problem, with certain edge cases manually included while other test cases were generated.

2.2 General Program Synthesis Benchmark Suite

The second set of problems used in this comparison was designed as a benchmark suite for general program synthesis [8]. The 29 problems here require the synthesis of programs with similar characteristics to programs that humans write. Each problem in this suite was taken from an introductory computer science textbook, and therefore emulates the type of programming expected of students who are learning to program. While these benchmarks were first used in genetic programming, they are designed to be usable in any inductive program synthesis system that supports the required data types.

The problems in this set include those that require a range of data types, including strings, integers, floats, characters, and vectors. Many of the problems require outputs to be printed in the style of standard output, and one problem requires multiple output values. Additionally, most of the problems require some level of control flow to solve, whether through iteration, recursion, conditional execution, or list-manipulating higher-order functions.

Where the Basic Execution Models problem set is designed to test a system's ability to perform in low-level domains (binary circuits, assembly language, etc), the General Program Synthesis Benchmark Suite tests the ability to perform higher-level tasks.

Since we do not have space to describe all 29 problems here, we will only describe a few problem that have traits we would like to highlight.

Double Letters:
Given an input string, a solution program must print the string, doubling every letter character and tripling every

exclamation point character. All other non-alphabetic and non-exclamation characters should be printed a single time each.

Replace Space with Newline:
In this problem, the input is a string, and the program must complete two tasks: it must print the input string after replacing each space character with a newline, and it must then functionally return the number of non-whitespace characters in the input string. At minimum, this problem requires the consideration of strings, characters, and printing; a solution will also likely include some type of iteration or string processing.

Even Squares:
Given an integer $0 < n < 10000$, print all of the positive even perfect squares less than n on separate lines.

Count Odds:
Given a vector containing at most 50 integers, return the number of integers in the vector that are odd.

These four problems show the range of requirements presented in this suite, from the use of multiple data types to various control flow techniques. The other 25 problems are thoroughly described in the original suite [8]. Specific implementation details can be found in an accompanying technical report [7].

3 CURRENT STATE OF THE ART

Below are descriptions of the five inductive program synthesis techniques compared in our experiments. The techniques are often some of the most well known, and yet still relatively recent, inductive program synthesis techniques.

3.1 Flash Fill

Flash Fill, recently added to Microsoft Excel, uses version-space algebras to perform program synthesis from examples on string manipulation tasks [3]. It was designed to help non-programmers perform repetitive tasks that would otherwise require them to write Excel macro programs. As such, it is able to quickly create simple programs for one-off repetitive tasks in spreadsheet applications for non-programmers [4, 5, 20, 21].

Building on the use of version-space algebras for programming by demonstration [18], Flash Fill assumes few example inputs and must make simplifying assumptions about the problem space. In particular, the domain-specific language used here is designed for small string manipulation tasks that an end-user may want to perform without knowing how to program them. Adapting the technique for new domains would require a different domain-specific language that is carefully crafted to meet problem requirements while restrictive enough to allow for quick searching. Each different domain-specific language would also require a new synthesis technique; it is unclear whether Flash Fill will even be able to tackle many general program synthesis problems.

To test Flash Fills performance with our problem set, an Excel spreadsheet with one column per input and one column for output was created for each problem. Each spreadsheet included training data, which had both the input columns and output column populated, and unseen testing data, which left the output column cells empty. Flash Fill is deterministic and analytic, thus it was only run once per problem on a single data set.

It is not possible to pose tasks to Flash Fill that required multiple outputs. This includes problems that require printing values in addition to returning an output. Asking Flash Fill to generate each output value in separate cell was considered, but this would be two separate tasks and would not be comparable to the other IPS methods.

3.2 MagicHaskeller

MagicHaskeller [10, 12] synthesizes functional Haskell programs through an exhaustive search of programs with the correct type signatures. It uses a Monte-Carlo algorithm to remove semantically equivalent programs from the search space [11]. More recently, it has also integrated an analytic component based on Igor2 [16], which allows it to synthesize a greater range of programs than can be found in reasonable time using exhaustive search [13]. Additionally, a web interface is available running a time-limited version of MagicHaskeller intended as a Haskell teaching tool for new programmers [14].

