Transfer Learning of Genetic Programming Instruction Sets

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

Edward Pantridge
Swoop
Cambridge, Massachusetts, USA
ed@swoop.com

Grace Woolson
Hamilton College
Clinton, New York, USA
gwoolson@hamilton.edu

Lee Spector
Amherst College, Hampshire College, and U. Massachusetts, Amherst
Amherst, Massachusetts, USA
lspector@amherst.edu

ABSTRACT

The performance of a genetic programming system depends partially on the composition of the collection of elements out of which programs can be constructed, and by the relative probability of different instructions and constants being chosen for inclusion in randomly generated programs or for introduction by mutation. In this paper we develop a method for the transfer learning of instruction sets across different software synthesis problems. These instruction sets outperform unlearned instruction sets on a range of problems.

CCS CONCEPTS

• **Computing methodologies → Genetic programming**;

KEYWORDS

genetic programming, transfer learning, instruction set, PushGP

ACM Reference Format:
Thomas Helmuth, Edward Pantridge, Grace Woolson, and Lee Spector. 2020. Transfer Learning of Genetic Programming Instruction Sets. In *Genetic and Evolutionary Computation Conference Companion (GECCO '20 Companion), July 8–12, 2020, Cancún, Mexico.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3377929.3389988

1 INTRODUCTION AND PRIOR WORK

Genetic programming (GP) synthesizes problem-solving programs through a process of random generation, random variation, and (partially-random) selection. Both for the random program generation and random variation steps, new program elements are chosen from a collection of elements that is sometimes referred to, informally, as the "primordial ooze." In PushGP [5], which we use in this paper, and other GP program representations, the instructions, literals, and input variables are put together into one *instruction set*, from which random elements are chosen to include in programs.

The conventional wisdom in the field, only occasionally addressed explicitly (such as in Koza's early work [3]), has been that the GP process is relatively robust to changes to the instruction set, and that it will perform reasonably well as long as the instructions needed for solutions are included. Neither the presence of unneeded instructions nor the relative likelihood of choosing different instructions has been thought to make a substantial difference in problem-solving performance, presumably because selection would amplify the prevalence of needed instructions and diminish the prevalence of others.

However, recent work in grammatical evolution has promoted the idea that the composition and relative probability of using specific instructions can have an impact on the performance of evolution [2]. Hemberg at al. modify the probabilities of selecting each production through use of domain knowledge gained from the problem description, and argue more generally for using domain knowledge to influence the instruction set and the GP algorithm as a whole [2].

Our experiments are conducted for software synthesis benchmark problems, for which prior work has used instruction sets that were derived from problem statements [1]. More specifically, in the prior work, all and only the available instructions that manipulate data types mentioned in the descriptions of the problems were used, and each had an equal probability of being chosen whenever an instruction was needed. We call this a *type-tuned* instruction set.

Some initial experiments indicated that instruction sets do matter, in that hand-choosing useful instructions improved performance. This demonstration led us to further investigate the possibility that performance on unsolved problems could be improved through transfer learning of instruction sets, automatically and without human insight.

The expression *transfer learning* is used to describe machine learning approaches in which products of a learning process on one problem are re-used to facilitate learning on a new problem. Transfer learning has been applied in several contexts and in several ways in GP, for example by seeding a population with solutions from related problems; [4] provides a thorough literature review and classification of approaches.

Within this context, the work presented below involves only the transfer of knowledge about instruction sets. Our work here is novel, so far as we know, in applying transfer learning to the construction of instruction sets for GP. In our experiments, the

