The approach to uniform variation that we describe in this chapter differs from that of the past work by prioritizing uniformity (of both kinds): we designed our single new genetic operator, which incorporates aspects of both mutation and crossover, in a way that causes uniformity to take precedence over the effects of program shape and size. We did this, essentially, by ignoring the syntactic structure of programs during the first phase of the action of the operator. This "syntax blindness" can produce children that violate syntactic constraints, so we must follow the syntax-blind variation step with a repair step that ensures or restores syntactic validity. While we do not claim that our new operator is "perfectly" uniform in the sense that we are using that term, we do believe that it is more uniform than other operators described in the literature and that its good performance is a consequence of this fact.

In the following sections we first describe the PushGP genetic programming system, within which all of our demonstrations are conducted; Push's minimal syntactic constraints make the repair step of our method particularly simple. We then describe our new operator, which we call ULTRA (for "Uniform Linear Transformation with Repair and Alternation"). We then demonstrate the utility of ULTRA on several problems. Our demonstrations include applications to the ~~difficult~~ drug bioavailability and Pagie-1 benchmark problems, for which ULTRA provides dramatic improvements both in problem-solving power and in control of program size. We also demonstrate the utility of ULTRA on a factorial regression problem that involves greater use of hierarchical program structure, again documenting significant improvements both in problem-solving power and in control of program size. Finally, we include results of an application to a Boolean multiplexer problem, for which the results are mixed. Following these demonstrations we conclude with some comments about directions for future research.

2 Push and PushGP

Push is a programming language that was designed specifically for use in evolutionary computation systems, as the language in which evolving programs are expressed (Spector 2001; Spector and Robinson 2002; Spector et al. 2005). Push is a stack-based programming language that is similar in some ways to others that have been used for genetic programming (e.g. Perkis 1994). It is a postfix language in which literals are pushed onto data stacks and instructions act on stack data and return their results to stacks.

One novel feature of Push is that a separate stack is used for each data type. Instructions take their arguments (if any) from stacks of the appropriate types and they leave their results (if any) on stacks of the appropriate types. This allows instructions and literals to be freely intermixed regardless of type while still ensuring execution safety. By convention, instructions that find insufficient data on the relevant stacks act as "no-ops"—that is, they do nothing.

Many of Push's most unusual and powerful features stem from the fact that code is itself a Push data type, and from the fact that Push programs can easily (and often do) manipulate their own code as they run. Push programs may be hierarchically structured with parentheses, and this hierarchical structure affects how code-manipulation instructions work. It also affects the ways that traditional genetic operators operate on programs, just as the analogous structure of tree-based programs affects the ways that traditional genetic operators operate on them. In the most standard configuration PushGP uses mutation and crossover operators that are almost identical to those used in tree-based genetic programming, with mutation replacing a sub-expression (a literal, an instruction, or a parenthesized code fragment) with a newly generated sub-expression, and with crossover replacing a sub-expression with a sub-expression randomly chosen from another program in the population.

Push and PushGP implementations have been written in C++, Java, JavaScript, Python, Common Lisp, Clojure, Scheme, Erlang, Scala and R. Many of these are available for free download from the Push project page.[2]

3 The ULTRA Operator

"ULTRA," which stands for "Uniform Linear Transformation with Repair and Alternation," is a new genetic operator that takes two parent programs and produces one child program. ULTRA acts on hierarchically structured programs but treats them as linear sequences. It uses each element of the parent sequences with uniform probability and modifies each element of the resulting child sequence with uniform probability. It was motivated by theoretical considerations regarding relations between program size, function, and mutability, and by analogies to the mechanics of mutation and crossover in biological (linear) genomes. We will describe ULTRA here in terms of the elements of Push programs, but the operator could be used on ~~any~~ Other program representations with suitable modifications. ★

ULTRA works by first "linearizing" each parent into a flat, depth-first sequence that includes a token for each literal, instruction, and delimiter (e.g. Push parentheses) in the parent program. It then pads the shorter parent program with null tokens so that both parent programs are the same length. These tokens ensure that instructions in programs of different lengths have approximately equal probabilities of being included in the child, no matter where those instructions occur. The null tokens are removed from the child at the end of ULTRA.

end of the ★

ULTRA next traverses the linearized parents, building the child as a linear sequence of tokens taken from the parents. Traversal begins with a "read head" on the first token of the first parent, and the copying of that token to the child. After this

[2]http://hampshire.edu/lspector/push.html

4 Experiments

To test the performance of ULTRA compared to standard genetic operators, we conducted runs of PushGP on four problems: drug bioavailability, Pagie-1 symbolic regression, factorial symbolic regression, and 6-multiplexer.