These implementations make great use of Haskell's functional instructions, and they perform well on problems that require list manipulations and structural changes. While MagicHaskeller performs quickly on simple problems without too many examples, it has trouble with problems that require a large number of examples to illuminate the relevant edge cases. Additionally, it seems to have trouble with problems that require conditional control flow.

MagicHaskeller uses specifications in the form of a predicate consisting of example function calls and their desired outputs. To then synthesize a function, MagicHaskeller generates a stream of functions that have the same signature as the example function calls. Thus generated functions have the same number of inputs, the same input types, and the same output type.

Generated functions are tested against the input predicate, and a sample of passing functions are presented to the user. The user can then 'Exemplify" the suggested solution functions to see how they would behave given a variety of other inputs. If the user cannot find an adequate solution function, more functions can be generated until the entire stream has been processed.

MagicHaskeller was originally applied to a set of problems that mainly dealt with list manipulation. These types of problems are represeted in the problem set used in this comparison as well, but are generally more complex than what MagicHaskeller was shown to solve in [14].

For our experiments, we used the web hosted version of MagicHaskeller. When using the MagicHaskeller web service, all MagicHaskeller users share the same dynamic programming table used for memoization, and the users cannot select which functions are included when searching. One benefit of using the publicly hosted version of MagicHaskeller is that users can suggest new contributions to the dynamic programming table, which potentially improves the search space for all users upon the next update of the

MagicHaskeller system. Due to MagicHaskeller being hosted on the web, it is difficult to embed in other systems.

Figure 1 shows an example input predicate. Notice that multiple nested predicates can be supplied. MagicHaskeller supports the common primitive data types such as: integers, floats, strings and characters. Vectors and tuples are also supported, which greatly expands the number of problems that were able to be posed to MagicHaskeller.

MagicHaskeller only allows for the synthesis of functions that produce a single output value. In order to pose questions that require multiple outputs, MagicHaskeller was given predicates that specified a tuple as an output. For problems that specify that these values should be printed in addition to the output value, it was considered sufficient if MagicHaskeller could synthesize a program that produced a single string containing all printed values (including white-space and newline) as an element of its output tuple.

3.3 TerpreT

TerpreT is a recently developed, probabilistic programming language that is designed for inductive program synthesis [2]. Problems are specified in the TerpreT language, which is then translated into four different back-end inference algorithms: Forward Marginals Gradient Descent (FMGD), Integer Linear Programming (ILP), Satisfiability Modulo Theories (SMT) and SKETCH.

The TerpreT system attempts to solve IPS problems using these back-end algorithms and returns source code containing the successful parameters found by the successful back-end algorithm, if a solution is present.

The FMGD and ILP back-end algorithms tackle problems through gradient based learning. The SMT and SKETCH back-end algorithms consider the IPS tasks to be constraint satisfaction problems.

Note that there is currently no publicly available implementation of TerpreT and thus only results on benchmark problems provided by the original authors could be obtained. It would be extremely valuable to compare TerpreT's performance on the rest of the problem set used in this paper once an implementation becomes available.

3.4 Genetic Programming

Before the 1990s, evolutionary algorithms were generally only used for optimizing a fixed structure of values. Eventually, this was built upon in such a way that produced executable programs. This technique, named Genetic Programming (GP), is considered inductive program synthesis because it uses input-output examples (refered to as test cases in the field of GP) to evolve a function. In fact, IPS was one of the original motivations for early work in GP [17].

Genetic Programming works by generating an initial population of random programs. This population then follows the evolutionary computation cycle of evaluation, selection, and variation until a solution is found or the run is considered a failure. To evaluate a program, it is given a set of inputs and executed. The set of outputs produced by the program is compared with the desired outputs in order to produce a set of errors.

This set of errors is used in the selection phase to pick programs whose performance merit them to be parents of the next generation of the population. This is done via a variety of selection algorithms, although it has been shown that lexicase selection generally leads to more solutions for IPS tasks [1, 6, 9].

