Evolving a Digital Multiplier with the PushGP Genetic Programming System

Thomas Helmuth
Computer Science
University of Massachusetts
Amherst, MA 01003
thelmuth@cs.umass.edu

Lee Spector
Cognitive Science
Hampshire College
Amherst, MA 01002
lspector@hampshire.edu

ABSTRACT

A recent article on benchmark problems for genetic programming suggested that researchers focus attention on the digital multiplier problem, also known as the "multiple output multiplier" problem, in part because it is scalable and in part because the requirement of multiple outputs presents challenges for some forms of genetic programming [20]. Here we demonstrate the application of stack-based genetic programming to the digital multiplier problem using the PushGP genetic programming system, which evolves programs expressed in the stack-based Push programming language. We demonstrate the use of output instructions and argue that they provide a natural mechanism for producing multiple outputs in a stack-based genetic programming context. We also show how two recent developments in PushGP dramatically improve the performance of the system on the digital multiplier problem. These developments are the "ULTRA" genetic operator, which produces offspring via "Uniform Linear Transformation with Repair and Alternation" [12], and "lexicase selection," which selects parents according to performance on cases considered sequentially in random order [11]. Our results using these techniques show not only their utility, but also the utility of the digital multiplier problem as a benchmark problem for genetic programming research. The results also demonstrate the flexibility of stack-based genetic programming for solving problems with multiple outputs and for serving as a platform for experimentation with new genetic programming techniques.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program synthesis*

General Terms

Algorithms

GECCO'13 Companion, July 6–10, 2013, Amsterdam, The Netherlands.

Keywords

stack-based genetic programming; Push; PushGP; digital multiplier problem; lexicase selection; ULTRA operator

1. INTRODUCTION AND RELATED WORK

A long-term goal of the field of genetic programming is to provide systems that can produce arbitrary computer programs automatically, generating software from specifications without further intervention by human programmers [4]. Most of the work actually being conducted in the field, however, is focused much more narrowly on the automated discovery of small programs that solve specific, well-constrained computational problems. While this focus has produced significant results, both in terms practical problem-solving technologies and in terms of foundational theoretical understanding, many researchers in the field appreciate the need to improve the genetic programming technique in fundamental ways in order to reach the stated long term goals for the field.

It is within this context that recent articles on benchmark problems for genetic programming suggested that the digital multiplier problem, also known as the multiple output multiplier problem, be used to assess the performance of genetic programming systems [20, 7]. In the digital multiplier problem, we seek a program that takes two sets of boolean inputs, each of which encodes a number in binary, and produces a set of boolean outputs that encodes the product of the input numbers. While this is not an intrinsically difficult problem—it is regularly solved anew by students of introductory digital circuit design—it does have several properties that make it attractive as a benchmark problem for genetic programming. For example, digital multiplication functions have been well studied, the problem can be scaled up to involve inputs of any number of bits, and the fitness values are not chunked in powers of two. These factors make it a good replacement for "toy" boolean problems, such as the parity and multiplexer problems, that have been heavily used for benchmarking in the genetic programming literature.

Of particular interest, however, is the fact that solutions to the digital multiplier problem must produce multiple outputs, one for each bit of the product of the input numbers. The most common genetic programming systems, which manipulate and produce programs in the form of function trees, are most commonly applied to problems that require the production of only a single output, which is produced by the function call at the root of the tree.

It is certainly possible to use tree-based genetic programming to evolve programs that produce multiple outputs. One technique for doing so is to use functions that act by "side effect" on multiple global state variables rather than by returning a value to the function higher in the tree; indeed, several of the classic examples of genetic programming use this technique [4].

It is certainly possible to use tree-based genetic programming to evolve programs that produce multiple outputs, and indeed several of the classic examples of genetic programming build solutions out of functions that act by "side effect" on global state variables rather than by returning a value to the function higher in the tree [4].

Nonetheless, the functional-style representations of tree-based genetic programming have often suggested applications only to problems with single outputs, and this may be the reason that multiple-output problems have received relatively little attention in the field.

