Genetic Source Sensitivity and Transfer Learning in Genetic Programming

Thomas Helmuth[1], Edward Pantridge[2], Grace Woolson[1], and Lee Spector[3,4,5]

[1]Hamilton College, Clinton, NY 13323
[2]Swoop, Cambridge, MA 02140
[3]Amherst College, Amherst, MA 01002
[4]Hampshire College, Amherst, MA 01002
[5]University of Massachusetts, Amherst, MA 01003
thelmuth@hamilton.edu

Abstract

Genetic programming uses biologically-inspired processes of variation and selection to synthesize computer programs that solve problems. Here we investigate the sensitivity of genetic programming to changes in the probability that particular instructions and constants will be chosen for inclusion in randomly generated programs or for introduction by mutation. We find, contrary to conventional wisdom within the field, that genetic programming can be highly sensitive to changes in this source of new genetic material. Additionally, we find that genetic sources can be tuned to significantly improve adaptation across sets of related problems. We study the evolution of solutions to software synthesis problems using untuned genetic sources and sources that have been tuned on the basis of problem statements, human intuition, or prevalence in prior solution programs. We find significant differences in performance across these approaches, and use these lessons to develop a method for tuning genetic sources on the basis of evolved solutions to related problems. This "transfer learning" approach tunes genetic sources nearly as well as humans do, but by means of a fully automated process that can be applied to previously unsolved problems.

Introduction and prior work

Genetic programming (GP) synthesizes problem-solving programs through a process of random generation, random variation, and (partially-random) selection (Koza, 1992). 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 tree-based GP, this "ooze" is often divided into the *function set*, from which internal tree nodes are chosen, and the *terminal set*, from which leaf nodes are chosen. For some other forms of GP, however, the distinction between different kinds of program elements may not be as clear or as important. For the purposes of the present paper, we will define the *genetic source* as the full (multi)set of *elements* from which randomly initialized new programs are created and program additions during mutation are selected.[1]

[1]We considered using the more common phrase *gene pool*, but this typically refers to genes that are already present in a population. By contrast, a *genetic source* provides the genes that will be introduced into individuals during mutation or during the construction of entirely new individuals, regardless of their presence or frequency in the existing population.

The conventional wisdom in the field, only occasionally addressed explicitly (such as in Koza's early work (Koza, 1992)), has been that the GP process is relatively robust to changes to the genetic source, and that it will perform reasonably well as long as the elements needed for solutions are included. Neither the presence of unneeded elements nor the relative likelihood of choosing different elements has been thought to make a substantial difference in problem-solving performance, presumably because selection would amplify the prevalence of needed elements 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 elements can have an impact on the performance of evolution (Hemberg et al., 2019). In grammatical evolution, genomes represented as lists of integers are translated into programs by using the integers to select which production to use at each point of a grammar representing the syntax of the language in which the programs are executed (Ryan et al., 1998; O'Neill and Ryan, 2001). Hemberg et al. (2019) 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 genetic source and the GP algorithm as a whole.

On the basis of this prior work, and also of anecdotal observations of differences in performance when using different genetic sources, we decided that it would be worthwhile to systematically explore the effects of providing different genetic sources. More specifically, we decided to empirically evaluate the problem-solving performance implications of several distinct methodologies for specifying the elements in the genetic source and the probabilities for choosing each element.

Our experiments are conducted for software synthesis benchmark problems, for which prior work has used ge-

netic sources that were derived from problem statements. 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* genetic source.

We explore alternative genetic sources to estimate the range of possible performance outcomes that could be expected with refined sources. One alternative that we explore here is to use all available elements, regardless of whether they manipulate types mentioned in the problem statement, each with equal probability. As this method includes "everything but the kitchen sink", we call this the *kitchen sink* source. Another option is to use only elements that we, as human programmers, believe to be important for solving the problem in question, which we call *hand-tuned* sources. Yet another possibility is to boost the probability of choosing elements that have appeared in prior solutions to the problem being solved, which we designate *solution-tuned* sources.

Our initial experiments, described below, demonstrate that genetic sources do matter, in the sense that the different ways of choosing and boosting sources lead to significantly different problem-solving performance. That said, the conditions that *improve* problem-solving performance in these initial experiments, hand-tuning and solution-tuning, cannot be automatically deployed for the solution of new, unsolved problems. Hand-tuning cannot be automated (by definition), and solution-tuning requires the problem to have been solved previously, which is not the case for unsolved problems.