The drug bioavailability problem is a predictive modeling problem in which the programs must predict the human oral bioavailability of a set of drug compounds given their molecular structure (Silva and Vanneschi 2009, 2010). This problem has been used for genetic programming benchmarking in various studies (Silva and Vanneschi 2009; Harper 2012), and is recommended as a benchmark problem in a recent article on improving the use of benchmarks in the field (McDermott et al. 2012). Each fitness case for this problem represents a molecule, with 241 floating point inputs, each of which represents a different molecular descriptor of the molecule, and a single floating point output representing the human oral bioavailability of that molecule. The dataset is available online.[3]

The Pagie-1 symbolic regression problem, proposed in Pagie and Hogeweg (1997), is a function on two variables of the form

$$f(x, y) = \frac{1}{(1 + x^{-4})} + \frac{1}{(1 + y^{-4})}.$$

Training set inputs are taken from the range $[-5, 5]$ in steps of 0.4, resulting in 676 fitness cases. This problem has also been used for benchmarking (Harper 2012), and has been recommended as a replacement for "toy" problems such as symbolic regression of the quartic polynomial (McDermott et al. 2012; White et al. 2013).

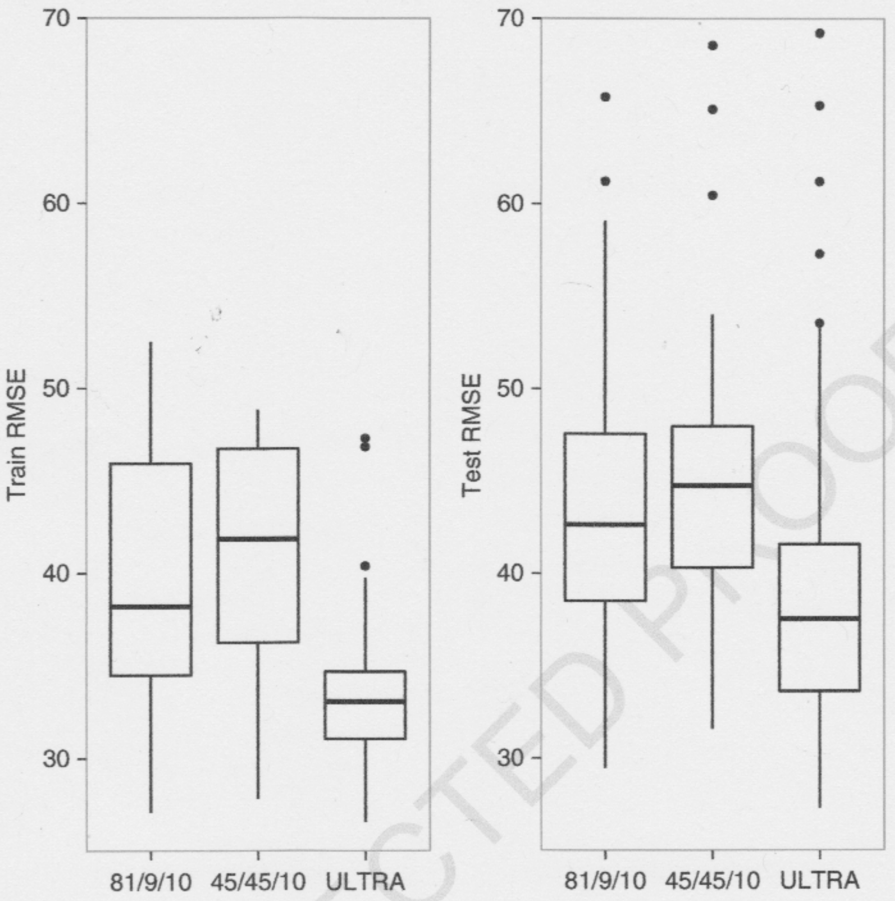
The factorial symbolic regression problem is an integer symbolic regression problem with one input and one output, in which the output should be the factorial of the input. We used 10 test cases, ranging from $1! = 1$ to $10! = 3{,}628{,}800$. Because error magnitudes vary significantly across cases we used "lexicase selection" instead of tournament selection for these runs. Lexicase selection is a parent selection algorithm that was developed to help solve problems that are "modal" in the sense that they require solution programs to perform qualitatively differently actions for inputs that belong to different classes, but it is also useful for problems in which error magnitudes are likely to vary significantly across cases. In lexicase selection a parent is selected by starting with a pool of potential parents—normally the entire population—and then filtering the pool on the basis of performance on individual fitness cases, considered one at a time (Spector 2012).