In this paper we demonstrate the use of PushGP, a stack-based genetic programming system, to evolve digital multipliers. Stack-based genetic programming evolves programs that pass data among instructions primarily by means of data stacks, rather than by means of return values passed through syntactically-nested function calls. This change in perspective makes it easier to imagine ways in which the technique can be applied to problems that require the production of multiple outputs.

One option for approaching the digital multiplier problem with stack-based genetic programming would be to evolve programs which take all of the inputs pre-loaded on a stack and leave all of the outputs on a stack after the program completes execution. However, prior work on stack-based genetic programming with the Push programming language [15, 13] has demonstrated that it is often useful to provide input instructions that can be called as needed, and possibly repeatedly, to re-push inputs onto stacks. In a similar spirit, it is natural to provide output instructions that take arguments from stacks and designate them as program outputs. In the present paper we use output instructions to designate outputs in solving the digital multiplier problem. The "output instruction" idea could also be applied in tree-based genetic programming, although the question of what the output instructions would return to their callers would have to be addressed, and to our knowledge this has not yet been investigated.

Prior work has been done to evolve digital multipliers using a variety of evolutionary algorithms, although none of them use traditional tree-based programs or stack-based genetic programming. Much of the work has been done using Cartesian genetic programming, in which designated output nodes play a role somewhat analogous to output instructions [18, 19, 8]. Others have used techniques such as genetic algorithms, including multiobjective genetic algorithms [2, 1], as well as enzyme genetic programming [5] to evolve digital multipliers. Many of these papers evolve 2-bit multipliers, and few tackle larger problems. Of particular note is an approach that uses a genetic algorithm that partitions the outputs and test cases into multiple different problems to evolve, and then reassembles a full solution afterward. This method was used to evolve a 5-bit digital multiplier [16]. We are not aware of a method that evolves single programs that has evolved digital multipliers this large.

In addition to demonstrating how output instructions can be used with stack-based genetic programming to solve the digital multiplier problem, the present paper also demonstrates two recently developed enhancements to PushGP. One of these enhancements is a new genetic operator, called "ULTRA," which produces offspring using "Uniform Linear Transformation with Repair and Alternation" [12]. The other enhancement is a parent selection algorithm called "lexicase selection," which selects parents according to performance on cases considered sequentially in random order [11]. Each of these enhancements improves the performance of PushGP on the digital multiplier problem significantly, and the effect of the two enhancements when combined is truly dramatic. These results demonstrate not only the utility of the ULTRA operator and lexicase selection, but also the utility of the digital multiplier problem as a benchmark problem for genetic programming research.

In the following sections we first describe the PushGP genetic programming system, with which we performed all of the experiments described in this paper. We then describe the two innovative techniques that we use in conjunction with PushGP in our experiments, lexicase selection and the ULTRA operator. These descriptions are followed by a detailed presentation of the digital multiplier problem and its implementation in PushGP. We then present and discuss our experimental results before concluding.

2. PUSH AND PUSHGP

Push is a programming language that was designed specifically for use in genetic programming and other evolutionary computation systems, as the language in which evolving programs are expressed [10, 15, 13]. Push is a stack-based programming language which is similar in some ways to others that have been used for GP, e.g. the stack-based language used in [9]. It is a postfix language in which literals, when encountered by the interpreter, are pushed onto data stacks, and instructions, when encountered by the interpreter, act on data taken from stacks and return results on 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. The convention in Push regarding instructions that are executed in contexts that provide insufficient arguments on the relevant stacks is that these instructions 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. However, these features of Push are not used in the experiments presented in the present paper, so they will not be described here.

Another somewhat novel feature of Push, in the context of stack-based languages, is that Push programs may be hierarchically structured with parentheses. This hierarchical structure affects how code-manipulation instructions work, and it also affects the ways that genetic operators operate on programs. For example, 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-

To select a parent for use in a genetic operation:

1. `Initialize`:
   (a) Set `candidates` to be the entire population.
   (b) Set `cases` to be a list of all of the fitness cases in random order.
2. `Loop`:
   (a) Set `candidates` to be the subset of the current `candidates` that have exactly the best fitness of any individual currently in `candidates` for the first case in `cases`.
   (b) If `candidates` or `cases` contains just a single element then return a randomly selected individual from `candidates`.
   (c) Otherwise remove the first case from `cases` and go to `Loop`.