The demonstration that genetic sources do matter, and that some can significantly improve problem-solving performance, led us to further investigate the possibility that performance on unsolved problems could be improved, automatically, through transfer learning of genetic sources. *Transfer learning* in machine learning refers to the reuse of products of a learning process on one problem 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; (Muñoz et al., 2019) provides a thorough literature review and classification of approaches, along with new contributions specifically regarding the cross-problem transfer of constructed features in regression and classification problems.

Within this context, the work presented below involves only the transfer of knowledge about genetic sources. Our work here is novel, so far as we know, in applying transfer learning to the construction of genetic sources for GP. In our experiments, the transfer-learned sources perform significantly better than the type-tuned sources used in previous studies and are produced in a fully automated way that can be applied to previously unsolved problems.[2]

While the work in this paper was conducted mainly with the PushGP system, many of the ideas and principles derived from the results are likely to apply to other GP systems as well. All GP-based program synthesis methodologies involve decisions about which elements are available for inclusion in programs, and with what probabilities. For example, after Forstenlechner et al. (2017) first applied grammar-guided GP to program synthesis problems, they found that using a modified grammar produced significantly better results (Forstenlechner et al., 2018); tuning the grammar further with transfer learning may provide additional improvements.

In the following sections we first describe Push and PushGP, followed by a discussion of our experimental methods and benchmark problems that are used to collect the data presented in this paper. Next, we detail each of our methods for genetic source construction, and discuss the performance of these methods, demonstrating the changes in performance as the sources become more fine-tuned to the problem. Finally, we provide a method for utilizing transfer learning to create genetic sources and discuss the results of implementing this method.

Push and Push Instructions

The experiments in this work are conducted using the PushGP genetic programming system, which evolves programs expressed in the Push programming language (Spector et al., 2005). More specifically, they use Clojush, an implementation of PushGP in Clojure.[3]

Push is a stack-based language in which instructions take their arguments from, and push their results onto, typed data stacks. Stacks are provided for all data types required for target problems, including integers, floating-point numbers, Booleans, strings, vectors of numbers, and so forth. When the arguments an instruction requires are not present on the relevant stacks, the instruction simply acts as a no-op.

Stacks are also provided for executable code, and a special stack called `exec` is used to store and manipulate code that is queued for execution; it is by manipulating items on the `exec` stack that programs can incorporate conditionals, loops, and novel control structures. Additionally, a `code` stack allows Push programs to manipulate Push code, potentially to later transfer it to the `exec` stack for execution.

Instructions that "print" values simply append the item they are printing onto a growing output string. This output string is used as the output of the program for problems that require printed outputs. Other problems in our benchmark suite require the functional returning of values as outputs; these problems take the top item on the relevant stack as the

[2] This paper is an expanded version of the poster paper Helmuth et al. (2020).

[3] `https://github.com/lspector/Clojush`

Table 1: PushGP system parameters.

Parameter	Value
population size	1000
max number of generations	300
parent selection	lexicase
genetic operator	UMAD
UMAD addition rate	0.09

output of the program. Program inputs are accessed via input instructions that push input values onto type-appropriate stacks. We provide one input instruction per input to the program, meaning the number of input instructions can vary per problem.

PushGP's genetic sources include not only instructions for manipulating data and code, but also problem-specific constants of various types and "ephemeral random constants" which, when chosen during program generation or variation, trigger the generation of random constants that are included in the program (Koza, 1992).

Experimental Methods

In all of our experiments we use identical PushGP parameters besides the composition of the genetic source; these parameters are given in Table 1. We use lexicase selection for parent selection, as it has shown to perform well on these problems and others (Helmuth and Spector, 2015; Helmuth et al., 2015). We only use one genetic operator, uniform mutation with addition and deletion (UMAD), as it has given the best known results for PushGP on these problems (Helmuth et al., 2018). We use the standard size-neutral form of UMAD, with an addition rate of 0.09, meaning that a new element from the genetic source will be added before or after each existing item in a parent genome 9% of the time, and a corresponding number of items will be deleted to stay size-neutral on average. We should specifically note that we use a mutation-only system; since all newly-introduced genes from mutation come from the genetic source, the genetic source may play a larger role in this system than one that heavily or exclusively uses crossover for genetic variation.

Program Synthesis Benchmark Problems

To evaluate the performance of PushGP with different genetic sources, we use 25 problems from the "General Program Synthesis Benchmark Suite" (Helmuth and Spector, 2015). These 25 problems include every problem from the benchmark suite (out of 29 total) that has been solved at least once by some run of PushGP. The problems in this suite were originally intended for use in introductory computer science classrooms to assess students' programming ability. Solutions to these problems require the use of various programming constructs, including multiple primitive data types, simple collection types, and assorted control flow structures.