The 6-multiplexer problem (MUX6) is the standard boolean multiplexer problem used in Koza (1992) and in many subsequent studies by many authors.

In our experiments, we used the PushGP parameters listed in Table 8.1. We made an effort to use parameters similar to those used in previous work on these problems where possible. We used unbiased node selection for all subtree replacement operators. Table 8.2 presents the parameters we used for ULTRA.

[3] http://personal.disco.unimib.it/Vanneschi/bioavailability.txt

ADD FOOTNOTE: Recently, however, concerns have been raised about the use of this problem; see http://jmmcd.net/2013/12/19/gp-needs-better-baselines.html



Use replacement figure in email

Fig. 8.1 Results from the bioavailability problem. We conducted 200 runs for each choice of operators. The RMSE of the best individuals on the training fitness cases (*left*) and on the test fitness cases (*right*). In each plot, subtree replacement 81/9/10 is plotted first, followed by subtree replacement 45/45/10 and then ULTRA. In each box plot, the box stretches from the first quartile to the third quartile with a line for the median in the middle. The whiskers extend to the furthest value within 1.5 times the inter-quartile range. Points beyond the whiskers are outliers, plotted as points. Note that in the *right plot*, 8 outliers in the 81/9/10 set, ~~1~~ outliers in the 45/45/10 set, and ~~1~~ outlier in the ULTRA set fell outside the of the visible plot

5 4

AQ1

Table 8.4 presents the results from our experiments using the factorial problem. ULTRA produced a better success rate and lower computational effort. The difference between the MBF subtree replacement 45/45/10 and ULTRA is statistically significant based on an unpaired t-test at $p = 0.01$

Mean program sizes for the factorial problem runs are presented in Fig. 8.4. The runs using ULTRA maintained a relatively constant mean program size, while runs using subtree replacement 45/45/10 show very fast code growth over the first 100 generations, followed by stable sizes near the maximum program size of 500.

Table 8.5 presents results from our experiments on the 6-multiplexer problem. In contrast to the results on other problems presented here, subtree replacement performs better than ULTRA on all measurements of problem-solving performance.

