Uniform Linear Transformation with Repair and Alternation in Genetic Programming

Lee Spector and Thomas Helmuth

Abstract Several genetic programming researchers have argued for the utility of genetic operators that act uniformly. By "act uniformly" we mean two specific things: that the probability of an inherited program component being modified during inheritance is independent of the size and shape of the parent programs beyond the component in question; and that pairs of parents are combined in ways that allow arbitrary combinations of components from each parent to appear in the child. Uniform operators described in previous work have had limited utility, however, because of a mismatch between the relevant notions of uniformity and the hierarchical structure and variable sizes of many genetic programming representations. In this chapter we describe a new genetic operator, ULTRA, which incorporates aspects of both mutation and crossover and acts approximately uniformly across programs of variable sizes and structures. ULTRA treats hierarchical programs as linear sequences and includes a repair step to ensure that syntax constraints are satisfied after variation. We show that on the drug bioavailability and Pagie-1 benchmark problems ULTRA produces significant improvements both in problem-solving power and in program size relative to standard operators. Experiments with factorial regression and with the boolean 6-multiplexer problem demonstrate that ULTRA can manipulate programs that make use of hierarchical structure, but also that it is not always beneficial. The demonstrations evolve programs in the Push programming language, which makes repair particularly simple, but versions of the technique should be applicable in other genetic programming systems as well.

Key words: uniform mutation, uniform crossover, ULTRA, Push, PushGP, drug bioavailability problem, Pagie-1 problem, factorial regression, boolean multiplexer problem

Lee Spector
Cognitive Science, Hampshire College, Amherst, MA USA

Thomas Helmuth
Computer Science, University of Massachusetts, Amherst, MA USA

ERRATUM NOTICE:
Subsequent to the publication of this chapter we discovered that there was a bug in our code for the padding step of our ULTRA algorithm. Flat programs would be padded as described in the text, but programs with significant nesting would be padded incorrectly, usually much less than expected. The ULTRA operator with this padding scheme does perform as reported here, and its source code is available at https://github.com/lspector/Clojush/blob/a1fd37f2a57b10319b16d66bcd2d6707d31c540c/src/clojush/pushgp/genetic_operators.clj

1 Introduction

One of the essential components of any genetic programming system, and of any evolutionary algorithm more generally, is the set of genetic operators used to produce genomes of children from genomes of parents. The operators that are used most commonly in genetic programming stem from Koza's work and involve the replacement of single subprograms either with newly generated random subprograms (for mutation) or with subprograms taken either from other programs in the population (for crossover) (Koza, 1992). Numerous alternatives have since been proposed, on the basis of a wide variety of motivations, including operators designed specifically to control program size (e.g. (Langdon, 2000; Crawford-Marks and Spector, 2002)), operators specialized for specific program element types (e.g. (Schoenauer et al, 1996)), operators that respect or enforce homology between parents (e.g. (D'haeseleer, 1994; Langdon and Poli, 2002)), and operators that combine the semantics of parents in tailored ways (e.g. (Moraglio et al, 2012)). A brief summary of several alternative operators is available in (Poli et al, 2008).

In the present chapter we are primarily concerned with operators that act uniformly across programs. By "act uniformly" we mean two different things, both of which have been designated as "uniform" by others as well. In the first place we mean that the probability of an inherited program component being modified during inheritance is independent of the size and shape of the parent programs beyond the component in question. This property does not hold for standard subtree crossover or mutation because components deeper in trees are more likely to be replaced or modified when those operators are used. Furthermore, two equal components at the same depth of different trees will have different probabilities of being replaced depending on the rest of the tree; the larger the rest of the tree, the less likely the component is to be changed. This property does hold to some extent, however, for versions of "uniform subtree mutation" that have been described in the genetic programming literature (Van Belle and Ackley, 2002). In the second place we are interested in operators that combine pairs of parents in ways that allow arbitrary combinations of components from each parent to appear in the child. Again, this property does not hold for standard subtree crossover, which replaces a single subtree of one parent with a single subtree of another parent. This property does hold to some extent, however, for previously-presented versions of "uniform crossover" (Page et al, 1998). Both properties hold in some traditional genetic algorithms with linear, fixed-length genomes.

