Solving Uncompromising Problems with Lexicase Selection

Thomas Helmuth, Lee Spector *Member, IEEE,* James Matheson

***Abstract*—We describe a broad class of problems, called "uncompromising problems," characterized by the requirement that solutions must perform optimally on each of many test cases. Many of the problems that have long motivated genetic programming research, including the automation of many traditional programming tasks, are uncompromising. We describe and analyze the recently proposed "lexicase" parent selection algorition and show that it can facilitate the solution of uncompromising problems by genetic programming. Unlike most traditional parent selection techniques, lexicase selection does not base selection on a fitness value that is aggregated over all test cases; rather, it considers test cases one at a time in random order. We present results comparing lexicase selection to more traditional parent selection methods, including standard tournament selection and implicit fitness sharing, on four uncompromising problems: finding terms in finite algebras, designing digital multipliers, counting words in files, and performing symbolic regression of the factorial function. We provide evidence that lexicase selection maintains higher levels of population diversity than other selection methods, which may partially explain its utility as a parent selection algorithm in the context of uncompromising problems.**

***Index Terms*—parent selection, lexicase selection, tournament selection, genetic programming, PushGP.**

I. Introduction

GENETIC programming problems generally involve test cases that are used to determine the performance of programs during evolution. While some classic genetic programming problems, such as the artificial ant problem and the lawnmower problem [1], involve only single test cases, most others involve large numbers of tests. There are several ways in which a genetic programming system can take multiple test cases into consideration during parent selection—that is, when determining which individuals to use as source material when producing offspring for the next generation—and the best choice may depend on the type of problem being solved.

For some problems it may be be appropriate to use methods that seek "compromises" among the different test cases. For example, we can imagine a problem involving control of a simulated wind turbine in which some test cases focus on performance in low wind conditions while others focus on performance in high wind conditions. It may not be possible to optimize performance on all of these test cases simultaneously, and some sort of compromise may therefore be necessary. Many common parent selection approaches, such as tournament selection, introduce compromises between test cases by aggregating the performance of an individual on those test cases into a single fitness value. The method of compromise may be as simple as summing the test case errors, or their squares, into a single error value; more complex methods such as implicit fitness sharing [2] dynamically weight test cases based on population statistics before aggregating them.

By contrast, we wish to consider what we call "uncompromising" problems: problems for which any acceptable solution must perform as well on each test case as it is possible to perform on that test case; that is, an uncompromising problem is a problem for which it is not acceptable for a solution to perform sub-optimally on any one test case in exchange for good performance on others. More formally, consider a problem defined by the set of test cases T where the set of programs in the search space is P and $p_j(t_i)$ is the error produced by program $p_j \in P$ on test case $t_i \in T$ with lower error being better. This problem is *uncompromising* if a program $p \in P$ would be considered a solution to the problem if and only if $p(t_i) \leq p_j(t_i)$ for all $t_i \in T$ and $p_j \in P$.

While this might at first appear to be a narrow group of problems, we believe that many important problems fall into this class. For example, all of the Boolean function induction problems commonly used in the genetic programming literature are uncompromising (e.g. the multiplexer problems in [1]), as are those symbolic regression problems for which a program must achieve an error of zero on all test cases in order to count as a solution. Other examples from mathematics, which we discuss further below, are problems of finding functional representations of terms in finite algebras; only programs that perform optimally on all test cases count as solutions to these problems.

Of possibly greater significance, the set of uncompromising problems includes most "traditional programming" or "software synthesis" problems, in which one seeks to automatically produce general software (which may require the use of multiple data types, conditionals, and loops) from specifications or behavioral tests. The automation of traditional programming has been presented as one of the primary goals for research in genetic programming at least since Koza's seminal 1992 book [1], and recent assessments of the state of the field

Manuscript received November 3, 2013; revised April 15, 2014 and August 5, 2014. This material is based upon work supported by the National Science Foundation under Grants No. 1017817, 1129139, and 1331283. 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.

T. Helmuth is a doctoral student in the Department of Computer Science, University of Massachusetts, Amherst, MA, 01003 USA e-mail: thelmuth@cs.umass.edu.

L. Spector is with Hampshire College, Amherest, MA 01002 USA email: lspector@hampshire.edu.

J. Matheson is a graduate of Hampshire College.

have highlighted both the importance and the difficulty of these kinds of problems [3]. Most traditional programming problems involve large numbers of test cases, and programs count as solutions to these problems only if they pass all tests[1]. Several recent projects at the intersection of genetic programming and software engineering, on automatic program repair and improvement, involve uncompromising problems framed in terms of tests that must all pass for the system to be considered successful [4], [5].

We note that uncompromising problems do not necessarily require "perfect" (zero error) performance on every test case, although problems that require perfect performance on every test case are indeed uncompromising by definition. In this context it is interesting to consider the perspective that Luke and Panait put forward in their paper titled "Is the Perfect the Enemy of the Good?" [6]. Luke and Panait argue that comparisons of genetic programming techniques based on solution counts could be misleading for types of problems in which "ideal solutions" are unlikely to be found, and for which one seeks a program with minimal—but probably not zero—error. We have no quarrel with Luke and Panait on this point *in the context of such problems*. But for many uncompromising problems, including the problems that we consider in this paper and for a great many potentially important applications of genetic programming, programs that do not pass all tests do not count as solutions. The value of a genetic programming technique with respect to such problems can only be meaningfully assessed in terms of the number of successes that it produces.

In this paper we present a relatively new parent selection algorithm, lexicase selection, which was originally proposed for solving "modal problems" in which programs must perform qualitatively different actions on different test cases [7]. Here we expand the scope and analysis of lexicase selection by giving evidence that it greatly improves genetic programming's ability to find solutions to uncompromising problems compared to selection techniques that permit compromises between test cases. We give a detailed description of the lexicase selection algorithm and demonstrate its effectiveness on four problems: the problem of finding discriminator terms in finite algebras, the problem of designing digital multipliers, the problem of replicating the core functionality of the "wc" word-count utility program, and symbolic regression of the factorial function. We give evidence that lexicase selection also maintains high levels of population diversity, possibly contributing to its utility as a parent selection algorithm.