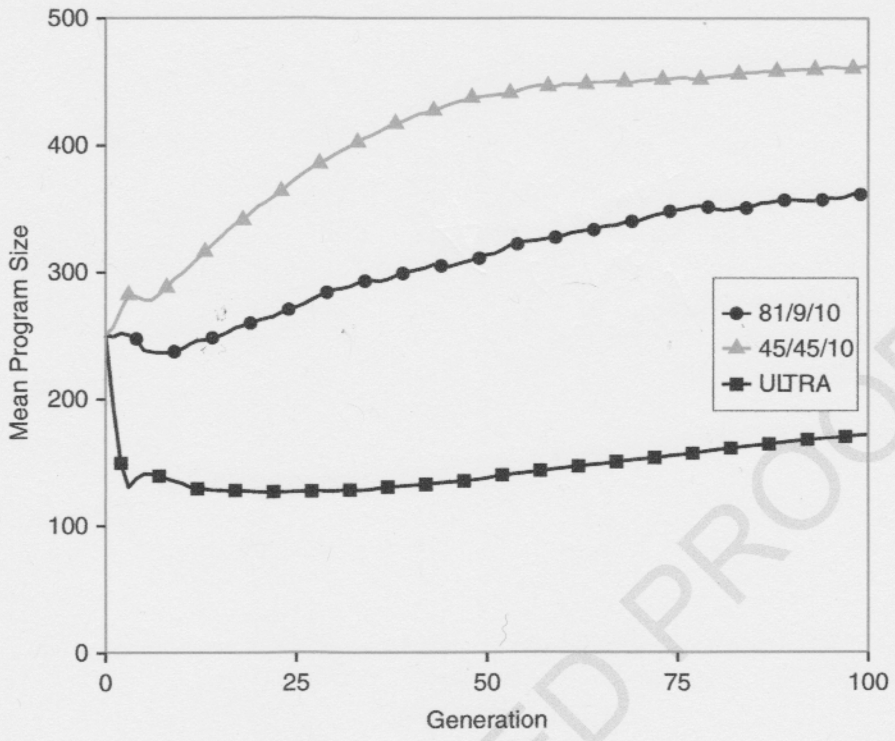


Fig. 8.2 Mean program sizes for the bioavailability problem

Table 8.3 Results on the Pagie-1 problem. We conducted 100 runs for each choice of operators. MBF is the mean best fitness of the run. Note that the reported fitnesses are the mean errors over test cases, not the summed errors

Operators	Successes	MBF
Subtree replacement 80/10/10	0	~~0.363~~ 0.304
Subtree replacement 45/45/10	0	~~0.319~~ 0.333
ULTRA	~~11~~ 3	~~0.031~~ 0.172

Table 8.4 Results on the factorial problem for 100 runs in each condition. CE is computational effort and MBF is the mean best fitness of the run. Note that the reported fitnesses are the mean errors over test cases, not the summed errors

Operators	Successes	CE	MBF
Subtree replacement 45/45/10	2	77,520,000	121,867
ULTRA	61	2,470,000	28,980

The difference between the MBF of subtree replacement 80/10/10 and ULTRA is statistically significant based on an unpaired t-test at $p = 0.01$.

Program sizes for the 6-multiplexer problem are shown in Fig. 8.5. As we have seen before, sizes in subtree replacement runs grow rapidly and stay high, whereas sizes in ULTRA runs decrease rapidly and stay relatively low.