As noted in the work cited above, there are several reasons to think that uniformity of both kinds may be helpful, including possible beneficial effects on program size control and search space coverage. For example, if genetic operators are uniform then program growth through the accumulation of non-functional code will not confer "protection from crossover" on a program's functional code, so "program bloat" (and consequent decreases in program search efficacy) due to this cause should be eliminated (Luke and Panait, 2006).[1] However, the operators described in previous research have been limited in their potential applications, largely because of a mismatch between notions of uniform action and the hierarchical structure and variable sizes of genetic programming representations.

For example, the uniform crossover operator developed by Poli, Langdon and Page swaps nodes only among parts of parent trees that are structurally equivalent between the two parents, and the uniform mutation operator that they use can only change nodes to other nodes of equivalent arity (Page et al, 1998; Poli and Langdon, 1998; Poli and Page, 2000). This means that non-matched subtrees of parents can appear in children only in an all-or-nothing fashion, and that mutations cannot change tree shapes. In addition, the hierarchical nature of the crossover operator biases the mixture differently at different tree depths. Furthermore, the amount of mixing permitted would be diminished in the context of strong typing or other enrichments of the program representation. The approach in this work is to vary programs uniformly where it is safe and clear how to do so. Poli et al. have demonstrated, both theoretically and empirically (with even-*n*-parity problems) that this approach can have significant benefits. But even in the context of simple, single-type genetic programming it is not clear how to vary programs uniformly everywhere, and even where it is clear the variations that this method produces are not fully uniform in all of the senses that might be helpful.

More recently Semenkin and Semenkina have extended the methods of Poli et al., specifying random selection of parental components when arities are mismatched. They also allowed the ratio of parent contributions to be adaptively controlled (Semenkin and Semenkina, 2012). While this work increases the amount and variety of mixing that can be produced, the constraints on the application of the technique and the deviations from uniformity are largely unchanged from the prior work.

Other researchers have explored an approach to uniform mutation involving iterative applications of subtree replacement, with the number of iterations depending on the size of the program (Van Belle and Ackley, 2002). This allows mutation to produce arbitrary changes in tree shape and it helps to decouple the chances of a node being modified from the size and shape of the remainder of the program, but it is still subject to significant depth-based biases in modification probabilities. Experiments with this technique have demonstrated improvements in program size control but less clear results with respect to problem-solving power (Van Belle and Ackley, 2002). In other work, different genetic operators have been applied to programs of different depths, leading to somewhat more uniform variation than is produced by standard systems (Kennedy and Giraud-Carrier, 1999).

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

[1] We make no claims here about the prevalence of “protection from crossover.” This is just one example of the effects that operator uniformity can have on program sizes and on genetic programming search efficacy; other effects may interact with other hypothesized causes of program bloat in other ways.

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 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 GP (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 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 modifyies 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 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 end of 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 the first parent, and the copying of that token to the child. After this and each subsequent step there is a fixed probability of alternating between parents; that is, of moving the read head to approximately the same location in the other parent program. The probability of alternating at any given step is specified as the "alternation rate." When alternating between parents, the position of the read head is subjected to Gaussian noise and may change to a higher or lower index; the standard deviation of the noise is given by the "alignment deviation" parameter. Note that alignment deviation may cause some elements of parent programs to be skipped or to be repeated in the child program. After deciding whether to alternate or not, the next token from the current parent is added to the child, and the read head is moved forward. The construction of the child terminates when the read head runs

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

off the end of the current parent or when the child reaches the maximum program size.

After the child sequence has been created through this traversal, it is subjected to a uniform mutation during which each token has uniform probability of being deleted or replaced by a randomly chosen literal, instruction, or delimiter. The probability of any specific token being mutated is given by the "mutation rate" parameter.

In the absence of mutations or alternations, ULTRA would simply traverse the first parent and copy all of its tokens to the child; the child would then be a clone of the first parent. However, alternation and mutation may produce novel programs, some of which may be syntactically invalid; in these cases, the child program must be repaired by ULTRA's repair step.