The selected parent programs are then recombined using a series of genetic operators in the variation phase of evolution. This results in a new generation of programs that share traits with their parent programs but are generally never identical.

After a program is produced that outputs the desired values on all input test cases, evolution is stopped and the program is tested on an entirely separate set of inputs to confirm generalization. If the program produces all correct outputs on the second set of inputs, it is considered a solution.

3.4.1 PushGP. Our first comparison genetic programming system, PushGP, evolves programs in a Turing complete, stack based language called Push [22, 23]. Push features separate stacks for each data type, including code. Push programs are lists of instructions and literals. Literals are values that get placed on the stack corresponding to their type. Instructions are built-in functions that pop values off the stacks, modify them, and push them back on the appropriate stacks. Programs are run through an interpreter, which modifies the stacks. After all instructions and literals have been processed through the interpreter, the final state of the stacks is the output of the program.

PushGP was chosen as a comparison method because of its ability to evolve programs that can do the following:

- Manipulate all basic data types, including vectors.
- Return multiple outputs.
- Utilize iteration, recursion, conditional execution, and other more complex control structures.

Given that the chosen problem set contains general software tasks that require all three of the above listed features, most commonly used Genetic Programming systems would be unable to adequately attempt to find a solution.

One of Push's strongest features for program synthesis is the ability of a program to manipulate its own code as it runs [22]. In particular, a running Push program's code exists on a stack just like every other data type, the `exec` stack. Thus, a program can manipulate instructions on the `exec` stack using a variety of generic stack manipulation operations, as well as instructions that implement specific control flow structures such as loops, recursion, if, and when. These instructions allow for a wide range of control strategies that may be useful when presented with a general program synthesis problem.

Implementations of PushGP systems are available in the Clojure programming language[1], as well as a new implementation in Python[2]. The Basic Execution Models problems were run in the Python implementation of PushGP and the General Program Synthesis Benchmark Suite problems were run in the Clojure implementation. In both implementations, the instruction sets and genetic operators were identically implemented for this comparison.

3.4.2 Grammar Guided Genetic Programming (G3P). The second genetic programming system used in this comparison is Grammar Guided Genetic Programming (G3P). It was recently shown by [1]

[1] https://github.com/lspector/Clojush
[2] https://github.com/erp12/pyshgp

```
( ( f [ 1, 2, 3 ] == [ 0, 1, 2 ] ) && ( f [ -1, 1 ] == [ -2, 0 ] ) && ( f [ 7, 5, 4 ] == [ 6, 4, 3 ] ) )
```

Figure 1: An example predicate that can be supplied to MagicHaskeller. This particular predicate produces a solution to the Decrement problem.

that G3P can perform well on general IPS tasks if implemented in a particular way.

The G3P system described in [1] requires the specification of a Grammar and a Skeleton.

The G3P system's grammar is composed of a separate context free grammar for each data type. Supported data types include: booleans, integers, floats, strings, lists of booleans, lists of integers, lists of floats, and lists of strings. There is also an addition grammar for the structure of the program. These grammars are defined in Backus-Naur Form. Together these grammars form a single, general grammar that can be used to attempt arbitrary program synthesis tasks.

Aside from the grammar, the G3P system must be provided a skeleton. This skeleton contains the signature of the function that is going to be evolved. The signature specifies the inputs to the function and the return statement. In order to help avoid runtime errors, functions that implement protected operations (ie. division and logarithm) can also be included in the skeleton.

When an individual in the G3P system is to be evaluated, the code from the individual is placed into the function signature, thus creating a complete function that can be executed. The output of this function on the test cases is compared to the target outputs to receive an error vector.

As part of the G3P grammar definition, it is required to specify the number of variables usable by the system. Forstenlechner et al. claim that 3 variables per data type is sufficient for all problems.

4 LIMITATION OF COMPARISON

The problems described in section 3 were chosen to demonstrate a system's ability to be applied to a wide range of program synthesis tasks. Each system considered in the comparison is designed for different situations which resulted in a difficult comparison. This section describes the various ways in which our comparison could be considered limited. These limitations speak to the flexibility and versatility when being applied to general IPS problems.

