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 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.

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

²http://hampshire.edu/lspecto/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.

3http://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
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, 7 outliers in the 45/45/10 set, and 1 outlier in the ULTRA set fell outside the of the visible plot.

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 Pajé-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.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Successes</th>
<th>MBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtree replacement 80/10/10</td>
<td>0</td>
<td>0.364</td>
</tr>
<tr>
<td>Subtree replacement 45/45/10</td>
<td>0</td>
<td>0.340</td>
</tr>
<tr>
<td>ULTRA</td>
<td></td>
<td>0.333</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Successes</th>
<th>CE</th>
<th>MBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtree replacement 45/45/10</td>
<td>2</td>
<td>77,520,000</td>
<td>121,867</td>
</tr>
<tr>
<td>ULTRA</td>
<td>61</td>
<td>2,470,000</td>
<td>28,980</td>
</tr>
</tbody>
</table>

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.
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


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.