Figure 1: Pseudocode for the lexicase selection algorithm used in the experiments in this paper.

expression, and with crossover replacing a sub-expression with a subexpression randomly selected from another program in the population.

Push and PushGP implementations now exist 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.[1]

3. LEXICASE SELECTION

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 different actions (or to respond in different "modes") for inputs that belong to different classes that may not be known in advance [11, 17]. We expect lexicase selection to improve the performance of PushGP on the digital multiplier problem, since this problem requires programs to exhibit different behavior on different test cases. As we will discuss later, a small subset of the test cases require different outputs on certain bits, making the search space difficult to navigate for traditional selection operators that act on summed errors.

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. The "lexicase" name stems from the similarity of this sequential consideration of cases to the "lexicographic ordering" of character strings, in which the first character has the largest effect, the second matters only for ordering within the set of strings that are equal in their first characters, and so on. The lexicase selection algorithm used here, which is called "global pool, uniform random sequence, elitist lexicase parent selection" in [11], is described in pseudocode in Figure 1.

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

4. 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 combines them in a way that makes each part of each program equally likely to appear in the resulting child and equally likely to be modified by mutation [12]. ULTRA acts on hierarchically structured programs but treats them as linear structures while including each element of the structures 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.

The ULTRA operator takes two parent programs and produces from them a single child program. It first "linearizes" each parent into a flat sequence that includes a token for each literal, instruction, and parenthesis in the parent program. It then traverses the linearized parents, building the child as a linear sequence of tokens taken from the parents. This traversal begins by adding the first token of the first parent to the child. After this first step, and after each further token is added to the child sequence, there is a fixed probability of alternating between parents; that is, of moving the traversal index to an index in the other parent program. The probability of alternating at any given index is specified as the "alternation rate." When alternating between parents, the index within the new parent from which the next token will be taken 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. After deciding whether to alternate or not, the next token from the current parent is added to the child, and the traversal index is incremented. The construction of the child terminates when the token index exceeds the size of the current parent or when it exceeds the maximum program size.

After the child sequence has been created through traversal of the parents, it is subjected to a uniform mutation during which each token has uniform probability of being deleted or replaced by any other token, including all literals and instructions used for the problem and open and close parentheses. The probability of any one 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 or mutation may create a syntactically invalid child program; in this case, the 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 from front to back, and then one from back to front) in order to ensure balance and to avoid bias in the structures produced by re-

Table 1: A list of the Push instructions used in our digital multiplier experiments. For the n-bit digital multiplier problem, there are $2n$ input instructions and $2n$ output instructions.

Boolean Stack	and, or, xor, invert_first_then_and, dup, swap, rot
Input / Output	in0, ..., in2n, out0, ..., out2n

pair. After repair, the tokenized child is transformed back into a hierarchical Push program.

5. THE DIGITAL MULTIPLIER PROBLEM

The digital multiplier problem requires the system to create a digital circuit that multiplies two binary numbers. An n-bit digital multiplier circuit takes two n-bit numbers represented in binary as input and multiplies them together to create a $2n$-bit number as output. As noted above, this problem was recommended by the authors of recent articles on genetic programming benchmarks as an alternative to other boolean problems such as multiplexer and parity, since it offers difficulties not seen in those problems [7, 20]. In particular, it forces the evolving programs to output multiple values and allows for trials of problems of varying sizes without constraining fitness values to powers of two.

The boolean n-bit digital multiplier problem uses each possible assignment of 0 and 1 to each of the $2n$ input bits to produce 2^{2n} test cases, each with $2n$ output bits. The fitness (error) of each test case is the number of bits that the program gets wrong compared to the desired output bits. Thus the error for a test case can be an integer between 0 and $2n$. The instruction set recommended in [18] uses four boolean functions: AND, AND with one input inverted, XOR, and OR. Human-designed digital multipliers use a series of full and half adders; adding these as instructions would presumably make the problem easier. A half adder can be represented by one AND and one XOR instruction, so we would expect to see extensive use of these instructions in our evolved programs.