The next section discusses the lexicase selection algorithm and what differentiates it from other selection methods. We then present a variety of related evolutionary computation techniques. Section IV describes the design and implementation of our experiments, including the genetic programming systems we apply, the problems we attempt to solve, and the performance measures we use in our comparisons. Section V presents the results of our experiments comparing lexicase selection to both standard tournament selection (at multiple tournament sizes) and implicit fitness sharing.

[1]We note that for many large-scale software applications, it is not normal or reasonable to expect all test cases to be passed. Indeed, most applications are released with known bugs. Nonetheless, the goal of passing all test cases is a useful approximation even for these cases, and is strictly required for mission critical programs and in other programming contexts.

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 test cases in random order.
2) `Loop`:
   a) Set `candidates` to be the subset of the current `candidates` that have exactly the best performance of any individual currently in `candidates` for the first case in `cases`.
   b) If `candidates` contains just a single individual then return it.
   c) If `cases` contains just a single test case then return a randomly selected individual from `candidates`.
   d) Otherwise remove the first case from `cases` and go to `Loop`.

Fig. 1. Pseudocode for the lexicase selection algorithm.

II. Lexicase Selection

Lexicase[2] selection is a method for selecting individuals from a population to serve as parents of new individuals. It can be used any time that potential parents are assessed with respect to multiple test cases.

The lexicase selection algorithm used here, which is called "global pool, uniform random sequence, elitist lexicase parent selection" in [7], is described in pseudocode in Figure 1. In each parent selection event, the lexicase selection algorithm first randomly orders the test cases. It then eliminates any individuals in the population that do not have the best performance on the first test case.[3] Assuming that more than one individual remains, it then loops, eliminating any individuals from the remaining candidates that do not have the best performance on the second test case. This process continues until only one individual remains and is selected, or until all test cases have been used, in which case it randomly selects one of the remaining individuals.

Lexicase selection sometimes selects individuals that perform well on a relatively small number of test cases. This differs from most other selection algorithms, which select individuals based on aggregations of performance on all test cases. Lexicase selection may select individuals that perform very poorly on some test cases if they excel on a combination of others. As such, lexicase often selects "specialist" individuals that solve parts of the problem extremely well. Although these individuals may have worse summed error across all test cases, the hope is they will be able to reproduce

[2]The term "lexicase" has been used previously in unrelated work [8].

[3]This retention of only "the best" could be relaxed to retain all individuals within some distance of the best, but the form of lexicase selection used here is "elitist" in that it retains only the best.

in ways that pass on their preeminence on certain cases while improving with respect to others. In order to give every test case equal selection pressure, each lexicase selection event uses a randomly shuffled list of test cases to determine which test cases are treated as most important.

Lexicase selection was originally developed for modal problems, where by "modal" we mean problems that require the solution program to take dissimilar actions—that is, to have different modes of operation—for different input test cases [7]. For evolution to be successful on these problems, it must produce programs or sections of programs that can perform disparate actions; in particular, it must be able to emphasize the requirements of different cases and combinations of cases in its selection procedure. Tournament selection and related methods do not do a good job here; more often, one or two of the modes receive the majority of the attention of selection, and others are neglected. Lexicase selection, on the other hand, emphasizes different test cases with each selection event, and therefore spreads the selection pressure more evenly between the modes of operation required by the problem. Previous work using lexicase selection has shown that it works well on a simple modal symbolic regression problem [7] and the 2-bit digital multiplier problem [9].

In the present paper, along with presenting much more extensive tests and analysis, we further argue that lexicase selection should be preferred for uncompromising problems, whether or not they are modal. To see why, consider an individual that performs extremely well on some of the test cases, but has very high error on a few others. Selection methods such as tournament selection that compute a fitness by reducing the error vector to a single value (e.g. by summation) will rarely if ever select this individual if other individuals exist with mediocre error on all test cases. For uncompromising problems, however, elite performance on any test case may be important and worth propagating to the next generation even if the individual in question performs relatively poorly on many or even all other test cases. That is, we may often be more interested in selecting an individual with some great and some bad errors than an individual with all mediocre errors. Lexicase selection will select an unbalanced individual when the test cases at which it excels come near the beginning of the randomly ordered test case list. This may help to drive evolution toward solutions to uncompromising problems.

The theoretical worst-case time complexity of the lexicase selection algorithm for selecting parents each generation is $O(P^2T)$, where P is the population size and T is the number of test cases. In comparison, traditional tournament selection must sum the errors from every test case for every individual, giving a time complexity of $O(PT)$. While lexicase selection is theoretically slower in the worst case, in practice it often quickly eliminates many candidates and does not need to loop over every test case, running much faster than the worst-case analysis suggests. Additionally, if lexicase selection allows us to more often solve problems than other selection methods, it may be preferred even if it runs slower than those methods. We examine the wall-clock times of our experimental runs in Section V-B.

III. Related Work

To some extent one can consider multiple test cases to be the multiple objectives in a multiobjective optimization problem [10]. The match is not perfect, however, because objectives are goals that we want to achieve while test cases are tools for measuring how well we meet our objectives. Nonetheless, many of the techniques that have been developed to cope with multiple objectives can also be applied to the problem of coping with multiple test cases.

As far as we are aware, Langdon's work on evolving data structures is the only work that has used any type of Pareto-aware selection where the test cases are used as the objectives of the Pareto selection. In [11], Pareto tournaments are used for selection in evolving queues. This problem uses six objectives, five of which are based on the performance of the individual, and the last of which is used to minimize memory use. Similarly, [12] uses Pareto tournaments in evolving a list data structure. This problem uses 21 normal test cases, and two other objectives of memory and time. The Pareto tournaments in these papers are modeled after those proposed in [13].