The benchmark problems are implemented as datasets of input-output examples.[4] These datasets contain a relatively small number of manually curated "edge-cases" that resemble a typical suite of unit tests that cover unexpected or extreme input values. In addition, the benchmark datasets contain a large set of randomly generated input-output pairs.

During evolution, programs are evaluated on a training set of all edge-cases and a sample of random cases, as recommended in (Helmuth and Spector, 2015). If any individual program found during evolution passes all training cases it is considered a candidate solution. For a candidate solution to be accepted as a true solution, it must also receive a total error of zero on a held out set of test cases not included in the training set. In this paper, we run an automatic simplification algorithm on each candidate solution that reduces its size without changing its behavior on the training set, which has ben shown to improve generalization (Helmuth et al., 2017). All of the success rates presented in our experiments are based on simplified generalized solutions. We test for significant differences in success rates using a chi-square test with a 0.05 significance level.

Boosted Genetic Sources

Some of our genetic source construction methods (hand-tuned, solution-tuned, and transfer learning) increase the probability of using specific elements. These methods create a *boosted genetic source*. Then, when selecting an element for inclusion in random programs or for mutation, we select a random instruction from the boosted genetic source 80% of the time, and from the original type-tuned genetic source 20% of the time. We chose to not select from the boosted genetic source 100% of the time in case there are useful elements that are not available in the boosted set. This choice may somewhat dilute the effects of the boosted source, but ensures that every instruction related to the problem has some probability of being used for mutation or initialization. Replicating our experiments with 100% boosted instructions remains for future work.

Elements from Solution Programs

For both our solution-tuned genetic sources and our transfer learned genetic sources (described below), we needed to collect, for each problem in our experiments, a set of elements that come from programs that solved the problem in a GP run. Of the sets of runs we could choose from, we selected a set of runs that had produced the most solutions on the most problems, since this would give the most successful programs with which to populate the genetic sources. That set of runs happens to be a set produced using down-sampled lexicase selection (Hernandez et al., 2019; Ferguson et al.,

[4]The datasets for these problems are available at `https://git.io/fjPeh`.

2019; Helmuth and Spector, 2020) and UMAD (Helmuth et al., 2018), although the GP settings of those runs likely do not matter. While we could have used a different set of runs as the source for our solution programs, leading to different transfer-learned genetic sources, we do not see any particular reason to choose one set over another beyond using the runs that produced the most solutions.

For each GP run that found a program that passed all of the tests derived from the training data, we used the same automatic simplification algorithm described above to remove most of the instructions that did not have a functional impact when the program was run on the training data (Helmuth et al., 2017). We then only use those simplified programs that generalize to an unseen test set. For those programs that do, we put all of their elements in a multiset for each problem, so that an element that appears multiple times in a simplified solution program will appear multiple times in the multiset. The more often an element appears in the multiset, the more often it is used by GP.

Effects of Genetic Source on Performance

The first goal of this research is to measure the sensitivity of a GP system's performance to its genetic source. A broad genetic source with many instructions and constants will give the GP system a larger search space to navigate, while a narrow source can either focus the system to search near solutions or, if too extreme, remove necessary instructions for solving the problem.

To measure GP's sensitivity to different genetic sources, we conducted experiments with a large range of possible sources, from uninformed to extremely tuned to the problem. To gather these measurements, we construct uninformed and problem-specific sources based on previous work and human intuition. We acknowledge that these methods are not viable in real-world applications and for unsolved problems, but rather serve to illuminate the potential impact on performance of genetic source optimization. The following subsections describe our four methods of composing genetic sources, giving a range from entirely uninformed (kitchen sink) to better than we could expect a genetic source to be based on any reasonable learning method (hand-tuned and solution-tuned).[5]

Kitchen Sink Genetic Source

We use no knowledge about the problem when creating this genetic source, but instead use a large set of elements that represent or manipulate many data types (i.e. "everything but the kitchen sink"). The genetic source for every problem is almost identical; each one contains 370 of the same instructions, plus a small number of problem-specific input instructions, problem-specific constants, and ephemeral random constants (ERCs). The common instructions include every Push instruction that has been implemented in Clojush for the stacks `exec`, `code`, `print`, `integer`, `float`, `boolean`, `string`, `char`, `vector_boolean`, `vector_integer`, `vector_float`, and `vector_string`. Additionally, it includes seven ERCs for the basic data types (two for `integer` and `float`, and one each for `boolean`, `string`, and `char`).