In our implementation of the digital multiplier problem in PushGP, we use not only input instructions (which are common in Push), but also output instructions. In our implementation we provide one input instruction for each input bit, and one output instruction for each output bit. For example, the 2-bit digital multiplier has input instructions in0, in1, in2, and in3, each of which pushes the test case's relevant input bit onto the boolean stack. It also uses output instructions out0, out1, out2, and out3, which set the given output bit to the top item on the boolean stack and pop that item. Each time an output instruction is called, the output for that bit is overwritten by the top item on the boolean stack so that only the last such instruction executed affects the behavior of the program. If a specific output instruction is never called within the program, that bit is considered wrong in each test case, but no further penalty is given.

Beyond the input and output instructions, we use the boolean stack instructions found in the top row of Table 1. The first four of these are the instructions recommended by Walker and Miller [18], and the other three are typical stack manipulation instructions that are often used in Push. The boolean_dup instruction duplicates the top item on the boolean stack, the boolean_swap instruction swaps the top two items on the boolean stack, and boolean_rot moves the third item on the boolean stack to the top of the stack. Our random code generator chooses to either use a boolean stack instruction or an input/output instruction randomly, and then selects from the chosen category uniformly. This ensures that the ratio of boolean stack instructions to input/output instructions remains 50% for different sizes of the problem, even though there are more input and output instructions in larger versions of the problem.

Note that for the digital multiplier problem as defined here and with this instruction set, parentheses within programs have no semantic effect. Since ULTRA also does not require parentheses, we could have entirely left them out of our evolving programs for those runs. But, since the genetic operators crossover and mutation that we use for comparison require them in PushGP, we left them in our ULTRA runs as well.

Table 2: Parameters for experiments.

Digital Multiplier Problem	2-bit	3-bit
Runs Per Condition	100	100
Population Size	1000	2000
Max Generations	500	500
Max Program Size	400	800
Max Inital Program Size	200	400
Max Size for Mutation Code	30	30

Table 3: Genetic operators used in the "Normal" and "Lexicase" runs. Crossover and mutation are the normal PushGP operators which are nearly identical to the standard operators in tree-based genetic programming; they replace one sub-expression in the program with another subexpression taken from another individual (crossover) or generated randomly (mutation). Crossover and mutation use unbiased node selection. Simplification [3] is an operator on one parent program that attempts to create a smaller program by randomly deleting instructions and/or removing parentheses; it retains the simplified program only if it has the same error vector as the parent program. Otherwise the parent is itself copied to the child population.

Crossover	80%
Mutation	10%
Simplification	5%
Reproduction	5%

6. EXPERIMENTS AND RESULTS

To test PushGP on the digital multiplier problem, we performed runs on both the 2-bit multiplier and 3-bit multiplier problems. The major parameters that we used in our exper-

Table 4: In the runs where we used ULTRA, we used it as the only genetic operator. These are the ULTRA parameters we used for those runs.

ULTRA Mutation Rate	1%
ULTRA Alternation Rate	1%
ULTRA Alternation Alignment Deviation	10

Table 5: Results on the 2-bit digital multiplier problem. Each condition used 100 runs. CE is the computational effort and MBF is the mean best fitness of the run. The last column gives 2-tailed p-values from unpaired t-tests that examine whether the MBF differs from that of lexicase + ULTRA, shown in the last row of the table.

Condition	Successes	CE	MBF	p-value
Normal	12	6,893,000	0.144	< 0.001
Lexicase	90	595,000	0.006	0.005
ULTRA	57	2,440,000	0.056	< 0.001
Lex+ULTRA	99	192,000	0.0006	-

Table 6: Results on the 3-bit digital multiplier problem. Each condition used 100 runs. MBF is the mean best fitness of the run. No runs found perfect solutions. The last column gives 2-tailed p-values from unpaired t-tests that examine whether the MBF differs from that of lexicase + ULTRA, shown in the last row of the table.