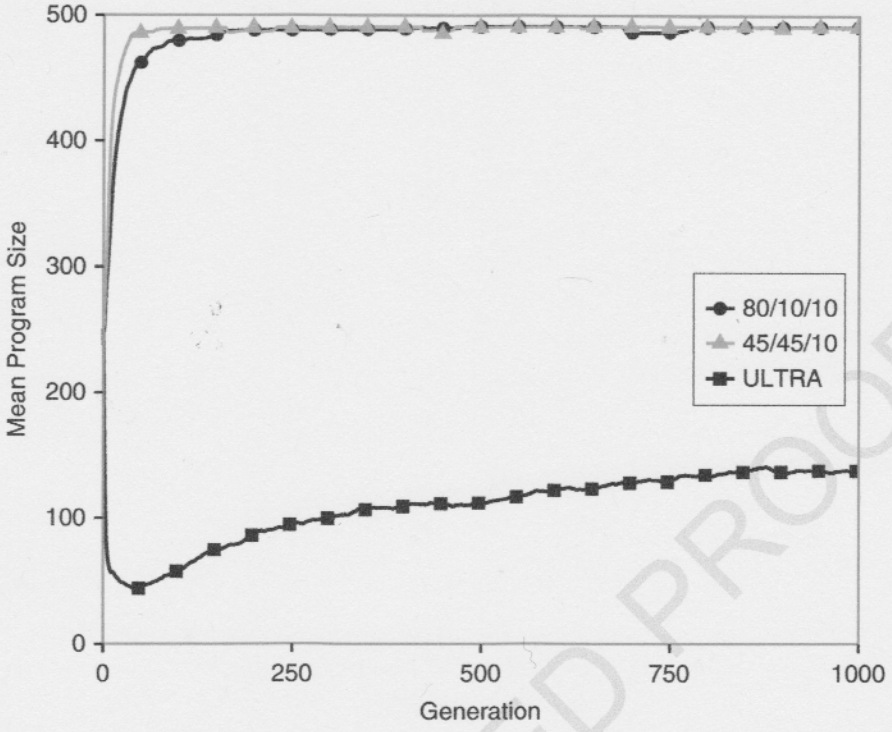


Fig. 8.3 Mean program sizes for the Pagie-1 problem

6 Discussion and Future Work

The results presented here demonstrate that ULTRA, a new genetic operator that prioritizes uniformity and incorporates features of both traditional mutation and traditional crossover, can be an effective tool in helping genetic programming to solve difficult programs and to manage program sizes over the evolutionary process.

The results on the drug bioavailability and Pagie-1 problems~~, which are difficult modeling and symbolic regression problems acknowledged in the field to be useful as benchmarks,~~ demonstrate that ULTRA can produce ~~truly~~ dramatic improvements both with respect to problem-solving power and with respect to managing program sizes. However, it should be noted that these problems do not rely on the hierarchical structure of Push programs when ULTRA is being used since they do not involve code manipulation instructions. A solution to one of these programs would, because of the way that the Push interpreter interprets programs, work just as well with its parentheses moved to different locations or eliminated entirely. Parentheses matter for these problems when traditional subtree-replacement operators are being used because parentheses delineate the units that can be replaced, but when only ULTRA is being used their effects would be limited to providing sites for insertion of new instructions via mutation, and for influencing the effects of deviations during alternation in minor ways.

References

Crawford-Marks R, Spector L (2002) Size control via size fair genetic operators in the PushGP genetic programming system. In: GECCO 2002: proceedings of the genetic and evolutionary computation conference, New York. Morgan Kaufmann, pp 733–739

D'haeseleer P (1994) Context preserving crossover in genetic programming. In: Proceedings of the 1994 IEEE world congress on computational intelligence, Orlando, vol 1. IEEE, pp 256–261

Harper R (2012) Spatial co-evolution: quicker, fitter and less bloated. In: GECCO '12: proceedings of the fourteenth international conference on genetic and evolutionary computation conference, Philadelphia. ACM, pp 759–766

AQ2 Helmuth T, Spector L (2013) Evolving a digital multiplier with the PushGP genetic programming system. In: Workshop on stack-based genetic programming ~~(in preparation)~~ e ☆

Kennedy CJ, Giraud-Carrier C (1999) A depth controlling strategy for strongly typed evolutionary programming. In: Proceedings of the genetic and evolutionary computation conference, Orlando, vol 1. Morgan Kaufmann, pp 879–885

Koza JR (1992) Genetic programming: on the programming of computers by means of natural selection. MIT, Cambridge

Langdon WB (2000) Size fair and homologous tree genetic programming crossovers. Genet Program Evolvable Mach 1(1/2):95–119

Langdon WB, Poli R (2002) Foundations of genetic programming. Springer. http://www.cs.ucl.ac.uk/staff/W.Langdon/FOGP/

Luke S, Panait L (2002) Is the perfect the enemy of the good? In: GECCO 2002: proceedings of the genetic and evolutionary computation conference, New York. Morgan Kaufmann, pp 820–828

Luke S, Panait L (2006) A comparison of bloat control methods for genetic programming. Evol Comput 14(3):309–344

McDermott J, White DR, Luke S, Manzoni L, Castelli M, Vanneschi L, Jaskowski W, Krawiec K, Harper R, De Jong K, O'Reilly UM (2012) Genetic programming needs better benchmarks. In: GECCO '12: proceedings of the fourteenth international conference on genetic and evolutionary computation conference, Philadelphia. ACM, pp 791–798

Moraglio A, Krawiec K, Johnson CG (2012) Geometric semantic genetic programming. In: Parallel problem solving from nature, PPSN XII (Part 1), Taormina. Lecture notes in computer science, vol 7491. Springer, pp 21–31