To our knowledge, modern multiobjective approaches such as NSGA-II [14] and SPEA2 [15] have not been applied to genetic programming problems where the test cases are treated as objectives. These algorithms make assumptions about the objectives that don't usually hold for genetic programming test cases. Multiobjective algorithms are typically used on problems with very few objectives; often two objectives are used, and rarely more than four or five. Genetic programming problems frequently have many more test cases than this, sometimes ranging from 50 to 100 or even more. With this many objectives, Pareto-based algorithms may have trouble, since most individuals will not dominate each other leading to little performance information on which to base selection [15], [16]. This "curse of dimensionality" must be overcome to apply multiobjective algorithms to genetic programming problems.

Besides multiobjective methods, other efforts have been made to create parent selection techniques that give different weights to different test cases during selection. Fitness sharing [17] decreases selection pressure for individuals that are similar to other individuals in the population. Each individual's fitness is penalized based on how many individuals are within a specified distance, with closer individuals giving more penalty. This requires the user to specify a distance metric between individuals; in genetic programming, researchers have used both a structural distance of programs themselves and a behavioral distance based on the outputs of the programs. Fitness sharing using behavioral distance requires each individual to be compared with each other individual in the population, giving a time complexity of $O(P^2T)$ for population size P and T test cases. Undesirably, fitness sharing requires the user to set three sensitive parameters that can significantly affect its performance [18].

Implicit fitness sharing, first described in [19] and adapted for genetic programming in [2], is a diversity preservation technique that distributes reward among the individuals that solve a test case, giving more reward for cases solved by fewer

individuals. In this way it is similar to fitness sharing, without requiring the calculation of distances between individuals. It is typically only applied to problems with binary test cases, where an individual either solves a test case or does not. Like fitness sharing, implicit fitness sharing produces scaled fitnesses, with a tournament then used to select parents. The implicit fitness sharing fitness function is defined as

$$f_{IFS}(i) = \sum_{t \in T_i} \frac{1}{n(t)} \quad (1)$$

where $T_i \subseteq T$ is the set of test cases solved by individual i, and $n(t)$ is the number of individuals in the population that solve test case t. Note that fitness is to be maximized in implicit fitness sharing. Since the number of individuals solving each test case is computed only once per generation, implicit fitness sharing has a time complexity of $O(PT)$, similar to traditional tournament selection.

Implicit fitness sharing has been adapted for non-binary test cases in [20]. Here, the raw fitness $f(i,t)$ of individual i on test case t falls in the range $[0,1]$ with 0 being worst and 1 being best. Implicit fitness sharing is then redefined as

$$f_{NBIFS}(i) = \sum_{t \in T} \frac{f(i,t)}{\sum_{i' \in P} f(i',t)} \quad (2)$$

This non-binary implicit fitness sharing still scales fitnesses based on fitnesses of the rest of the population, and even reduces to traditional implicit fitness sharing when fitnesses are binary. The time complexity is still $O(PT)$.

The "historically assessed hardness" technique uses a different generalization of implicit fitness sharing for non-binary test cases, where fitness on each test case is scaled by the success rate of the population [21].

"Co-solvability" fitness extends implicit fitness sharing to consider pairs of test cases instead of single test cases [22]. Similarly to lexicase selection, this method emphasizes solving subsets of the test cases. For each pair of test cases, reward is given to each individual that solves both test cases, with the reward being higher for pairs of cases not solved by many individuals. The co-solvability fitness function is defined as

$$f_{CS}(i) = \sum_{t_j, t_k \in T_i : j<k} \frac{1}{n(t_j, t_k)}$$

where $T_i \subseteq T$ is the set of test cases solved by individual i, and $n(t_j, t_k)$ is the number of individuals that solve both case t_j and case t_k. Although this enhancement to implicit fitness sharing shares some motivations with lexicase selection, it only considers pairs of test cases, whereas lexicase selection considers prioritized lists of all test cases. This method has only been described for binary test cases, and it does not have an obvious generalization for non-binary test cases. Calculating co-solvability fitness requires each pair of test cases to be considered for each member of the population, giving a time complexity of $O(PT^2)$.

IV. Methods

In order to test the utility of lexicase selection on uncompromising problems, we used it in experiments with two different genetic programming systems and on four different problems. Here we present the GP systems, problems, and methods used in our experiments.

In our experiments, we compare lexicase selection to standard tournament parent selection. In tournament selection with tournament size n, n individuals are randomly selected from the population, and the individual with the lowest total error is selected to be the parent. We also present results using implicit fitness sharing (IFS), which scales fitnesses and then performs standard tournaments using that scaled fitness[4]. For the finite algebras problem, which has binary errors, we use the standard IFS fitness function given in Eq. (1). For the factorial problem and the wc problem, which have non-binary errors, we use the modified non-binary IFS fitness function given in Eq. (2). For both tournament selection and IFS, we present data for a variety of tournament sizes that span the range commonly used in the literature. The specific set of tournament sizes is not identical for each experiment, although it always includes low, medium and high values; we sampled the range of tournament sizes more sparsely for experiments that required greater computational resources. Note also that *within* each experiment—that is, for each problem—the same set of tournament sizes is used across all conditions.

A. Genetic Programming Systems

We use two different genetic programming systems in this paper: a tree-based genetic programming system and PushGP. For the finite algebras problem we use the tree-based system, which provides a natural representation for the problem, since it searches for a function composed of nested calls to a single algebraic operator. For the other three problems we use PushGP since it allows more expressive programs to be evolved than can easily be represented in most tree-based genetic programming systems. The digital multiplier and wc problems require evolved programs to return multiple outputs, which is not easy to do in the functional representations of most tree-based systems. Additionally, the factorial and wc problems require a large range of expressive instructions not easily implemented in tree-based systems, such as those that allow for iteration and multiple data types. We believe the use of different genetic programming systems depending on the requirements of the problem is not only reasonable, but shows that our conclusions are not limited to a single representation.