4.1 Support For Multiple Outputs

The benchmark problems we consider have different requirements on their outputs, including some problems that print outputs, some that functionally return outputs, and some that require multiple outputs. If a system does not support printing values, it was considered sufficient to return a single string containing all printed text, including newlines, in addition to the other output values.

Flash Fill only supports populating cells in a single column at once. The only way to produce multiple outputs using Flash Fill would be to generate each output value separately. Solving a problem in multiple parts was considered a fundamentally different IPS task, and thus a variety of problems were considered unable to be fully posed to Flash Fill.

MagicHaskeller also only supports synthesizing programs that produce a single output, however MagicHaskeller supports the tuple data type. Given that elements of a tuple do not have to be of the same data type, it was deemed adequate for MagicHaskeller to produce a single tuple containing all outputs for problems that required multiple outputs. For problems that called for printing values, one of the elements of the output tuple would be a string containing all printed text, including newlines.

The G3P grammars do not contain any methods of printing or formatting output values. Thus, any problems that required the use of printing or outputs given in a particular format (i.e. a string with values printed on particular lines) must have been implemented in a way that did not require these behaviors. This implies that the G3P system was not posed the exact same IPS tasks, albeit similar ones.

4.2 Supported Data Types

Many problems in the comparison required the handling of multiple data types, often including vectors.

Excel does not include a native vector or list data structure, and it is not clear what the best way to phrase problems that require vectors to Flash Fill. A string representation of vectors was attempted for some problems. If a problem specifies an input value will be a vector of fixed length, the problem can be posed with each element of the vector in its own cell. If the vector's length is not fixed, this cannot be done, because a tabular structure cannot be formed. If a problem requires a vector output, it cannot be posed to Flash Fill with elements in their own cell because generating each output value would be a separate IPS task. Due to this shortcoming of Flash Fill, there are a number of problems in this comparison that Flash Fill was unable to attempt.

G3P supports list types, but the supported list types contain values of a single data type. In other words, G3P can create and manipulate integer lists that can only contain integer values. There is no list type that can hold values of arbitrary types. This limitation, combined with the previous mentioned lack of printing methods, resulted in no adequate G3P implementation of the "String Differences" problem.

4.3 Number of Training and Test Cases

The number of nested predicates that can be given to MagicHaskeller is limited, and if too many predicates are given the system will produce memory errors. This is a consequence of MagicHaskeller being a web hosted service where resources are shared between all users. Due to this limitation, it was not possible to give MagicHaskeller the same training dataset as the other IPS methods. Although this weakens the presented results, it speaks to the usability and flexibility of the web hosted MagicHaskeller system.

4.4 Access To Systems

There is no publicly available implementation of TerpreT, and thus it is only known how TerpreT performs on the *Basic Execution*

	TerpreT	Flash Fill	MH	PushGP
Invert	✓	*x*	✓	✓
Prepend Zero	✓	✓	✓	✓
Binary Decrement	✓	*x*	*x*	*x*
2BCSR	✓		*x*	✓
Full Adder	✓		*x*	✓
2 Bit Adder	✓		x	*x*
Access	✓	*x*	✓	✓
Decrement	✓	*x*	✓	✓

Figure 2: Results of applying TerpreT, Flash Fill, MagicHaskeller (MH) and PushGP on the Basic Execution Models problems from [2]. A check denotes the system could find a solution. An *x* denotes the problem was fully posed to the system, but a solution was not found. No symbol denotes the problem could not be fully posed to the system. It is not yet known how G3P preforms on this dataset due to how recently the specific G3P method discussed in this paper was presented.

Models problem set. It is unknown how TerpreT would perform on the *General Program Synthesis Benchmark Suite*.

Given how recently the G3P system for general program synthesis was presented, it has not been applied to the *Basic Execution Models* problem set. These results would be informative and likely be fairly easy to obtain.

Flash Fill is not open-source, which makes it impossible to modify for new tasks.