Condition	MBF	p-value
Normal	0.89	< 0.001
Lexicase	0.39	< 0.001
ULTRA	0.53	< 0.001
Lex+ULTRA	0.12	-

iments can be found in Table 2. In our runs, we used either tournament selection with tournament size 7 or lexicase selection. For genetic operators, we either used the "normal" genetic operators with the probabilities in Table 3, or ULTRA with the parameters found in Table 4. For program initialization, we created random programs with sizes uniformly chosen between 1 and the max initial program size found in Table 2. We conducted sets of runs using four different parameter conditions: normal (tournament selection and normal genetic operators), lexicase (with normal genetic operators), ULTRA (with tournament selection), and lexicase + ULTRA.

In order to test the performance of each condition, we measured the success rate, computational effort, and the mean best fitness for each set of runs. Mean best fitness is the mean of the best individual fitnesses attained in each run. For all runs described here, fitness is defined as a measure of error with lower numbers being better and solutions having fitness values of zero. The fitnesses given here are the mean errors across test cases, not the sums of those errors. As recommended in [6, 7], we conducted t-tests comparing the mean best fitnesses of each condition with the best set of runs.

We also present the success rate and computational effort of each run. The success rate is the number of runs that find a perfect solution. Computational effort measures the expected number of individuals that the genetic programming algorithm needs to evaluate in order to have a 99% confidence of finding a solution. A lower computational effort means that fewer fitness evaluations have to be made in order to find a solution. Computational effort was computed as described by Koza [4, pp. 99–103]: We first calculate $P(M, i)$, the cumulative probability of success by generation i with population size M; this is the number of runs that succeeded on or before the ith generation, divided by the number of runs conducted. $I(M, i, z)$, the number of individuals that must be processed to produce a solution by generation i with probability greater than z (here $z = 99\%$) is then calculated as:

$$I(M, i, z) = M * (i + 1) * \left\lceil \frac{\log(1 - z)}{\log(1 - P(M, i))} \right\rceil$$

The minimum of $I(M, i, z)$ over all generation values of i is defined to be the "computational effort" required to solve the problem. The measures of success rate and computational effort give an idea as to how efficient the algorithm is at finding full solutions to a problem.

The results of our runs on the 2-bit digital multiplier problem can be found in Table 5. Using both lexicase selection and ULTRA resulted in the best performance, which found 100% correct programs in almost every run. The computational effort for lexicase + ULTRA is much lower than with lexicase selection only, ULTRA only, or neither. The mean best fitness of the lexicase + ULTRA runs is better than any of the other conditions. We conducted unpaired t-tests between the lexicase + ULTRA runs and all other conditions' mean best fitnesses, and found the differences to be significant at the 0.01 level each time.

Table 6 presents the results of our runs on the 3-bit digital multiplier problem. In this case, no runs found perfect solutions, so we do not report solution rates or computational effort. Lexicase + ULTRA produced the best mean best fitness, with the differences between it and the other runs significant at the 0.01 level. We did conduct a small set of runs on the 3-bit problem using a larger population size (5000) and larger max generations (4000). Many of these runs did find perfect solutions, but we were unable to perform enough runs with these larger parameters to produce meaningful results in the time that we had available prior to the deadline for this paper.

7. DISCUSSION

Based on our results, both lexicase selection and ULTRA significantly improve the performance of PushGP on the digital multiplier problem. Let us consider each of their effects and why they may improve performance.

Lexicase selection puts equal pressure on solving each test case, which in turn makes it helpful on "modal" problems where different test cases require qualitatively different actions to solve them [11]. In the digital multiplier problem, some output bits should return False in almost every test case. For example, in 2-bit digital multiplier, the first bit is False in every test case besides when both binary inputs represent the number 3, in which case it is True. So, a program can achieve near-perfect fitness on the first bit by simply outputting False for it in every test case. Using normal summed-error fitness with a selection method such as tournament selection, a program can essentially ignore this first bit and get a fitness of 1 by just solving the other three bits perfectly. This leads evolution into a local optimum, where it is difficult to solve the 3 times 3 test case without disrupting another test case more significantly.