The standard tree-based genetic programming system used here is designed in the style of Koza's 1992 system [1]. Programs are Lisp-style symbolic expressions in prefix notation, represented internally as trees. Random trees are generated using the ramped half-and-half algorithm [1]. We use standard tree-replacement mutation and crossover, without any biases in the selection of nodes to be mutated or used in crossover.

The second genetic programming system, PushGP, evolves programs expressed in the Push programming language. Push is a stack-based language that was designed specifically for use in genetic programming systems as the language in which evolving programs are expressed [23], [24], [25]. It is a

[4]Time and resource constraints did not permit the inclusion of IFS results for the digital multiplier problem, which is computationally expensive.

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.

The Push interpreter uses a separate stack 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.

In the PushGP runs in this paper we only use the genetic operator ULTRA, which stands for "Uniform Linear Transformation with Repair and Alternation" [26], [9][5]. ULTRA creates a child from two parent programs by treating them as linear sequences and traversing them in parallel while copying program elements from one parent or the other to the child. The ULTRA "alternation rate" parameter specifies the probability of alternating between source parents during the traversal, while the "alignment deviation" parameter specifies the standard deviation of random index offsets that may occur during alternation. The ULTRA "mutation rate" parameter specifies the probability of each element being mutated (replaced with another random element) during a subsequent traversal of the child program.

Push and PushGP implementations exist in C++, Java, JavaScript, Python, Common Lisp, Clojure, and Scheme, among other languages. Many of these are available for free download from the Push project page.[6] The results presented below were obtained using Clojush, the Clojure implementation of PushGP.[7]

TABLE I
PARAMETERS FOR OUR EXPERIMENTS. FA IS FOR THE FINITE ALGEBRAS PROBLEM, DM IS FOR THE DIGITAL MULTIPLIER PROBLEM, FACT IS FOR THE FACTORIAL SYMBOLIC REGRESSION PROBLEM, AND WC IS FOR THE TRADITIONAL PROGRAMMING WORD COUNT PROBLEM. IN THE SYSTEM ROW, TREE IS THE TREE-BASED GENETIC PROGRAMMING SYSTEM DESCRIBED IN SECTION IV-A, AND PUSH IS THE PUSHGP SYSTEM.

Problem	**FA**	**DM**	**Fact**	**wc**
System	Tree	Push	Push	Push
Runs Per Condition	100	100	100	200
Number of Test Cases	27	64	10	242
Population Size	1000	5000	1000	1000
Max Generations	1000	4000	500	300
Max Program Size	1000	1000	500	1000
Max Initial Program Size	-	400	100	400
Expected Initial Program Size	50	-	-	-
Max Initial Program Depth	20	-	-	-
Expected Mutation Code Size	10	-	-	-
Max Mutation Code Depth	10	-	-	-
Max Instructions Executed	-	1000	1000	2000
Crossover Probability	50%	0%	0%	0%
Mutation Probability	50%	0%	0%	0%
ULTRA Probability	0%	100%	100%	100%
ULTRA Mutation Rate	-	0.01	0.05	0.01
ULTRA Alternation Rate	-	0.01	0.05	0.01
ULTRA Alignment Deviation	-	10	10	10

TABLE II
ALGEBRAIC OPERATORS DEFINING THE FINITE ALGEBRAS IN THIS PAPER.

$\mathbf{A}_1$ $*$	0	1	2
0	2	1	2
1	1	0	0
2	0	0	1

$\mathbf{A}_2$ $*$	0	1	2
0	2	0	2
1	1	0	2
2	1	2	1

B. Problems

We used the tree genetic programming and PushGP systems to test lexicase selection on four problems: a problem in pure mathematics related to finite algebras, a boolean digital multiplier problem, a factorial symbolic regression problem, and a traditional programming problem called wc (for "word count"). The genetic programming parameters that we used in our experiments are presented in Table I.[8] We varied some parameters, such as population size, maximum generations, and maximum program sizes, for different problems based partially on exploratory runs for the problems and partially on our prior experiments with the problems; none were particularly optimized.

Previous work in genetic programming for finite algebras has created human-competitive results (and won a "Humies" Gold Prize) [27]. Here, we borrow a problem from that work to use as a benchmark. This problem, which we will simply call the **finite algebras problem**, is to find a discriminator term in a three-element, single-operator algebra. A *discriminator term* [28] is a ternary function $t(x, y, z)$ satisfying

$$t(x,y,z) = \begin{cases} x \text{ if } x \neq y \\ z \text{ if } x = y \end{cases}$$

The algebras presented here only have one operator $*$, which is therefore the only function in our function set. Since we are evolving ternary terms, we use the three terminals x, y, and z. Each test case for this problem uses values for x, y, and z chosen from the set $\{0, 1, 2\}$, giving 27 test cases. If an evolved program correctly solves a particular case it gets an error of 0 for that case, and otherwise gets an error of 1.

To test the differences between lexicase, tournament, and implicit fitness sharing (IFS) parent selection methods on this problem, we used the tree genetic programming system described in Section IV-A to search for discriminator terms for

[5]The ULTRA operator used here is essentially that described in [9]. The version described in [26] and also used here was intended to pad the shorter program with no-op instructions that would later be removed, but in fact the padding rarely occurred. See erratum notice: http://hampshire.edu/lspector/pubs/spector-gptp-2013-preprint-erratum.pdf

[6]http://pushlanguage.org

[7]https://github.com/lspector/Clojush

[8]We conducted our runs over a period of time during which we made minor changes to our implementation of ULTRA. For all wc problem runs and for the lexicase and size 7 tournament digital multiplier problem runs, empty pairs of parentheses were removed at the end of ULTRA. In addition, for the wc problem runs parentheses could be added or removed only by ULTRA's crossover and repair steps, but not by its mutation step. We have found no evidence that either of these changes has any meaningful impact on the PushGP algorithm for the problems studied here.