5 RESULTS

Although inductive program synthesis is often thought of as a single field, it is clear from this comparison that IPS methods are generally implemented with a narrow domain in mind. This is made most evident by the limitations discussed in section 4. After making strong efforts to overcome these limitations, the results of applying each IPS method to the problem sets described in section 2 were compiled into Figure 2 and Figure 3.

Flash Fill performed particularly poorly on the the considered problems. Being designed for string manipulation, it is not surprising that it can only solve the "Prepend Zero" problem, where the input binary string is implemented as a text string of 1s and 0s. On all other problems, Flash Fill does not produce a solution. This is not unexpected because no other problem is solely comprised of such simple text manipulation.

MagicHaskeller was only able to solve 10 out of the 37 problems it was applied to. These problems tended to mainly involve applying simple operations to each element of a vector, or simple forms of aggregation.

The system that performed the best on the *Basic Execution Models* problems was TerpreT, while PushGP failed to solve 2 of the 8 problems. Given the success of TerpreT in this domain, it would be interesting to know how the system performs on the *Software Synthesis Benchmark Suite*. The only system for which there is no results on the *Basic Execution Models* problems is G3P. These results would be informative, but given how recently this specific G3P technique was posed these results have not been gathered yet.

	Flash Fill	MH	PushGP	G3P
Number IO	*x*	✓	✓	✓
Small Or Large	*x*	*x*	✓	✓
For Loop Index	*x*	*x*	✓	✓
Compare String Lengths	*x*	*x*	✓	✓
Double Letters	*x*	*x*	✓	*x*
Collatz Numbers	*x*	*x*	*x*	
RSWN		*x*	✓	*x*
String Differences	*x*	*x*	*x*	
Even Squares	*x*	*x*	✓	✓
Wallis Pi	*x*	*x*	*x*	
String Lengths Backwards		✓	✓	✓
Last Index of Zero		*x*	✓	✓
Vector Average		✓	✓	*x*
Count Odds		*x*	✓	✓
Mirror Image		*x*	✓	✓
Super Anagrams	*x*	*x*	*x*	✓
Sum of Squares	*x*	*x*	✓	✓
Vectors Summed		✓	✓	✓
X-Word Lines	*x*	*x*	✓	x
Pig Latin	*x*	*x*	*x*	
Negative To Zero		✓	✓	✓
Scrabble Score	*x*	*x*	✓	*x*
Word Stats	*x*	*x*	*x*	
Checksum	*x*	*x*	✓	
Digits	*x*	*x*	✓	*x*
Grade	*x*	*x*	✓	✓
Median	*x*	*x*	✓	✓
Smallest	*x*	✓	✓	✓
Syllables	*x*	*x*	✓	*x*

Figure 3: Results of Flash Fill, Magic Haskeller (MH), PushGP, and Grammar Guided Genetic Programming (G3P0) on the Software Synthesis Benchmark Suite from [8]. A check denotes the system could find a solution. An *x* denotes the problem was fully posed to the system, but a solution was not found. No symbol denotes the problem could not be fully posed to the system. When the original benchmark paper [8] was published, PushGP had not been able to find solutions to the Checksum problem. Subsequent unpublished work found that expanding the set of test cases led to the discovery of solutions. TerpreT is not seen in this figure, as it has never been applied to these problems to our knowledge, and there is no publicly available implementation.

Both genetic programming systems clearly outperformed Flash Fill and MagicHaskeller on the *Software Synthesis Benchmark Suite*. PushGP solved 23 out of the 29 problems, and G3P solved 16 of the 29 problems.

The IPS methods compared in this paper have different characteristics, such as supported data types. These characteristics are summarized in Figure 5. The differing capabilities impacted the comparison made in this paper, and would likely impact many other comparisons made between other IPS systems via the use of benchmark problems. This highlights an issue in the research area

Problem	Solution
Invert	`f = (map (\b -> abs(b -1 )))`
Prepend Zero	`f = (0:)`
Decrement	`f = (map (subtract 1))`
String Lengths Backwards	`f = (\a -> reverse (map length a))`
Negative To Zero	`f = (map (\b -> max b 0)`
Smallest	`f = (\a b c d -> min d (min c (min b a)))`

Figure 4: Some example solutions produced by MagicHaskeller.

of IPS, in that it is difficult to adequately compare IPS methods that cannot be applied to the same benchmark problems.