Lexicase selection avoids this problem by putting equal selection pressure on solving each test case. Programs that correctly compute the first bit of the 3 times 3 test case perfectly will be selected for when that test case is near the

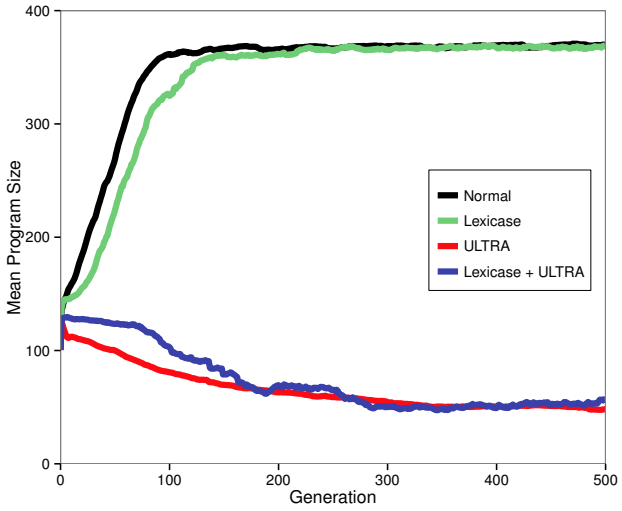

Figure 2: Mean program size for each set of 2-bit digital multiplier runs over evolutionary time.

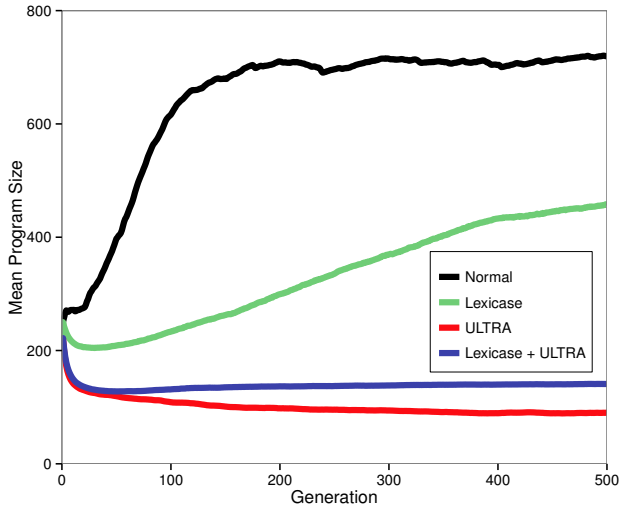

Figure 3: Mean program size for each set of 3-bit digital multiplier runs over evolutionary time.

front of the lexicase random test case ordering. This helps evolution avoid the local optimum where the first output bit is essentially ignored, leading to a solution much more easily. We believe this modal effect is responsible for the improved performance of lexicase selection over tournament selection on the digital multiplier problem.

ULTRA's contribution to the improved performance of PushGP on the digital multiplier problem may stem from a variety of sources, from the way it changes evolutionary dynamics to how it seems to inhibit code growth. Figures 2 and 3 plot the mean program sizes of our 2-bit and 3-bit digital multiplier problem runs against evolutionary time. In both of these figures we see that program sizes are much larger in the runs using normal genetic operators than in runs using ULTRA as the only genetic operator. In fact, program sizes tend to decrease or remain relatively constant using ULTRA, whereas we see large amounts of code growth that levels off near maximum program sizes when using normal genetic operators. From these figures and the performance measures of our runs, we infer that ULTRA guides evolution toward programs that perform well on the problem while maintaining reasonable program sizes. We discuss ULTRA's ability to avoid code growth elsewhere [12]; in short, we believe the way ULTRA treats any given part of the program uniformly makes it difficult for bloated programs to reproduce successfully.