TABLE III
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, $invertFirstThenAnd$, dup, $swap$, rot
Input/Output	$in1$, ..., $in2n$, $out1$, ..., $out2n$

the finite algebras $\mathbf{A}_1$ and $\mathbf{A}_2$ given in Table II. We explore the parameter space of tournament sizes by using sizes between 2 and 10. The finite algebras problem is uncompromising, in that we are only interested in solutions with zero error. Runs that do not find perfect solutions do not tell us anything about the finite algebras themselves.

The **digital multiplier problem** requires the system to create a program representing a digital circuit that multiplies two binary numbers. An n-bit digital multiplier circuit takes two n-bit numbers represented in binary by booleans as input and multiplies them together to create a $2n$-bit number as output. 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 [29], [3]. 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. Previous work has shown PushGP's ability to evolve 2-bit digital multipliers [9]. Here, we use PushGP to evolve solutions to the more difficult 3-bit digital multiplier problem.

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 will be an integer between 0 and $2n$. In our implementation of the problem in PushGP, we provide one input instruction for each input bit, and one output instruction for each output bit. 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 III. The first four of these are the instructions recommended by Walker and Miller [30], each popping the top two items on the boolean stack and pushing the result onto the boolean stack. The other three are typical stack manipulation instructions that are often used in Push. The $booleanDup$ instruction duplicates the top item on the boolean stack, the $booleanSwap$ instruction swaps the top two items on the boolean stack, and $booleanRot$ 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 for random code, 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. The digital multiplier problem is uncompromising, since programs that do not achieve zero error on every test case are not of interest.

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. Our version uses 10 input test cases, ranging from 1 to 10 (with outputs ranging from $1! = 1$ to $10! = 3628800$).

When using normal summed tournament selection some test cases will have a much larger impact on an individual's fitness than others, since the larger test cases will likely have much larger error magnitude than the smaller test cases. To try to make the influence of test cases more even, we also tested tournament selection using normalized error values in the range $[0, 1]$. Additionally, we used normalization when using implicit fitness sharing, since it requires an error in $[0, 1]$.

We tested three different methods of normalization, each of which returns 0 if the program returns the target output. The first takes the raw error value $e_t = |c_t - y_t|$ on each test t, where c_t is the correct output and y_t is the output of the program, and normalizes it to

$$e_{t1} = 1 - \frac{1}{e_t + 1} = \frac{e_t}{e_t + 1} \tag{3}$$

This method is very similar to Koza's adjusted fitness [1], although adjusted fitness was used to scale the total error of an individual for the use with fitness proportionate selection. The second method, suggested for use with the factorial problem in [31], normalizes the error as $e_{t2} = \frac{e_t}{|c_t|+|y_t|}$. Although this method creates a flatter normalization function with respect to y_t, it behaves oddly in that it is not symmetrical around $y_t = c_t$. In fact, if c_t is positive and y_t is negative, the maximum normalized error of 1.0 will be given even if y_t and c_t have a relatively small absolute distance, where a smaller normalized error will be given for an output of y_t that is positive but with a much larger absolute distance from c_t. In order to obtain the gentler normalization provided by e_{t2}, but without the odd behavior near zero, we created a third method that behaves exactly like e_{t2} when $c_t \leq y_t$, but is symmetrical when $c_t > y_t$. This uses the normalization function $e_{t3} = \frac{e_t}{e_t+2|c_t|+1}$, where the addition of 1 in the denominator avoids issues when e_t and c_t are zero.

Note that the relative magnitude of error values between test cases does not affect lexicase selection; it only takes into account differences within a test case. Therefore we did not need to test lexicase selection using normalization of error values. This nice side-effect of using lexicase selection means a researcher need not consider differences in magnitudes of error values or options for normalization across test cases.

For this problem we used a Push instruction 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 (see [25]).

The **wc problem** is a recently proposed traditional programming benchmark problem for genetic programming [32]. The objective of the wc problem is to find a program that takes as input a file and outputs the number of characters, words, and newlines in the file. Such a program mirrors the functionality of the Unix word count utility program wc. This problem requires programming concepts used frequently by human programmers but rarely by genetic programming, including the use of multiple data types, multiple outputs, and control flow manipulation. We use the PushGP implementation of the wc problem found in [32], and supplement the runs presented there with runs using additional tournament sizes and IFS.

With the wc problem, we wish to find programs that perfectly count the numbers of characters, words, and newlines in a file. This problem is uncompromising in that a program that mostly passes the test cases does not constitute a useful word count program. To keep the problem tractable, we only require that the program work on files that contain at most 100 characters. Since we cannot use every single file containing at most 100 characters as a test case input, we follow [32] and use a combination of random and prescribed inputs for each run. In each run, 200 of the inputs are random strings of length at most 100; 20 of the inputs are random strings of length at most 100 that are guaranteed to end in a newline, and 22 of the inputs are prescribed strings that cover edge cases, such as very short and very long files. For a program to count as a success, it must also achieve zero error on a withheld set of test data to show that it generalizes to unseen inputs. This withheld test set contains 500 random strings and 50 random strings ending in newlines, all at most 100 characters long.

Each test case has three expected outputs, for the counts of the characters, words, and newlines in the input file. We therefore assign three error values per test case, with each error being the absolute difference between the expected output and the program's output. If a program does not return a particular output, it receives a penalty error of $100,000$.

Like many traditional programming problems, the wc problem requires a wide and varied set of instructions. We use an instruction set with 71 instructions and ERCs; the specific instructions are exactly the same as the ones recommended in [32]. These instructions include many standard Push stack manipulation instructions for the string, integer, boolean, and exec stacks. As with the factorial problem, the exec stack instructions here allow for a variety of control flow constructs, including looping, conditionals, and recursion. Additionally, two instructions allow for character and line input, and three output instructions store integers to use as outputs for number of characters, words, or newlines. The instruction set also supports tagging of code, integers, and strings, allowing programs to store values and modules that can be retrieved later [33]. Finally, three ERCs allow random code to create integer literals, single-character whitespace string literals, and single-character non-whitespace string literals.