The motivation for studying performance with the kitchen sink genetic source is that this should allow us to see if the choices made to limit the source for each problem to instructions for specific data types (see next section) actually helps (or hurts) the performance of GP.

Type-tuned Genetic Source

For each problem, the data types relevant to the problem are specified, and then all instructions that make use of those data types are included in the genetic source. Type-tuned genetic sources have been used in all previous studies with Push on the program synthesis benchmark problems used here. More information about which data types are used with which problems and how instructions are associated with data types can be found in the benchmark descriptions (Helmuth and Spector, 2015).

Hand-tuned Genetic Source

For each problem, we picked out the specific elements that seemed useful and necessary to solve the problem, and used them as the boosted genetic source as described above. The process of selecting elements was entirely subjective, based on experience with the problems and a consideration of which Push instructions might be the most useful.

Solution-tuned Genetic Source

When considering how to create a source that was most likely to contain useful instructions and constants, we realized we could take elements from programs that solved each problem in prior GP runs. Since these instructions appeared in solutions to a problem, we expect that they will be useful for solving the problem again. We used the process described above to create a multiset consisting of elements found in simplified solutions to the problem itself. Then, this multiset was used as the boosted genetic source.

Results

Figure 1 presents the number of successful PushGP runs out of 100 for each of our four types of genetic source on each benchmark problem. Note that even though we ran all methods on the String Differences and Even Squares problems, none of them found a single solution, so we omit those problems from our results figures. Recall that type-tuned genetic sources have been used in all prior studies and will be our point of comparison.

[5] Descriptions of the exact genetic sources that we used for each method and each problem can be found at `https://git.io/JfO2I`.

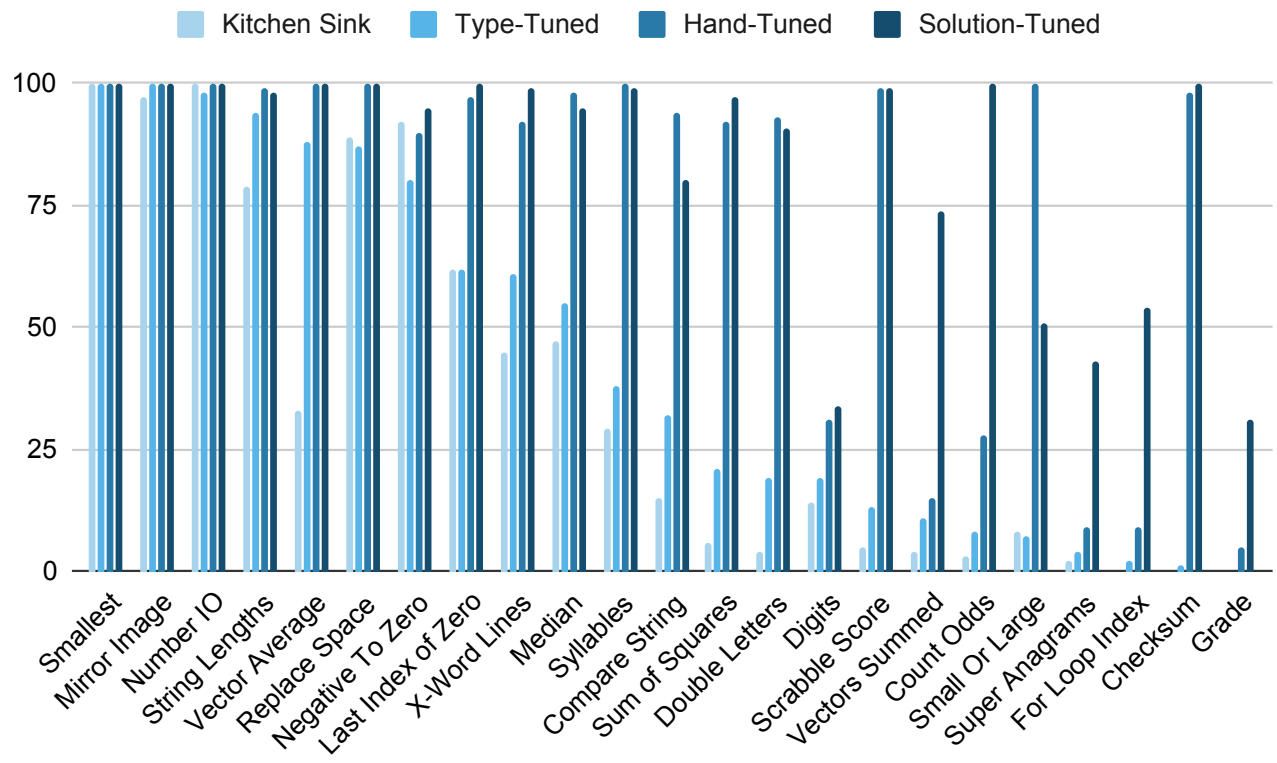

Figure 1: The number of successes out of 100 runs of GP with each type of genetic source on each problem. We order problems based on success rate of Type-Tuned.