Fortunately, Push programs are particularly easy to repair. Any sequence of valid instructions, literals, and parentheses is a syntactically valid Push program as long as its parentheses are balanced and as long as its outermost parentheses enclose the entire program. This means that ULTRA can repair a program by simply adding and/or deleting parentheses. Our repair algorithm traverses the child until an imbalance is detected and then fixes the imbalance either by deleting the source of the imbalance or by adding a matching parenthesis in a random location on the appropriate side of the imbalance. The complete repair algorithm requires two passes through the program, one in each direction, and minimizes structural bias arising from repair choices (as might occur, for example, if repairs were always accomplished by adding parentheses to the very beginning or very end of the program). After repair, the child sequence is transformed back into a hierarchical Push program. Finally, all null tokens are removed from the child.

As an example of the overall operation of ULTRA, consider a case involving the two parent programs "(a b (c (d)) e (f g))" and "(1 (2 (3 4) 5) 6)". After linearization the first parent has 15 tokens, while the second has only 12, so the second would be padded with 3 null tokens. Alternation might then produce a sequence of tokens like "(a b 2 (3 4 d)) 6) null null null", which has an extra ")". After repair and removal of null tokens we might have a valid child program such as "(a (b 2 (3 4 d)) 6)".

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)[3]. 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 on the online.[4]

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 $10! = 3628800$. 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.

Table 1 Parameters for experiments.

Problem	Bioavailability	Pagie-1	Factorial	MUX6
Runs Per Condition	200	100	100	100
Population Size	500	1000	1000	500
Max Generations	100	1000	500	200
Max Program Size	500	500	500	200
Max Inital Program Size	500	500	100	200
Max Size for Mutation Code	50	50	20	20
Parent Selection Tournament Size	7	7	Lexicase	7

In our experiments, we used the PushGP parameters listed in Table 1. We made an effort to use parameters similar to those used in previous work on these prob-

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

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

Table 2 ULTRA parameters used in our experiments.

Problem	Bioavailability, Pagie-1, MUX6	Factorial
ULTRA Mutation Rate	0.01	0.05
ULTRA Alternation Rate	0.01	0.05
ULTRA Alignment Deviation	10	10

lems where possible. We used unbiased node selection for all subtree replacement operators. Table 2 presents the parameters we used for ULTRA

For the bioavailability and Pagie-1 problems, we used the float stack instructions $add, sub, mult$, and div as the only non-input instructions. The bioavailability problem uses 241 input instructions, one for each molecular descriptor. As in (Silva and Vanneschi, 2009), we made input instructions and arithmetic instructions equally likely to be chosen by the random code generator. The Pagie-1 problem only requires the input instructions x and y. We also used the constant 1.0, but did not provide an ephemeral random constant.

For the factorial symbolic regression problem we used a more extensive function set that allowed for the manipulation of integers, boolean values, and the execution stack (to permit conditional branches and recursion), but we did not include Push's high-level iteration instructions that allow for trivial solutions. Specifically, we used the constants 0 and 1; an input instruction in; the boolean instructions $and, dup, eq, frominteger, not, or, pop, rot$, and $swap$; the integer instructions add, div, dup, eq, $fromBoolean$, $greaterThan$ (which pushes a boolean), $lessThan$, mod, $mult$, pop, rot, sub, and $swap$; and the exec instructions dup, eq, if, $noop$, pop, rot, $swap$, $when$, and the combinators k, s, and y (Spector et al, 2005).

Our instruction set for the 6-multiplexer problem included the boolean instructions and, or, and not, the exec stack instruction if, and the input instructions $a0$, $a1$, $d0$, $d1$, $d2$, and $d3$.

For some problems we conducted runs in multiple non-ULTRA conditions to show that the relative performance of ULTRA was not due solely to poor choices of parameters for the traditional genetic operators; in these cases we describe the runs with notation such as "81/9/10," meaning that the run used 81% subtree-replacement crossover, 9% subtree-replacement mutation, and 10% straight reproduction.