For this problem we compare lexicase selection to tournaments with sizes 3, 5, and 7, as well as IFS with the same tournament sizes. Since the wc problem returns non-binary errors, we use the non-binary IFS formula from Eq. (2). Since this formula requires fitnesses in the range $[0, 1]$, we normalize errors by dividing them by $100,000$, which is the penalty error given when no output is produced by a program, and for IFS is also the largest error a program can receive if it does return an output.

C. Performance Measures

Since we are primarily interested in the extent to which selection algorithms help to evolve programs with zero error on every test case, we focus our results on measures of these successes, in particular the success rate (i.e. the percent of runs that find programs with zero error). We test the statistical significance and reliability of the difference in success rates found using different selection methods by using Fisher's exact test (with significance level of 0.05) and confidence intervals of the difference in success rates.

We use the same population size and maximum number of generations for each parent selection algorithm, resulting in the same maximum number of program evaluations. While this does not fully ensure that every run will consume the same computational resources, it is a commonly used approximation. We additionally present the wall-clock time consumed by each set of runs, which can provide information about differences in computational resources that stem from processes other than program evaluation, including parent selection.

In order to measure the time each algorithm used in our experiments, we recorded the wall-clock times used by each run as well as the number of generations executed, to calculate the time used per generation. Most of our runs were performed on a cluster of machines with varying computational resources, although the finite algebras runs were performed on a 2012 MacBook Pro. Since we cannot prescribe which runs are performed on which machines in the cluster, some sets of runs may have randomly run on faster machines than others. Additionally, since we share the cluster with other users, we cannot guarantee a consistent load; some runs may have shared a machine with other processes where others did not. Finally, while we did our best to implement efficient algorithms, none of them were specifically optimized, and faster implementations may be possible. Taking all of this into consideration, small differences in observed wall-clock times do not necessarily indicate significant differences in run times; we are only really interested in orders of magnitude distinctions.

V. Experimental Results

Here we report the results of our experiments, including differences in performance, execution times, and population diversity.

TABLE IV
RESULTS ON THE FINITE ALGEBRAS PROBLEM USING THE ALGEBRA $\mathbf{A}_1$ WITH 100 RUNS IN EACH CONDITION. IFS GIVES RESULTS USING IMPLICIT FITNESS SHARING PARENT SELECTION. THE SUCCESS RATE OF EACH SET OF RUNS USING TOURNAMENT SELECTION OR IMPLICIT FITNESS SHARING IS COMPARED WITH THE SUCCESS RATE USING LEXICASE SELECTION; WE PRESENT THAT DIFFERENCE AND A 95% CONFIDENCE INTERVAL OF THAT DIFFERENCE.

Parent Selection Method	Tournament Size	Success Rate	Difference in Success Rate with Lexicase	95% Confidence Interval of Difference in Success Rate
Lexicase	-	0.99	-	-
Tournament	2	0.01	0.98	[0.923, 0.999]
Tournament	3	0.01	0.98	[0.923, 0.999]
Tournament	4	0.05	0.94	[0.869, 0.975]
Tournament	5	0.02	0.97	[0.909, 0.993]
Tournament	6	0.04	0.95	[0.882, 0.981]
Tournament	7	0.03	0.96	[0.895, 0.987]
Tournament	8	0.06	0.93	[0.856, 0.968]
Tournament	9	0.07	0.92	[0.843, 0.961]
Tournament	10	0.04	0.95	[0.882, 0.981]
IFS	2	0.13	0.86	[0.771, 0.915]
IFS	3	0.43	0.56	[0.449, 0.649]
IFS	4	0.58	0.41	[0.302, 0.501]
IFS	5	0.55	0.44	[0.331, 0.532]
IFS	6	0.64	0.35	[0.246, 0.440]
IFS	7	0.57	0.42	[0.312, 0.512]
IFS	8	0.64	0.35	[0.246, 0.440]
IFS	9	0.71	0.28	[0.182, 0.367]
IFS	10	0.73	0.26	[0.164, 0.346]

TABLE V
RESULTS ON THE FINITE ALGEBRAS PROBLEM USING THE ALGEBRA $\mathbf{A}_2$ WITH 100 RUNS IN EACH CONDITION. IFS GIVES RESULTS USING IMPLICIT FITNESS SHARING PARENT SELECTION. THE SUCCESS RATE OF EACH SET OF RUNS USING TOURNAMENT SELECTION OR IMPLICIT FITNESS SHARING IS COMPARED WITH THE SUCCESS RATE USING LEXICASE SELECTION; WE PRESENT THAT DIFFERENCE AND A 95% CONFIDENCE INTERVAL OF THAT DIFFERENCE.

Parent Selection Method	Tournament Size	Success Rate	Difference in Success Rate with Lexicase	95% Confidence Interval of Difference in Success Rate
Lexicase	-	1.0	-	-
Tournament	2	0	1.0	[0.953, 1.0]
Tournament	3	0.06	0.94	[0.869, 0.974]
Tournament	4	0.12	0.88	[0.795, 0.930]
Tournament	5	0.14	0.86	[0.772, 0.914]
Tournament	6	0.16	0.84	[0.749, 0.898]
Tournament	7	0.17	0.83	[0.737, 0.890]
Tournament	8	0.10	0.90	[0.819, 0.946]
Tournament	9	0.26	0.74	[0.638, 0.813]
Tournament	10	0.18	0.82	[0.726, 0.882]
IFS	2	0.28	0.72	[0.616, 0.795]
IFS	3	0.61	0.39	[0.286, 0.479]
IFS	4	0.74	0.26	[0.167, 0.343]
IFS	5	0.83	0.17	[0.090, 0.243]
IFS	6	0.84	0.16	[0.082, 0.232]
IFS	7	0.83	0.17	[0.090, 0.243]
IFS	8	0.88	0.12	[0.050, 0.185]
IFS	9	0.79	0.21	[0.124, 0.288]
IFS	10	0.72	0.28	[0.185, 0.364]

A. Performance

Table IV presents the results of our runs on the finite algebras problem using algebra $\mathbf{A}_1$. The runs using lexicase selection found solutions in almost every run, and outperformed tournament selection and IFS. The best non-lexicase run used IFS with tournament size of 10. Comparing lexicase selection with any non-lexicase method, Fisher's exact test gives a p-value less than 0.0001, so we can reject the null hypothesis at the 0.05 significance level that there is no association between selection method and the number of successes. Most of the tournament and IFS runs show substantial differences in success rate when compared with the lexicase selection runs, and even the best IFS conditions are significantly worse than lexicase selection.