GP with kitchen sink sources performed worse than type-tuned on most problems, significantly so on six of them. The one exception is Negative to Zero, on which kitchen sink is significantly better than type-tuned. These results verify the utility of restricting the genetic source to specific data types that seem useful to the problem.

Since both hand-tuned and solution-tuned sources are unrealistically good when attempting to solve previously unsolved problems, we will discuss them together. The results using hand-tuned or solution-tuned are clearly better than type-tuned on most problems, only being similar to type-tuned on the easiest problems. In fact, hand-tuned is significantly better than type-tuned on 13 problems, while solution-tuned is significantly better on 19 problems.

The range of performance between kitchen sink sources on one hand and hand-tuned and solution-tuned on the other shows the importance of the composition of the genetic source. These results indicate that if we can intelligently tune or learn a source for a problem, there exists great potential for gains in performance compared to using an untuned source. While we find it unrealistic to expect sources as well-suited for a problem as the hand-tuned and solution-tuned sources, we expect that realistic learning methods could produce significant improvements even if they only make modest changes to the genetic source. We now turn to one such method of learning a genetic source.

Transfer Learning for Genetic Sources

In order to apply transfer learning to GP genetic sources, we need to learn something useful about the elements used to solve one problem and apply them to another problem. Our general approach is to take elements from solution programs on a set of problems in a problem domain, and use those elements as the genetic source for another, previously unsolved problem in the same domain. Our hypothesis is that the elements from solution programs in one problem domain are more useful than a generic genetic source for the domain. In the domain of program synthesis, we expect the transferred sources to produce better performance than the type-tuned sources that have been the default previously.

To test this hypothesis, we use the multisets of elements scraped from simplified solution programs for each benchmark problem as described above. However, instead of using elements from solutions to a problem to then try to solve the problem itself (as in solution-tuned genetic sources), we use elements from solutions to the other 24 problems in our

benchmark set, leaving out the elements from that problem's solutions. We take these learned elements and use them as the boosted genetic source, as described above. This allows us to empirically test the idea of a transfer-learned genetic source on each of the 25 benchmark problems.

We want each problem to contribute equally to the genetic source. If we simply choose randomly between any of the elements in the scraped solution programs to select a random element, then the chance of selecting elements from different problems will depend on the number of solutions in our sample set of runs, and on the lengths of those solution programs. This would make it much more unlikely to select an element from a rarely solved problem such as String Differences than from an often-solved problem such as Smallest. To make each problem contribute equally, we populate the genetic source with exactly 2000 elements from each problem. As this is more elements than are contained in any of the scraped solutions for any of the problems, we simply included copies of elements from each solution until reaching 2000. Thus across 25 problems there are 50,000 instructions, although 2000 of those will be left out from the problem that is being solved.[6]

One wrinkle in this algorithm is that different problems require different numbers of input instructions. Some only require one input, while others require as many as five inputs. To ensure that each problem has the correct number of input instructions in equal proportions, we replace all input instructions in the transfer-learned genetic source with evenly distributed input instructions that are correct for the problem. For example, the Median problem requires three input instructions, so we replace all of the input instructions in the genetic source (about 9000 in total) with equal numbers of $in1$, $in2$ and $in3$. This ensures that the correct input instructions are present in the equal quantities. The input instructions make up approximately 17% of the instructions in the transfer-learned genetic source.

For 16 of the 25 problems, the benchmark suite recommends including problem-specific constants related directly to the problem description (Helmuth and Spector, 2015). However, the transfer-learned genetic source will not include these constants, since the constants were not used for other problems. Thus we need to add those constants to the genetic source, since they might be necessary to solve the problem. We chose to add a collection of 5000 problem-specific constants, with a uniform distribution across the set of constants specific to the problem. Thus, along with the 48,000 transfer-learned elements, there are a total of 53,000 elements in the genetic source, almost 10% of which are problem-specific constants.[7]

[6]Note that this process is essentially equivalent to choosing a random problem and then choosing a random element from a solution to that problem, but easier to implement.

[7]The exact frequencies of instructions and constants used for every problem can be found at`https://git.io/JfNex`.