Niehaus J, Banzhaf W (2003) More on computational effort statistics for genetic programming. In: Genetic programming, proceedings of EuroGP'2003, Essex. Lecture notes in computer science, vol 2610. Springer, pp 164–172

O'Neill M, Ryan C (2001) Grammatical evolution. IEEE Trans Evol Comput 5(4):349–358. doi:10.1109/4235.942529

Page J, Poli R, Langdon WB (1998) Smooth uniform crossover with smooth point mutation in genetic programming: a preliminary study. Technical report CSRP-98-20, School of Computer Science, University of Birmingham. ftp://ftp.cs.bham.ac.uk/pub/tech-reports/1998/CSRP-98-20.ps.gz

Pagie L, Hogeweg P (1997) Evolutionary consequences of coevolving targets. Evol Comput 5(4):401–418

Perkis T (1994) Stack-based genetic programming. In: Proceedings of the 1994 IEEE world congress on computational intelligence, Orlando, vol 1. IEEE, pp 148–153

Poli R, Langdon WB (1998) On the search properties of different crossover operators in genetic programming. In: Genetic programming 1998: proceedings of the third annual conference, University of Wisconsin, Madison. Morgan Kaufmann, pp 293–301

Poli R, Page J (2000) Solving high-order Boolean parity problems with smooth uniform crossover, sub-machine code GP and demes. Genet Program Evolvable Mach 1(1/2):37–56

Poli R, Langdon WB, McPhee NF (2008) A field guide to genetic programming. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, http://www.gp-field-guide.org.uk, (With contributions by J. R. Koza)

, Amsterdam. ACM, pp 1627-1634

Schoenauer M, Sebag M, Jouve F, Lamy B, Maitournam H (1996) Evolutionary identification of macro-mechanical models. In: Angeline PJ, Kinnear KE Jr (eds) Advances in genetic programming 2. MIT, Cambridge, chap 23, pp 467–488

Semenkin E, Semenkina M (2012) Self-configuring genetic programming algorithm with modified uniform crossover. In: Proceedings of the 2012 IEEE congress on evolutionary computation, Brisbane, pp 2501–2506

Silva S, Vanneschi L (2009) Operator equalisation, bloat and overfitting: a study on human oral bioavailability prediction. In: GECCO '09: proceedings of the 11th annual conference on genetic and evolutionary computation, Montreal. ACM, pp 1115–1122

Silva S, Vanneschi L (2010) State-of-the-art genetic programming for predicting human oral bioavailability of drugs. In: Advances in bioinformatics. Springer, pp 165–173

Spector L (2001) Autoconstructive evolution: Push, PushGP, and pushpop. In: Proceedings of the genetic and evolutionary computation conference (GECCO-2001), San Francisco. Morgan Kaufmann, pp 137–146

Spector L (2012) Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report. In: 1st workshop on understanding problems (GECCO-UP), Philadelphia. ACM, pp 401–408

Spector L, Robinson A (2002) Genetic programming and autoconstructive evolution with the Push programming language. Genet Program Evolvable Mach 3(1):7–40

Spector L, Klein J, 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, Washington, vol 2. ACM, pp 1689–1696

Van Belle T, Ackley DH (2002) Uniform subtree mutation. In: Foster JA, Lutton E, Miller J, Ryan C, Tettamanzi AGB (eds) Genetic programming, proceedings of the 5th European conference, EuroGP 2002, Kinsale. Lecture notes in computer science, vol 2278. Springer, pp 152–161

White DR, McDermott J, Castelli M, Manzoni L, Goldman BW, Kronberger G, Jaskowski W, O'Reilly UM, Luke S (2013) Better GP benchmarks: community survey results and proposals. Genet Program Evolvable Mach 14(1):3–29

See email for corrected citation information.

AUTHOR QUERIES

AQ1. "Figure 8.4" has been changed to "Table 8.4". Please check if okay.
AQ2. Please provide proceedings location for "Helmuth and Spector (2013) and Silva and Vanneschi (2010)".

Yes, this is okay: Thank you for the correction.

Corrected in text.

New citation information provided in email.