For all runs described here, fitness is defined as a measure of error, with lower numbers being better. For the bioavailability problem, we use root mean square error (RMSE) as the fitness measure. On this problem, we separate the fitness cases into training and test sets by randomly selecting 70% of the fitness cases for training and 30% of them for testing. For this problem, we determine the statistical significance of whether the RMSE results of two runs come from the same distribution using the Kruskal-Wallis one-way analysis of variance at $p = 0.01$.

For the Pagie-1, factorial, and 6-multiplexer problems we used mean error across fitness cases and no separate test set. We present the number of successes and mean best fitness (MBF: the mean of the best individual fitnesses attained in each run) for these problems. The fitnesses given here are mean errors across test cases, not the sums of those errors. As recommended in (Luke and Panait, 2002; McDermott et al,

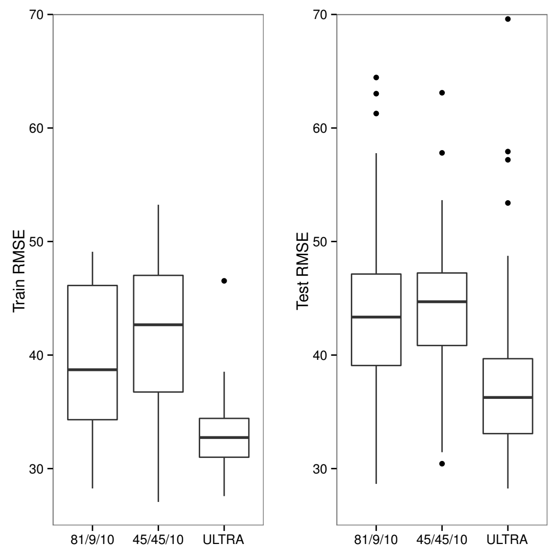

Fig. 1 Results from the bioavailability problem. We conducted 200 runs for each choice of operators. The RMSE of the best individuals on the training fitness cases (left) and on the test fitness cases (right). In each plot, subtree replacement 81/9/10 is plotted first, followed by subtree replacement 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, 5 outliers in the 45/45/10 set, and 4 outliers in the ULTRA set fell outside the of the visible plot.

2012), we use unpaired t-tests to compare differences in MBF for different conditions. For the factorial and 6-multiplexer runs, we also present the computational effort, which gives an estimate of the number of fitness evaluations required to have a good chance of finding a solution (Koza, 1992; Niehaus and Banzhaf, 2003).

5 Results

Figure 1 gives two box plots from our runs of the bioavailability problem, where each genetic operator setting was used in 200 runs. The left plot shows the root mean square error (RMSE) of the best program as measured on the training set. The right plot shows the RMSE of the same individuals on the test set. Both subtree replacement 81/9/10 and subtree replacement 45/45/10 differ statistically significantly from ULTRA on both the training and test sets. ULTRA appears to be able to find more accurate models of the training data than subtree replacement without overfitting the training data.

The mean program sizes with respect to generation are plotted in Figure 2. The runs using subtree replacement show steady growth in program sizes, whereas those using ULTRA quickly fall at the beginning of the run and then remain relatively

Fig. 2 Mean program sizes for the bioavailability problem.

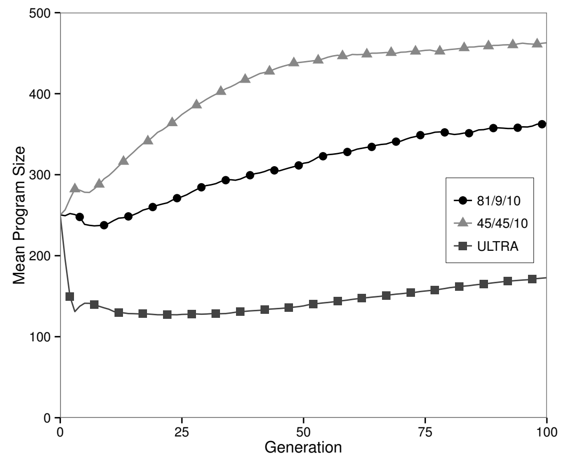

Table 3 Results on the Pagie-1 problem. We conducted 100 runs for each choice of operators. MBF is the mean best fitness of the run. Note that the reported fitnesses are the mean errors over test cases, not the summed errors.