Results

We present the number of successes out of 100 PushGP runs for type-tuned and transfer-learned genetic sources in Figure 2. Note that we also ran transfer learned sources on the String Differences and Even Squares problems, and these problems contributed elements to the the sources used for other problems, but since GP found no solutions on the problems we do not include them in Figure 2.

PushGP with transfer-learned genetic sources performed significantly better than with type-tuned sources 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.[8] While transfer-learned performed worse on a few problems, it was never significantly worse than type-tuned.

Is there any relation between the types of elements required to solve the 8 problems on which we showed improvement with transfer-learned genetic sources? At least on first look, no: these problems require a variety of data types and control-flow structures to solve. For example, the output types for these 8 problems include strings, integers, floats, and vectors of integers. So, it does not seem to be the case that the transfer-learned genetic source simply contains more of one type of instruction that benefits all of these problems.

Inputs and Constants

One thing that is clearly different between the transfer-learned genetic source and the previously-used type-tuned genetic source is that the transfer-learned source includes a much larger proportion of input instructions and problem-specific constants. In the type-tuned source, input instructions and problem-specific constants make up about 2-3% of the source each. On the other hand, with the transfer-learned source, input instructions make up about 17% and problem-specific constants make up about 9.5% of the source.

Input instructions (and for some problems specific constants) clearly need to be present in a program for it to solve a problem. Additionally, having more inputs and constants in Push programs means that more of their instructions will actually run instead of being treated as no-ops, as occurs when the stacks they operate on have insufficient elements. Thus increasing their frequency may make it easier for PushGP to find solution programs. We present the hypothesis that the improvements generated by the transfer-learned sources are due to the increased rates of input instructions and problem-specific constants, and not based on learning better proportions of other instructions.

To test this hypothesis we conducted a set of runs on each problem using the type-tuned genetic sources, except that we

[8]We mention all significant differences in text; any other differences are not significant in a chi-squared test at the 0.05 significance level.

