Genetic Programming Is Best At Software Synthesis

A work in progress title

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 emerged from a variety of research fields in recent decades. These techniques are beginning to be applied in many modern contexts[2][4]. However, these techniques have not been adequately compared on sufficiently demonstrative problems. In this paper, several state-of-the-art methods of IPS are compared across a comprehensive set of problems that search for programs dealing with various levels of abstraction, data types, control structures and number of outputs. Although there are many helpful metrics to consider when comparing these methods, this comparison is mainly focused on success rate.

Keywords

CCS CONCEPTS

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

KEYWORDS

ACM proceedings, LaTeX, text tagging

ACM Reference format:
Edward Pantridge, Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2017. Genetic Programming Is Best At Software Synthesis. In *Proceedings of the Genetic and Evolutionary Computation Conference 2017, Berlin, Germany, July 15–19, 2017 (GECCO '17),* 3 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Since the creation of Inductive Program Synthesis (IPS) in the 1970s[5], 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.

The similarities and differences of these methods have been discussed[5], but their performance is rarely compared on problem sets that could provide concrete insight into the capabilities and limitations of each method.

A demonstrative problem set has been compiled that assess an IPS method's ability to work within a range of levels of abstraction[1], manipulate a variety of data types, produce complex control structures and produce an arbitrary number of outputs of various forms[3].

This investigation is exclusively considering a methods ability to find solutions. Other measures, such as runtime or hardware models, are not discussed. In order to asses 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.

The conclusions drawn from this comparison will speak to the flexibility of each considered method, as well as each method's success rate.

2 CURRENT STATE OF THE ART

2.1 Flash Fill

Flash Fill is a program synthesis technique found in Microsoft Excel [2]. It was designed to help non-programmers perform repetitive tasks that would otherwise require them to write Excel macro programs. It specializes in tasks that require string manipulations.

To test Flash Fills performance with our problem set, an Excel spreadsheet with one column per input and one column per output was created for each problem. Each spreadsheet included training data, which had both the input 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.

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, as well as putting each item from each vector in a separate cell. These approaches occasionally yielded results, but it is not clear what the optimal usage is.

2.2 MagicHaskeller

Write me!

GECCO '17, Berlin, Germany

DOI: 10.1145/nnnnnnn.nnnnnnn

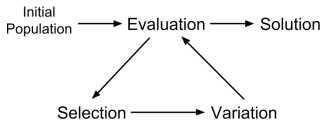

Figure 1: Evolutionary Computation process.

2.3 TerpreT

TerpreT is a recently developed, probabilistic programming language that is designed for inductive program synthesis. 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 one is present.

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 our problem set once an implementation becomes available.

2.4 Genetic Programming

Soon after the rise of evolutionary computations, John R. Koza recognized that evolution could be used for more than optimizing a fixed structure of values. In the 1990s, Koza built upon genetic algorithms 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 origional motivations for creating the field of GP[6].

Genetic Programming works by generating a initial population of random programs. Traditionally these programs are represented as trees where non-leaf nodes each denote a function. The children of each non-leaf node are used as the arguments to their parents. Leaf nodes denote terminal values that could be either a constant or input value. This initial population of random programs then follows the cycle in Figure 1 until a solution is found or the run is considered a failure.

2.4.1 *PushGP.*

Write me!

Why PushGP for IPS?

3 PROBLEMS

3.1 Basic Execution Models Problems

The first set of problems were taken from [1] and were intended 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 [1], the problems in this set progress from more abstract execution models towards models which resemble assembly languages. This makes these problems demonstrative of how a system performs across a variety of low-level domains.

The problems in this set are describes below:

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

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

3.1.3 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.

3.1.4 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 $r_1 == 1$.

3.1.5 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 $s+2c_{out} = c_{in} + a_1 + b_1$.

3.1.6 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 $s_1 + 2s_2 + 4c_{out} = a_1 + b_1 + 2(a_2 + b_2)$.

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

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

3.2 General Program Synthesis Benchmark Suite

Where were problems found

The second set of problems used in this comparison was proposed by [3] in order to provide the field of Genetic Programming with a set of non-trivial benchmark problems.

This problems set included problems that deal with multiple data types, including strings, numbers, boolean and vectors. There are also multiple problems in the set that require multiple output values, or printing certain values to the screen.

Given that the Basic Execution Models problem set is designed to test a systems ability to perform in low-level domains (binary circuits, assembly language, etc), the General Program Synthesis Benchmark Suite is an excellent addition, given that the problems' origins assume a much higher-level implementation (ie. java).

3.2.1 Number IO. Given an integer and a float, print their sum.

3.2.2 Small or Large.

3.2.3 For Loop Index.

3.2.4 Compare String Lengths.

3.2.5 Double Letters.

3.2.6 Collatz Numbers.

3.2.7 Replace Space with Newline.

3.2.8 String Differences.

3.2.9 *Even Squares.*

3.2.10 *Wallis Pi.*

3.2.11 *String Lengths Backwards.*

3.2.12 *Last Index of Zero.*

3.2.13 *Vector Average.*

3.2.14 *Count Odds.*

3.2.15 *Mirror Image.*

3.2.16 *Super Anagrams.*

3.2.17 *Sum of Squares.*

3.2.18 *Vectors Summed.*

3.2.19 *X-Word Lines.*

3.2.20 *Pig Latin.*

3.2.21 *Negative To Zero.*

3.2.22 *Scrabble Score.*

3.2.23 *Word Stats.*

3.2.24 *Checksum.*

3.2.25 *Digits.*

3.2.26 *Grade.*

3.2.27 *Median.*

3.2.28 *Smallest.*

3.2.29 *Syllables.*

4 RESULTS

5 CONCLUSION

6 OUTLINE

- Introduction
  - Motivation for Software synthesis
  - Historical connection to GP
- State of the art
  - Flash Fill
  - Magic Haskeller
  - TerpreT
  - GP
    - Push GP is the right way to do software synthesis
- Comparisons
  - Problems from TerpreT paper
  - Problems from Software Synthesis Benchmark paper
- Results
  - Which problems were able to be posed to each method
  - Which methods were able to find solutions to each problem
- Conclusion

6.1 Things to mention somewhere

- Differences in runtime
- FashFill and MagicHaskeller come up with same think every time. GP might not.
- MagicHaskeller being hosted on web has pros (crowd-sources training) and cons (difficult to integrate in other systesm, large usages could bring it down for everyone)
- TerpreT is not available as an open source project yet.
- Some systems designed with program readability in mind, others not.
- We used boolean vectors as binary tape from TerpreT problems.
- We considered the Access problem and the List-k problem to be synonymous in most contexts.

7 ACKNOWLEDGEMENTS

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] 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

[2] 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

[3] 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

[4] Susumu Katayama. 2013. MagicHaskeller on the Web: Automated Programming as a Service. haskell-symposium. (2013).

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

[6] 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