Based on a brief scan of both 2-bit and 3-bit solution programs that our system found, our system did not produce much programmatic modularity. That is, we did not see much use of the boolean_dup instruction, which is the only instruction we used in our instruction set that allows for reproducing work done previously in the program without repeating the same instructions. On the other hand, we did not check for evolutionary modularity, in which a program segment (such as a few instructions that implement a half adder) appears many times throughout both single programs and the population as a whole. Based on the results found in Cartesian GP [18], we would expect that modularity-improving features such as tags [14] may improve the system's performance. Surprisingly, in experiments not presented here the addition of tags to the instruction set actually decreased the performance of PushGP on both the 2-bit and 3-bit digital multiplier problems.

8. CONCLUSIONS

We have used the stack-based PushGP genetic programming system to search for solutions to the 2-bit and 3-bit digital multiplier problems. We have shown how output instructions can be used in finding solutions to these problems, which require multiple bits to be output during each program execution. Our efforts have included using two recent developments in PushGP, lexicase selection and the ULTRA genetic operator, both of which significantly improve the performance of PushGP on the digital multiplier problem.

At this point we have only evolved perfect solutions to the 2-bit and 3-bit digital multiplier problems, with the 3-bit version requiring many extra fitness evaluations. We hope that future work on this problem will allow us to solve the 3-bit and 4-bit multiplier problems more readily, and possibly the 5-bit problem. As the problem difficulty seems to grow exponentially with the size of the problem, new techniques will likely have to be used to solve 5-bit or greater problems. In particular, since human designs of digital multiplier circuits rely on large amounts of modularity, we presume that modularity-encouraging techniques such as tags [14] and code manipulating instructions may dramatically help PushGP to more efficiently solve these larger problems. The work that has evolved the largest digital multiplier to our knowledge used separately evolved output chromosomes to evolve a 5-bit digital multiplier [16].

In our comparison runs that used tournament selection and normal genetic operators such as crossover and mutation, we chose typical parameters for those methods but did not try variants that may have led to improved performance. Since the main focus of this paper is not to argue for lexicase selection and ULTRA, we consider this to be sufficient. But, future work arguing for lexicase selection or ULTRA versus

more traditional approaches should make more effort to find more optimized parameter settings for these methods.

We believe that tree-based genetic programming could be adapted to evolve solutions for this problem. The main hurdle for tree-structured programs is that they often only return a single value, whereas this problem requires multiple output bits. This limitation may be avoided in a variety of ways. One such way would be to include output instructions similar to the ones used here. In trees, these instructions would need to return values, which would likely just be the value that they output. Another possibility is to use multi-chromosomal evolution, in which each chromosome is a function that returns the value for a single output bit [19]. This allows each chromosome to be evolved independently, such that tree-based programs can remain entirely stateless while still returning multiple output bits.

9. ACKNOWLEDGMENTS

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.

10. REFERENCES

[1] C. A. C. Coello, A. D. Christiansen, and A. H. Aguirre. Use of evolutionary techniques to automate the design of combinational circuits. *International Journal of Smart Engineering System Design*, 2:299–314, 2000.

[2] C. A. Coello Coello and A. H. Aguirre. Design of combinational logic circuits through an evolutionary multiobjective optimization approach. *Artif. Intell. Eng. Des. Anal. Manuf.*, 16(1):39–53, Jan. 2002.

[3] J. Klein and L. Spector. Unwitting distributed genetic programming via asynchronous JavaScript and XML. In D. Thierens, H.-G. Beyer, J. Bongard, J. Branke, J. A. Clark, D. Cliff, C. B. Congdon, K. Deb, B. Doerr, T. Kovacs, S. Kumar, J. F. Miller, J. Moore, F. Neumann, M. Pelikan, R. Poli, K. Sastry, K. O. Stanley, T. Stutzle, R. A. Watson, and I. Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1628–1635, London, 7-11 July 2007. ACM Press.

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

[5] M. A. Lones and A. M. Tyrrell. Biomimetic representation with genetic programming enzyme. *Genetic Programming and Evolvable Machines*, 3(2):193–217, 2002.