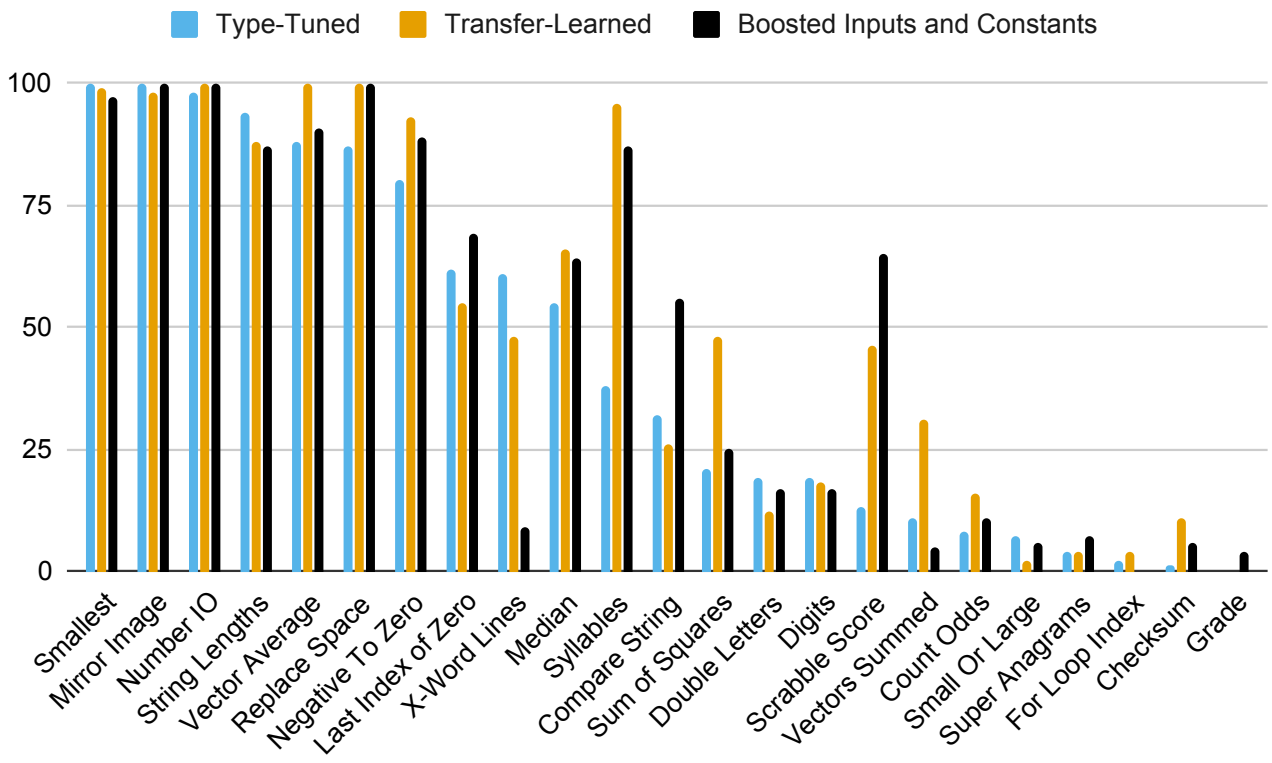

Figure 2: The number of successes out of 100 runs of GP using type-tuned, transfer-learned, and "boosted inputs and constants" genetic sources. We order problems based on success rate of Type-Tuned.

increased the proportions of input instructions and problem-specific constants to the levels found in the transfer-learned sources. We present the number of successes out of 100 runs for these runs (which we refer to as "boosted inputs and constants") in Figure 2. The genetic sources with boosted inputs and constants performed significantly worse than transfer-learned sources on 5 problems (Vector Average, X-Word Lines, Syllables, Sum of Squares, and Vectors Summed), and significantly better on 2 (Compare String Lengths and Scrabble Score).

These results give an edge to transfer-learned sources over type-tuned with boosted inputs and constants, lending evidence against the hypothesis that transfer-learned sources performed better than type-tuned sources due entirely to the increased rates of input instructions and problem-specific constants. This means that some other properties of the transfer-learned source provide benefits as well.

This is not to say that the increased presence of inputs and constants in the transfer-learned sources have no influence. In fact, it appears that one of the important things discovered by transfer-learning of genetic sources here is that increasing inputs and constants helps for many problems. The boosted inputs and constants sources performed significantly better than type-tuned on 4 problems, and significantly worse on 1. Thus some, but not all, of the benefit of transfer-learned sources can be attributed to increased proportions of input instructions and problem-specific constants.

Discussion

It is clear from Figure 1 that different genetic sources have a dramatic impact on evolution's search performance. The lowest overall solution rates were observed with the broadest source, the kitchen sink. Furthermore, the hand-tuned and solution-tuned sources produced the highest overall solution rates. This implies that the choice of a good genetic source is important for solving problems with GP.

The results of our investigation into evolution's sensitivity to genetic sources show that previous benchmarks of PushGP on solving program synthesis problems used a sub-optimal type-tuned source. Performance of PushGP clearly improves as the genetic source is focused. This result agrees with similar findings related to the performance of grammatical evolution (Hemberg et al., 2019).

Our exploration of the use of transfer learning as a means to calibrate genetic sources shows that the method is generally beneficial. Many of the problems for which the transfer-

learned source does not achieve a significantly better result already had solution rates of near 100% with the previously used type-tuned sources.

We acknowledge that more effective means of optimizing genetic sources may exist. The selection of genetic sources is similar in some respects to the hyperparameter tuning of statistical models, which has been the subject of rigorous study. Unfortunately, many of the standard techniques used for hyperparameter tuning (such as cross-validated grid search) are not feasible when searching over large spaces, especially when the evolution process is computationally expensive. The combinatorics of searching for all possible subsets of program elements, along with the long run-time of each run, require us to develop methods for genetic source tuning that either 1) adapt the genetic source during a single evolutionary run or 2) infer an improved genetic source before the evolutionary run using additional origins of information.

Hemberg et al. (2019) proposed that analysis of a problem's textual description could be used to infer a problem specific genetic source. For benchmark problems, the source of domain knowledge can be the text of the problem description. For real-world applications, it is unclear what sources of domain knowledge would be available. Furthermore, state-of-the-art knowledge extraction methods require large datasets of problem descriptions (Sutskever et al., 2014). These challenges motivate the need for genetic source tuning methods that do not require external signals, such as transfer learning or adaptive tuning mid-run.

Conclusions

In this paper we explore the range of effects that the genetic source can have on problem-solving performance of PushGP when applied to program synthesis problems. The results ranged from poor when using a problem-agnostic genetic source to exceptionally better than prior efforts when using unrealistically well-tuned problem-specific genetic sources. These results demonstrate the importance of the composition of the genetic source.

We then show how transfer learning can provide tuned genetic sources to improve GP, performing significantly better on 8 of 25 benchmark problems. We note that the transfer-learned genetic sources are rich in input instructions (which fetch problem inputs) and problem-specific constants, but we also demonstrate, empirically, that the improvements provided by the transfer-learned genetic sources stem from more than just the specific enrichment of input instructions and constants.