It must be mentioned that there is an enormous difference in runtime and computation power required for these IPS systems. Flash Fill and MagicHaskeller produce an output, or report failure, in near instantaneous time. TerpreT, PushGP and G3P require many hours to complete a single run and due to their stochastic nature, often require many runs. It is not entirely uncommon to spend weeks of CPU time on multiple PushGP runs of a single problem and only encounter a small number of solutions, meanwhile MagicHaskeller can produce the same solution every time, nearly instantaneously.

Some of the considered IPS methods were designed with other functionality in mind, aside from solving IPS task. For example, MagicHaskeller, G3P, and TerpreT produce readable source code. PushGP produces code in the Push language, but it cannot be used outside of a Push interpreter and is not easily read by humans. Flash Fill does not provide any access to the solution program.

Figure 4 gives the source code generated by MagicHaskeller to solve six of the problems. For example, the solution for Invert takes each element in the bit string, subtracts 1 from it, and takes the absolute value. The solution for Negative To Zero returns a vector containing either the original number or 0, whichever is greater, thus replacing all negative elements with zero. These examples make it clear that MagicHaskeller tends to produce small solutions when it solves a problem, making them much simpler to understand than solutions given by genetic programming. On the other hand, since it searches the exponential search space of possible programs, it is not able to solve problems that require larger programs to solve.

6 CONCLUSION

The comparison presented in this paper highlights the difficulty of using benchmark problems to compare IPS methods. The IPS methods chosen for this experiment were TerpreT, PushGP, G3P, Flash Fill and MagicHaskeller. The two distinct problem sets used in this comparison require the IPS method to produce programs in low level domains, such as boolean logic, as well as high level domains that require the synthesis of programs similar to what humans write.

The recently unveiled TerpreT system shows effectiveness in low level domains, but there are no results to suggest it can perform well on different kinds of problems. Flash Fill performs quickly and adequately on some simple string manipulation task as designed, but did not show any capabilities of operating outside this narrow domain. MagicHaskeller is capable of performing list comprehensions and certain other simple tasks, but consistently fails to solve problems that require a large number of training instances. PushGP continues to do reasonably well across a broad range of problems, but can be quite expensive when compared to Flash Fill and MagicHaskeller. Grammar Guided Genetic Programming has the added functionality over PushGP of producing usable source code in any language, while still finding solutions to almost as many problems.

The most conclusive finding that has come out of this comparison is that not all IPS systems can be applied to the same problems. This makes comparison extremely difficult. As access to open source implementations of these systems becomes available, it will be easier to extend them to allow for more supported data types, and other features that make comparison easier.

It is clear that certain IPS systems are not suited for some applications, depending in part on whether runtime and computational power are concerns. It does appear that the genetic programming methods, while significantly more computationally expensive, are more flexible in regards to the types of problems to which they can be applied.

7 ACKNOWLEDGEMENT

This material is based upon work supported by the National Science Foundation under Grants No. 1617087, 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.

REFERENCES

[1] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2017. A Grammar Design Pattern for Arbitrary Program Synthesis Problems in Genetic Programming. In *20th European Conference on Genetic Programming*. In press.

[2] Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. 2016. TerpreT: A Probabilistic Programming Language for Program Induction. *CoRR* abs/1608.04428 (2016). http://arxiv.org/abs/1608.04428

[3] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. *SIGPLAN Not.* 46, 1 (Jan. 2011), 317–330. DOI:http://dx.doi.org/10.1145/1925844.1926423

[4] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet Data Manipulation Using Examples. *Commun. ACM* 55, 8 (Aug. 2012), 97–105. DOI:http://dx.doi.org/10.1145/2240236.2240260

[5] William R Harris and Sumit Gulwani. 2011. Spreadsheet table transformations from examples. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 317–328.

[6] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. Lexicase selection for program synthesis: a diversity analysis. In *Genetic Programming Theory and Practice XIII*. Springer.