[6] S. Luke and L. Panait. Is the perfect the enemy of the good? In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 820–828, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[7] J. McDermott, D. R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong, and U.-M. O'Reilly. Genetic programming needs better benchmarks. In T. Soule, A. Auger, J. Moore, D. Pelta, C. Solnon, M. Preuss, A. Dorin, Y.-S. Ong, C. Blum, D. L. Silva, F. Neumann, T. Yu, A. Ekart, W. Browne, T. Kovacs, M.-L. Wong, C. Pizzuti, J. Rowe, T. Friedrich, G. Squillero, N. Bredeche, S. Smith, A. Motsinger-Reif, J. Lozano, M. Pelikan, S. Meyer-Nienberg, C. Igel, G. Hornby, R. Doursat, S. Gustafson, G. Olague, S. Yoo, J. Clark, G. Ochoa, G. Pappa, F. Lobo, D. Tauritz, J. Branke, and K. Deb, editors, *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 791–798, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[8] J. F. Miller, D. Job, and V. K. Vassilev. Principles in the evolutionary design of digital circuits - part i. *Genetic Programming and Evolvable Machines*, 1(1-2):7–35, Apr. 2000.

[9] T. Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 148–153. IEEE Press, 1994.

[10] L. Spector. Autoconstructive evolution: Push, pushGP, and pushpop. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 137–146, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[11] L. Spector. Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report. In K. McClymont and E. Keedwell, editors, *1st workshop on Understanding Problems (GECCO-UP)*, pages 401–408, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[12] L. Spector and T. Helmuth. Uniform linear transformation with repair and alternation in genetic programming. In R. Riolo, E. Vladislavleva, and J. H. Moore, editors, *Genetic Programming Theory and Practice XI*, page In preparation. Springer, 2013.

[13] L. Spector, J. Klein, and M. Keijzer. The push3 execution stack and the evolution of control. In H.-G. Beyer, U.-M. O'Reilly, D. V. Arnold, W. Banzhaf, C. Blum, E. W. Bonabeau, E. Cantu-Paz, D. Dasgupta, K. Deb, J. A. Foster, E. D. de Jong, H. Lipson, X. Llora, S. Mancoridis, M. Pelikan, G. R. Raidl, T. Soule, A. M. Tyrrell, J.-P. Watson, and E. Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1689–1696, Washington DC, USA, 25-29 June 2005. ACM Press.

[14] L. Spector, B. Martin, K. Harrington, and T. Helmuth. Tag-based modules in genetic programming. In N. Krasnogor, P. L. Lanzi, A. Engelbrecht, D. Pelta, C. Gershenson, G. Squillero, A. Freitas, M. Ritchie, M. Preuss, C. Gagne, Y. S. Ong, G. Raidl, M. Gallager, J. Lozano, C. Coello-Coello, D. L. Silva, N. Hansen,

S. Meyer-Nieberg, J. Smith, G. Eiben, E. Bernado-Mansilla, W. Browne, L. Spector, T. Yu, J. Clune, G. Hornby, M.-L. Wong, P. Collet, S. Gustafson, J.-P. Watson, M. Sipper, S. Poulding, G. Ochoa, M. Schoenauer, C. Witt, and A. Auger, editors, *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1419–1426, Dublin, Ireland, 12-16 July 2011. ACM.

[15] L. Spector and A. Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, Mar. 2002.

[16] J. Torresen. Evolving multiplier circuits by training set and training vector partitioning. In *Evolvable Systems: From Biology to Hardware*, pages 228–237. Springer, 2003.

[17] L. Trujillo, E. Naredo, Y. Martínez, and L. Spector. A behavior-based analysis of modal problems. In *2nd workshop on Understanding Problems (GECCO-UP)*, 2013. To appear.

[18] J. A. Walker and J. F. Miller. The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 12(4):397–417, Aug. 2008.

[19] J. A. Walker, K. Volk, S. L. Smith, and J. F. Miller. Parallel evolution using multi-chromosome cartesian genetic programming. *Genetic Programming and Evolvable Machines*, 10(4):417–445, Dec. 2009. Special issue on parallel and distributed evolutionary algorithms, part I.

[20] D. R. White, J. Mcdermott, M. Castelli, L. Manzoni, B. W. Goldman, G. Kronberger, W. Jaśkowski, U.-M. O'Reilly, and S. Luke. Better gp benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines*, 14(1):3–29, Mar. 2013.