The results of our runs on the finite algebras problem using algebra $\mathbf{A}_2$ are presented in Table V. The runs using lexicase selection found solutions in every run, with the success rates for tournament selection and IFS also being higher than for algebra $\mathbf{A}_1$. For this algebra, IFS with tournament size 8 had the highest success rate; Fisher's exact test for lexicase and size 8 IFS gives a p-value of 0.0003, so we can reject the null hypothesis at the 0.05 significance level. Since size 8 IFS performed best out of the tournament and IFS runs, this rejection of the null hypothesis holds when comparing any of these runs to lexicase selection. The differences in success rate and their 95% confidence intervals indicate that lexicase selection performed vastly better than tournament selection and moderately better than even the best of IFS setting.

TABLE VI
RESULTS ON THE 3-BIT DIGITAL MULTIPLIER PROBLEM WITH 100 RUNS IN EACH CONDITION. THE SUCCESS RATE OF EACH SET OF RUNS USING TOURNAMENT SELECTION IS COMPARED WITH THE SUCCESS RATE USING LEXICASE SELECTION; WE PRESENT THAT DIFFERENCE AND A 95% CONFIDENCE INTERVAL OF THAT DIFFERENCE.

Parent Selection Method	Tournament Size	Success Rate	Difference in Success Rate with Lexicase	95% Confidence Interval of Difference in Success Rate
Lexicase	-	1.0	-	-
Tournament	2	0	1.0	[0.953, 1.0]
Tournament	4	0	1.0	[0.953, 1.0]
Tournament	6	0	1.0	[0.953, 1.0]
Tournament	7	0	1.0	[0.953, 1.0]
Tournament	8	0	1.0	[0.953, 1.0]

Table VI presents our results on the 3-bit digital multiplier problem. We compare lexicase selection with tournament selection using various tournament sizes. On this problem, PushGP with lexicase selection found successful programs in every run, whereas tournament selection never found a solution. A comparison of lexicase selection with each of the tournament selection conditions using Fisher's exact test gives a p-value less than 0.0001, meaning we can reject the null hypothesis of no association between selection method and number of successes at the 0.05 significance level. The

TABLE VII
RESULTS ON THE FACTORIAL SYMBOLIC REGRESSION PROBLEM WITH 100 RUNS IN EACH CONDITION. NORMALIZED USES NORMALIZED ERRORS WITH NORMAL TOURNAMENT SELECTION; IFS GIVES RESULTS USING IMPLICIT FITNESS SHARING PARENT SELECTION. BOTH NORMALIZED AND IFS USE THE NORMALIZATION FUNCTION e_{t1} GIVEN IN EQ. (3). THE SUCCESS RATE OF EACH SET OF RUNS USING TOURNAMENT SELECTION OR IMPLICIT FITNESS SHARING IS COMPARED WITH THE SUCCESS RATE USING LEXICASE SELECTION; WE PRESENT THAT DIFFERENCE AND A 95% CONFIDENCE INTERVAL OF THAT DIFFERENCE.

Parent Selection Method	Tournament Size	Success Rate	Difference in Success Rate with Lexicase	95% Confidence Interval of Difference in Success Rate
Lexicase	-	0.51	-	-
Tournament	2	0	0.51	[0.401, 0.599]
Tournament	4	0	0.51	[0.401, 0.599]
Tournament	6	0	0.51	[0.401, 0.599]
Tournament	8	0	0.51	[0.401, 0.599]
Normalized	2	0	0.51	[0.401, 0.599]
Normalized	4	0	0.51	[0.401, 0.599]
Normalized	6	0	0.51	[0.401, 0.599]
Normalized	8	0.01	0.50	[0.390, 0.591]
IFS	2	0	0.51	[0.401, 0.599]
IFS	4	0	0.51	[0.401, 0.599]
IFS	6	0	0.51	[0.401, 0.599]
IFS	8	0	0.51	[0.401, 0.599]

differences in success rate between lexicase selection and each of the tournament sizes is obviously large. The results using lexicase selection are much better than in [9], in which no perfect solutions were found for the 3-bit digital multiplier problem. This difference is likely because of the larger population size and maximum generations used in this paper.

Table VII presents the results of our runs on the factorial problem. We tested each of the three normalization methods, both with and without IFS, using tournament sizes of 2, 4, 6, and 8. We only present normalized tournaments and IFS using the normalization function e_{t1} given in Eq. (3), since neither of the other normalization methods resulted in more than one successful program across all sets of runs using them. Lexicase selection found successful programs in just over half its runs, whereas none of the other selection methods found more than one successful program. Comparing lexicase selection with any of the tournament, normalized, and IFS sets of runs, Fisher's exact test gives a p-value less than 0.0001, so we can reject the null hypothesis at the 0.05 significance level that there is no association between selection method and the number of successes. The differences in success rate between the lexicase selection runs and all other runs are large, near 0.5.