Operators	Successes	MBF
Subtree Replacement 80/10/10	0	0.304
Subtree Replacement 45/45/10	0	0.333
ULTRA	3	0.172

Table 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

steady. The lower program sizes of ULTRA runs may contribute to the relative lack of overfitting (that is, better generalization) of the programs produced in these runs.

Table 3 presents the results of our experiments on the Pagie-1 problem. PushGP using ULTRA found perfect solutions in 15 out of 100 runs, whereas runs with subtree replacement found none with either parameter setting. The differences in MBF between subtree replacement 80/10/10 and ULTRA, and between subtree replacement 45/45/10 and ULTRA, are statistically significant at the $p = 0.01$ level according to an unpaired t-test.

The mean program sizes in our Pagie-1 experiments are shown in Figure 3. Runs using subtree replacement experienced quick code growth, reaching mean sizes near the maximum program size of 500 within the first 50 generations. After this point, it is difficult for the genetic operators to make changes to large programs without exceeding the program size limit. On the other hand, the mean program sizes of ULTRA runs quickly drop to around size 50, and then climb to approach 100.

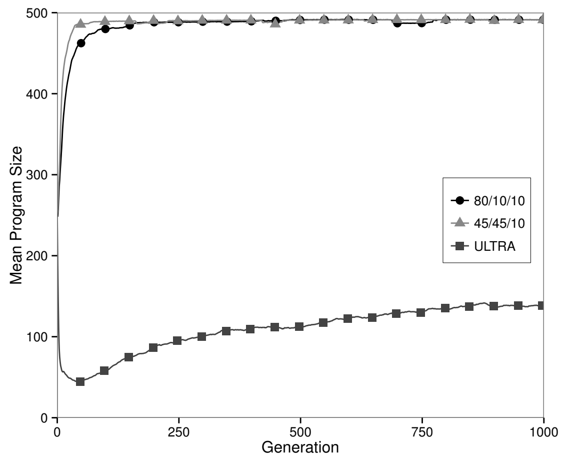

Fig. 3 Mean program sizes for the Pagie-1 problem.

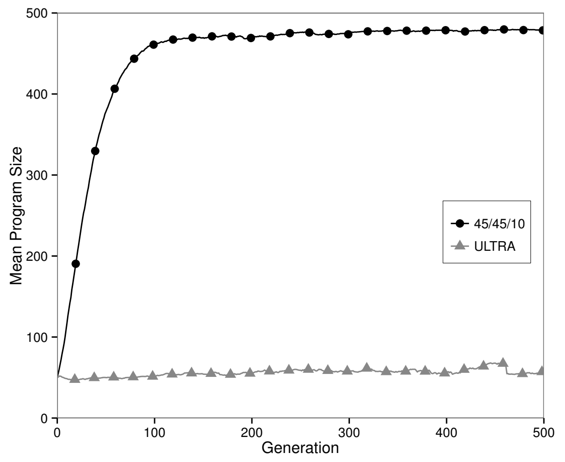

Fig. 4 Mean program sizes for the factorial problem.

Table 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 Figure 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 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.

Table 5 Results on the 6-multiplexer problem, with 100 runs in each condition. CE is computational effort and MBF is the mean best fitness of the run.

Operators	Successes	CE	MBF
Subtree Replacement 80/10/10	85	135,000	0.009
ULTRA	58	369,000	0.038

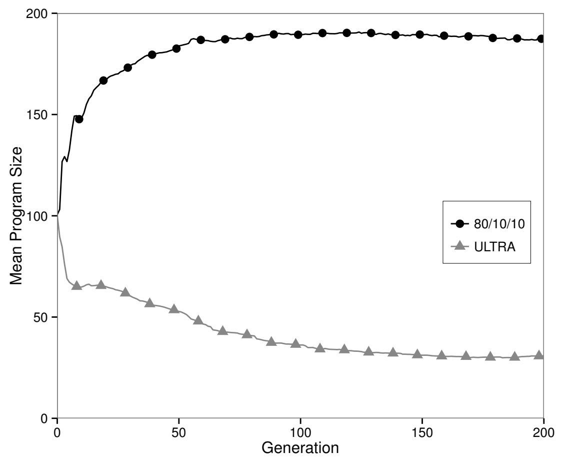

Fig. 5 Mean program sizes for the 6-multiplexer problem.