[7] Thomas Helmuth and Lee Spector. 2015. *Detailed Problem Descriptions for General Program Synthesis Benchmark Suite.* Technical Report UM-CS-2015-006. Computer Science, University of Massachusetts, Amherst. https://web.cs.umass.edu/publication/docs/2015/UM-CS-2015-006.pdf

[8] Thomas Helmuth and Lee Spector. 2015. General Program Synthesis Benchmark Suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO '15)*. ACM, New York, NY, USA, 1039–1046. DOI:http://dx.doi.org/10.1145/2739480.2754769

Method	Language	Runtime	Stochastic	Boolean	Numeric	String	List
TerpreT	TerpreT	Non-trivial	Yes	Yes	Yes	No	Yes
PushGP	Push	Non-trivial	Yes	Yes	Yes	Yes	Yes
G3P	Python	Non-trivial	Yes	Yes	Yes	Yes	Yes
Flash Fill	N/A	Trivial	No	No	No	Yes	No
MagicHaskeller	Haskell	Trivial	No	Yes	Yes	Yes	Yes

Figure 5: Each considered IPS method's capabilities. Language column t. Runtime column denotes if the method requires more than a few seconds to produce a result. Stochastic column indicates if a method is stochastic or deterministic. The remaining columns indicate if the method supports the data type denoted by the column header.

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

[10] Susumu Katayama. 2005. Systematic search for lambda expressions. In *Trends in Functional Programming (Trends in Functional Programming)*, Marko C. J. D. van Eekelen (Ed.), Vol. 6. Intellect, 111–126.

[11] Susumu Katayama. 2008. *Efficient Exhaustive Generation of Functional Programs Using Monte-Carlo Search with Iterative Deepening*. Springer Berlin Heidelberg, Berlin, Heidelberg, 199–210. DOI: http://dx.doi.org/10.1007/978-3-540-89197-0_21

[12] Susumu Katayama. 2010. Recent Improvements of MagicHaskeller. In *Approaches and Applications of Inductive Programming*. Springer. DOI: http://dx.doi.org/10.1007/978-3-642-11931-6_9

[13] Susumu Katayama. 2011. MagicHaskeller: System demonstration. In *Proceedings of AAIP 2011 4th International Workshop on Approaches and Applications of Inductive Programming*.

[14] Susumu Katayama. 2013. MagicHaskeller on the Web: Automated Programming as a Service. In *Haskell '13: Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*. ACM.

[15] Emanuel Kitzelmann. 2009. Inductive Programming A Survey of Program Synthesis Techniques. (2009).

[16] Emanuel Kitzelmann. 2011. Two New Operators for IGOR2 to Increase Synthesis Efficieny. In *Proceedings of AAIP 2011 4th International Workshop on Approaches and Applications of Inductive Programming*.

[17] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA. http://mitpress.mit.edu/books/genetic-programming

[18] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. 2003. Programming by demonstration using version space algebra. *Machine Learning* 53, 1-2 (2003), 111–156.

[19] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-driven synthesis. *ACM SIGPLAN Notices* 49, 6 (6 2014), 408–418. DOI: http://dx.doi.org/10.1145/2594291.2594297

[20] Rishabh Singh and Sumit Gulwani. 2012. Learning Semantic String Transformations from Examples. *Proc. VLDB Endow.* 5, 8 (April 2012), 740–751. DOI: http://dx.doi.org/10.14778/2212351.2212356

[21] Rishabh Singh and Sumit Gulwani. 2012. Synthesizing number transformations from input-output examples. In *Computer Aided Verification*. Springer, 634–651.

[22] Lee Spector, Jon Klein, and Maarten Keijzer. 2005. The Push3 execution stack and the evolution of control. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*. ACM Press, Washington DC, USA, 1689–1696. DOI: http://dx.doi.org/doi:10.1145/1068009.1068292

[23] Lee Spector and Alan Robinson. 2002. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines* 3, 1 (March 2002), 7–40. DOI: http://dx.doi.org/doi:10.1023/A:1014538503543