The ideas of transfer learning of genetic sources should be tested in other GP systems (such as grammar-guided GP or grammatical evolution) and in other problem domains. While this paper offers one method of tuning genetic sources through transfer learning, other machine learning methods might also be applied to the development and tuning of genetic sources, whether through transfer learning from other problem-solving sessions or through other kinds of learning from other information sources. For example, it may be possible extract relevant information from problem descriptions, or from reliable sources of domain knowledge for specialized domains.

One aspect of our GP system that is different from many others is that it is entirely mutation-based; crossover is not used at all. Since mutation relies on the genetic source for the instructions and constants it introduces, this system makes more use of the genetic source than would systems that primarily or exclusively use crossover to create new children. Such systems would still require a genetic source when creating random programs, but future work would be needed to test the influence of genetic source on these crossover-heavy systems.

While this research concentrated on using GP to solve program synthesis problems, we could also imagine transfer learning of genetic sources being applied to artificial life systems. For example, Hickinbotham and Stepney (2015) limit the availability of opcodes (genetic material) as conservation of matter in the Stringmol artificial chemistry; learning the availability of opcodes from one source to another may lead to other interesting effects. Similar experiments could be tried for tuning instruction sources for mutations and new individuals in artificial life systems such as Avida (Ofria and Wilke, 2004).

Acknowledgements

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

Ferguson, A. J., Hernandez, J. G., Junghans, D., Dolson, E., and Ofria, C. (2019). Characterizing the effects of random subsampling on lexicase selection. In *Genetic Programming Theory and Practice XVII*, East Lansing, MI, USA.

Forstenlechner, S., Fagan, D., Nicolau, M., and O'Neill, M. (2017). A grammar design pattern for arbitrary program synthesis problems in genetic programming. In *EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming*, volume 10196 of *LNCS*, pages 262–277, Amsterdam. Springer Verlag.

Forstenlechner, S., Fagan, D., Nicolau, M., and O'Neill, M. (2018). Extending program synthesis grammars for grammar-guided genetic programming. In Auger, A., Fonseca, C. M., Lourenco, N., Machado, P., Paquete, L., and Whitley, D., editors, *15th International Conference on Parallel Problem Solving from Nature*, volume 11101 of *LNCS*, pages 197–208, Coimbra, Portugal. Springer.

Helmuth, T., McPhee, N. F., Pantridge, E., and Spector, L. (2017). Improving generalization of evolved programs through automatic simplification. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pages 937–944, Berlin, Germany. ACM.

Helmuth, T., McPhee, N. F., and Spector, L. (2018). Program synthesis using uniform mutation by addition and deletion. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, pages 1127–1134, Kyoto, Japan. ACM.

Helmuth, T., Pantridge, E., Woolson, G., and Spector, L. (2020). Transfer learning of genetic programming instruction sets. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '20, Cancun, Mexico. ACM.

Helmuth, T. and Spector, L. (2015). General program synthesis benchmark suite. In *GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1039–1046, Madrid, Spain. ACM.

Helmuth, T. and Spector, L. (2020). Explaining and exploiting the advantages of down-sampled lexicase selection. In *Artificial Life Conference Proceedings*. MIT Press.

Helmuth, T., Spector, L., and Matheson, J. (2015). Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation*, 19(5):630–643.

Hemberg, E., Kelly, J., and O'Reilly, U.-M. (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*, pages 1039–1046, Prague, Czech Republic. ACM.

Hernandez, J. G., Lalejini, A., Dolson, E., and Ofria, C. (2019). Random subsampling improves performance in lexicase selection. In *GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 2028–2031, Prague, Czech Republic. ACM.

Hickinbotham, S. and Stepney, S. (2015). Conservation of matter increases evolutionary activity. In *Artificial Life Conference Proceedings*, pages 98–105. MIT Press.

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

Muñoz, L., Trujillo, L., and Silva, S. (2019). Transfer learning in constructive induction with genetic programming. *Genetic Programming and Evolvable Machines*.

Ofria, C. and Wilke, C. O. (2004). Avida: A software platform for research in computational evolutionary biology. *Artificial Life*, 10(2):191–229.

O'Neill, M. and Ryan, C. (2001). Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358.

Ryan, C., Collins, J. J., and O'Neill, M. (1998). Grammatical evolution: Evolving programs for an arbitrary language. In *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 83–96, Paris. Springer-Verlag.

Spector, L., Klein, J., and Keijzer, M. (2005). The push3 execution stack and the evolution of control. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1689–1696, Washington DC, USA. ACM Press.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.