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 Figure 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 demonstrate that ULTRA can produce 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.

For this reason we sought to demonstrate the use of ULTRA on a problem for which parentheses play a semantic role in the execution of programs, through the use of code manipulation instructions. The factorial symbolic regression problem that we demonstrated includes several instructions that are affected by the placement of parentheses. For example, the $exec\_if$ instruction will execute one of the two expressions that follows it and skip the other, depending on the value on top of the boolean stack. Since parentheses delineate the boundaries of these expressions, their placement is crucial. Several other instructions used in this problem, including the Y combinator instruction, rely on the placement of parentheses in a similar way. The fact that ULTRA performed so well on this problem indicates that it is capable of evolving hierarchically structured program representations even though it treats these programs as linear structures and uses a repair process to ensure that syntactic constraints (parentheses matching) is maintained.

In other work, not presented here, we have shown that ULTRA is also useful for the evolution of multiple-output digital multipliers (Helmuth and Spector, 2013) [5].

The results on the 6-multiplexer problem indicate that ULTRA is not a panacea, at least not with the parameters that were used here. It may be the case that if we explored the space of parameters for ULTRA as well as subtree replacement operators that we would find settings that allow ULTRA to perform better than subtree replacement. On the other hand, these results may indicate that there aren't as many solutions to the 6-multiplexer problem at the smaller mean program sizes that ULTRA tends to produce.

The 6-multiplexer results lead to one obvious avenue for future research: How should ULTRA's parameters be set? We set them more or less arbitrarily for the runs presented here; conceivably we could develop guidelines for their values, based on characteristics of a problem, or mechanisms by which the parameters could be set adaptively over the course of a run.

Another important avenue for future research concerns more rigorous analysis of when ULTRA has better performance than traditional subtree-replacement operators, when ULTRA produces smaller programs than traditional subtree-replacement operators, and when and how these two things are related to each other. Furthermore, the extent to which ULTRA is truly uniform should be studied more systematically. An initial examination of ULTRA on flat programs shows that instructions within the parent programs have approximately equal probability of being included in the child program, and that the mean program size of a child produced by ULTRA with differently sized parents is approximately the mean of the parent program sizes. But it is possible, for example, that program repair and/or alignment deviations near the beginnings or ends of parent programs will produce non-uniform effects. It would be worthwhile to investigate these issues further.

[5] Note that this work gives an outdated description of ULTRA that does not pad the shorter program before alternation.

Finally, while the application of ULTRA to the evolution of Push programs is particularly simple—because Push program repair requires only the balancing of parentheses—we would like to look at the application of ULTRA in several different contexts. On the one hand it would be interesting to look at the application of ULTRA to programs in Push-like languages that do not even require parentheses, but rather use high-level instructions and/or markers within a linear program to delineate program structure. This would make it unnecessary to repair programs at all, and might provide both enhanced power and enhanced elegance.

On the other hand it would be interesting to look at the application of ULTRA to traditional tree-based genetic programming and to other genetic programming representations. For any such representation a repair mechanism will have to be developed that can re-establish syntactic constraints that may be violated by mutation and alternation. For example, in tree-based genetic programming, using traditional representations, it would be necessary not only to balance parentheses but also to ensure that only functions appear in function position (first after a "("), that only sub-expressions and terminals appear in non-function positions, and that the arities of functions are respected. Alternative representations, for example representations that omit parentheses and require structure to be inferred from the positions of functions and a table of function arities, might allow for simpler repair mechanisms. In any event, while repair in some representations will be more difficult than it is in Push, repair should nonetheless be possible and once a repair mechanism has been implemented one could use ULTRA with any program representation. Indeed, for some other representations, e.g. grammatical evolution with linear genomes (O'Neill and Ryan, 2001), the implementation of ULTRA should be particularly straightforward.