GECCO '20 Companion, July 8–12, 2020, Cancún, Mexico

ACM ISBN 978-1-4503-7127-8/20/07.
https://doi.org/10.1145/3377929.3389988

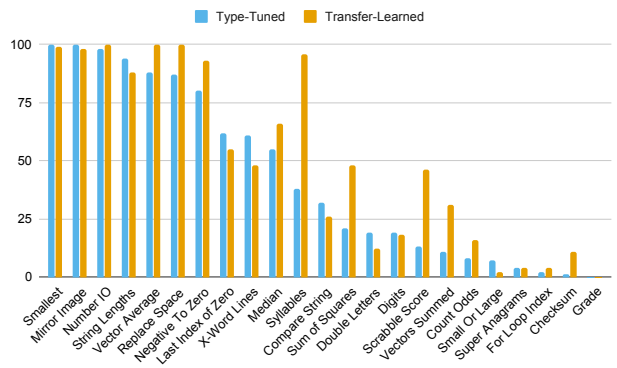

Figure 1: The number of successes out of 100 runs of GP using type-tuned and transfer-learned instruction sets.

transfer-learned instruction sets perform significantly better than the type-tuned instruction sets used in previous studies and are produced in a fully automated way that can be applied to previously unsolved problems.

2 EXPERIMENT

Using a set of 25 program synthesis benchmark problems, we learn instruction sets by using solution programs from the 24 problems not being tested to make up the instruction set when testing the 25^{th} problem. This process ensures that each problem contributes an equal number of instructions to the instruction set.

We present the number of successes out of 100 PushGP runs for type-tuned and transfer-learned instruction sets in Figure 1. PushGP with transfer-learned instruction sets performed significantly better than with type-tuned instruction sets on 8 out of the 25 problems we tested: Vector Average, Replace Space with Newline, Negative To Zero, Syllables, Sum of Squares, Scrabble Score, Vectors Summed, and Checksum. While transfer-learned performed worse on a few problems, it was never significantly worse than type-tuned.

In other experiments, not fully documented here, we show that largely untuned instruction sets, which include "every instruction but the kitchen sink" perform worse than those tuned only on the basis of data types included in the problem description, as has previously been standard when running PushGP on these benchmark problems. Additionally, we find that methods that produce unrealistically well-tuned instruction sets, based on either human intuition or previous solution programs, perform significantly better than type-tuned instruction sets on almost every one of our 25 benchmark problems. These results demonstrate the importance of the composition of the instruction set, encouraging us to seek reasonable methods for automatically tuning the instruction set. The transfer learning approach described here appears to provide such a method, improving problem-solving performance on problems from the same domain as other problems that have already been solved.

3 CONCLUSIONS

In this paper we explore the use of transfer learning of instruction sets on problem-solving performance of PushGP when applied to program synthesis problems. This method adopts instructions from solution programs found in previous runs on other problems to tune the instruction sets on new problems. Our experiments show the benefits of transfer learning of instruction sets, performing significantly better on 8 of the 25 benchmark problems.

It is important to note that while this paper operates mainly in the PushGP system, the ideas and principles derived from the results should transfer to other GP systems. All program synthesis processes must have some means of determining what elements comprise a problem's instruction set, and the experiments discussed below are intended to show the impact of making these sets more or less fine tuned, which impacts all GP systems.

The ideas of transfer learning of instruction sets should be tested in other GP systems (such as grammar-guided GP or grammatical evolution) or even in other problem domains. While the instruction sets in typical GP applications such as symbolic regression tend to be much smaller than those with 100+ instructions in our PushGP experiments, we could imagine that tuning probabilities of using each instruction for random code and mutation could provide benefits in these other domains.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1617087. 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] Thomas Helmuth and Lee Spector. 2015. General Program Synthesis Benchmark Suite. In *GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, Madrid, Spain, 1039–1046. https://doi.org/10.1145/2739480.2754769

[2] Erik Hemberg, Jonathan Kelly, and Una-May O'Reilly. 2019. On domain knowledge and novelty to improve program synthesis performance with grammatical evolution. In *GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, Prague, Czech Republic, 1039–1046. https://doi.org/doi:10.1145/3321707.3321865

[3] 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

[4] Luis Muñoz, Leonardo Trujillo, and Sara Silva. 2019. Transfer learning in constructive induction with Genetic Programming. *Genetic Programming and Evolvable Machines* (Nov. 2019). https://doi.org/10.1007/s10710-019-09368-y

[5] 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. https://doi.org/doi:10.1023/A:1014538503543