Acknowledgements Thanks to Micah Savitzky for exploratory work that later led to the ideas in this paper, to Emma Tosch and Kyle Harrington for discussions and code, to Josiah Erikson for systems support, and to Hampshire College for support for the Hampshire College Institute for Computational Intelligence. This material is based upon work supported by the National Science Foundation under Grant No. 1017817. 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

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, Morgan Kaufmann Publishers, New York, pp 733–739

D'haeseleer P (1994) Context preserving crossover in genetic programming. In: Proceedings of the 1994 IEEE World Congress on Computational Intelligence, IEEE Press, Orlando, Florida, USA, vol 1, 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, ACM, Philadelphia, Pennsylvania, USA, pp 759–766

Helmuth T, Spector L (2013) Evolving a digital multiplier with the pushgp genetic programming system. In: GECCO '13 Companion: Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion, ACM, Amsterdam, The Netherlands, pp 1627–1634

Kennedy CJ, Giraud-Carrier C (1999) A depth controlling strategy for strongly typed evolutionary programming. In: Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann, Orlando, Florida, USA, vol 1, pp 879–885

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

Langdon WB (2000) Size fair and homologous tree genetic programming crossovers. Genetic Programming and Evolvable Machines 1(1/2):95–119

Langdon WB, Poli R (2002) Foundations of Genetic Programming. Springer-Verlag, URL 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, Morgan Kaufmann Publishers, New York, pp 820–828

Luke S, Panait L (2006) A comparison of bloat control methods for genetic programming. Evolutionary Computation 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, ACM, Philadelphia, Pennsylvania, USA, pp 791–798

Moraglio A, Krawiec K, Johnson CG (2012) Geometric semantic genetic programming. In: Parallel Problem Solving from Nature, PPSN XII (part 1), Springer, Taormina, Italy, Lecture Notes in Computer Science, vol 7491, pp 21–31

Niehaus J, Banzhaf W (2003) More on computational effort statistics for genetic programming. In: Genetic Programming, Proceedings of EuroGP'2003, Springer-Verlag, Essex, LNCS, vol 2610, pp 164–172

O'Neill M, Ryan C (2001) Grammatical evolution. IEEE Transactions on Evolutionary Computation 5(4):349–358, DOI 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. Tech. Rep. CSRP-98-20, University of Birmingham, School of Computer Science, URL 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. Evolutionary Computation 5(4):401–418

Perkis T (1994) Stack-based genetic programming. In: Proceedings of the 1994 IEEE World Congress on Computational Intelligence, IEEE Press, Orlando, Florida, USA, vol 1, 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, Morgan Kaufmann, University of Wisconsin, Madison, Wisconsin, USA, pp 293–301

Poli R, Page J (2000) Solving high-order Boolean parity problems with smooth uniform crossover, sub-machine code GP and demes. Genetic Programming and Evolvable Machines 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`, URL http://www.gp-field-guide.org.uk, (With contributions by J. R. Koza)

Schoenauer M, Sebag M, Jouve F, Lamy B, Maitournam H (1996) Evolutionary identification of macro-mechanical models. In: Angeline PJ, Kinnear, Jr KE (eds) Advances in Genetic Programming 2, MIT Press, Cambridge, MA, USA, 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, Australia, 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, ACM, Montreal, pp 1115–1122

Silva S, Vanneschi L (2010) State-of-the-art genetic programming for predicting human oral bioavailability of drugs. In: 4th International Workshop on Practical Applications of Computational Biology and Bioinformatics 2010 (IWPACBB 2010), Springer, Guimarães, Portugal, vol 74, pp 165–173

Spector L (2001) Autoconstructive evolution: Push, pushGP, and pushpop. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), Morgan Kaufmann, San Francisco, California, USA, 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), ACM, Philadelphia, Pennsylvania, USA, pp 401–408

Spector L, Robinson A (2002) Genetic programming and autoconstructive evolution with the push programming language. Genetic Programming and Evolvable Machines 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, ACM Press, Washington DC, USA, vol 2, 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, Springer-Verlag, Kinsale, Ireland, LNCS, vol 2278, 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. Genetic Programming and Evolvable Machines 14(1):3–29