The wc problem proved more difficult for lexicase selection than any of the other problems presented here; results are in Table VIII. Lexicase selection found 11 successful programs that achieved zero error on both the training and withheld test sets, whereas the other methods found none. When comparing the lexicase runs to the tournament and IFS runs, Fisher's exact test gives a p-value of 0.001, indicating we can reject the null hypothesis at the 0.05 significance level that there is no

TABLE VIII
RESULTS ON THE WC PROBLEM WITH 200 RUNS IN EACH CONDITION. IFS GIVES RESULTS USING IMPLICIT FITNESS SHARING PARENT SELECTION. THE SUCCESS RATE OF EACH SET OF RUNS USING TOURNAMENT SELECTION OR IMPLICIT FITNESS SHARING IS COMPARED WITH THE SUCCESS RATE USING LEXICASE SELECTION; WE PRESENT THAT DIFFERENCE AND A 95% CONFIDENCE INTERVAL OF THAT DIFFERENCE.

Parent Selection Method	Tournament Size	Success Rate	Difference in Success Rate with Lexicase	95% Confidence Interval of Difference in Success Rate
Lexicase	-	0.055	-	-
Tournament	3	0	0.055	[0.020, 0.088]
Tournament	5	0	0.055	[0.020, 0.088]
Tournament	7	0	0.055	[0.020, 0.088]
IFS	3	0	0.055	[0.020, 0.088]
IFS	5	0	0.055	[0.020, 0.088]
IFS	7	0	0.055	[0.020, 0.088]

association between parent selection method and the success rate. The differences in success rate and the 95% confidence intervals of those differences show that the effect of lexicase selection on success rate is likely small but nonetheless meaningful, particularly since it appears to be difference between "no successes" and "occasional successes."

B. Execution Times

We have shown that lexicase selection allows genetic programming to find many more solutions than traditional tournament selection or IFS on a variety of uncompromising problems. Since the time complexity of lexicase selection is worse than tournament selection and IFS, we expect it to be slower, but not prohibitively so. Here we present the time used by our runs to produce our results.

For each parent selection method, the mean time per generation did not vary much between different tournament sizes. For each problem, Table IX presents the mean wall-clock time per generation of the sets of runs that took the least and most time for each selection method. Since we only performed one set of runs for lexicase selection on each problem, only one time is presented for both minimum and maximum. For example, our one set of runs using lexicase selection on the $\mathbf{A}_1$ finite algebras problem took 2.6 seconds per generation on average, whereas the set of tournament selection runs using least time took 1.2 seconds (using tournament size 2), and the most time took 1.4 seconds (using tournament size 5).

For the finite algebras problem, runs using lexicase selection took about twice as long per generation as those using tournament selection or IFS. For the 3-bit digital multiplier problem, runs using lexicase selection were significantly slower than those using tournament selection, about 7 times slower than the longest tournament selection runs. This large increase in time can be attributed to the large population size of 5000 we used for this problem, whereas we used a population size of 1000 for all other problems. Since lexicase selection's time complexity is quadratic in the population size, we would expect it to be comparatively slower for the larger population size we used.

TABLE IX
WALL-CLOCK TIMES FOR THE SHORTEST AND LONGEST MEAN TIME PER GENERATION ACROSS SETS OF RUNS USING THE SAME PARENT SELECTION METHOD AND DIFFERENT TOURNAMENT SIZES. DATA COMES FROM THE SAME RUNS AS THOSE PRESENTED IN PREVIOUS RESULTS TABLES: $\mathbf{A}_1$ AND $\mathbf{A}_2$ ARE FOR THE FINITE ALGEBRAS PROBLEM USING ALGEBRAS $\mathbf{A}_1$ AND $\mathbf{A}_2$ RESPECTIVELY, DM IS FOR THE DIGITAL MULTIPLIER PROBLEM, FACT IS FOR THE FACTORIAL PROBLEM, AND WC IS FOR THE WC PROBLEM.

Problem	Parent Selection Method	Minimum mean time per generation (seconds)	Maximum mean time per generation (seconds)
$\mathbf{A}_1$	Lexicase	2.6	2.6
	Tournament	1.2	1.4
	IFS	1.2	1.3
$\mathbf{A}_2$	Lexicase	2.5	2.5
	Tournament	1.2	1.4
	IFS	1.0	1.2
DM	Lexicase	464	464
	Tournament	25	71
Fact	Lexicase	11.9	11.9
	Tournament	5.4	6.7
	Normalized	0.4	0.6
	IFS	3.0	4.7
wc	Lexicase	394	394
	Tournament	142	295
	IFS	136	229

For the factorial problem, lexicase selection is again the slowest, taking about twice as long as tournament selection and three times as long as IFS. The runs using tournament selection on normalized errors ran surprisingly quickly; we discovered that this is likely because many of the runs resulted in populations of programs that contained a single instruction that had no effect. This strange result can be explained by the fact that with this normalization scheme, most random programs do not achieve better error than a program that does nothing and simply returns the input. Some, but not all, of our runs using other normalization techniques both with and without IFS had this problem, though even the ones with longer execution times did not find more than 1 successful program.

The mean generation lengths on the wc problem are at most about 3 times longer when using lexicase selection than when using tournament or IFS selections. For all methods times are longer than for the other problems; this can be explained by a combination of many more test cases and a larger maximum number of instructions executed per execution.

All of our sets of runs using lexicase selection had longer wall-clock times per generation than tournament selection or IFS. But, for many of these problems the difference between using lexicase selection and using tournament selection or IFS is the difference between finding successful programs and not. Additionally, many of the lexicase runs actually finished faster than the other runs, since they found successful programs earlier in the runs. Since the difference in wall-clock time is not many orders of magnitude slower for any problem, we think the benefits of using lexicase selection far outweigh the extra cost in time.

C. Population Diversity

The results presented here raise the question of why lexicase selection performs significantly better on uncompromising problems than tournament selection or IFS. One hypothesis is that the way lexicase selection emphasizes the selection of individuals that are extremely good on at least a few test cases but possibly not great on others allows runs using lexicase selection to maintain higher levels of population diversity than techniques that reduce fitnesses to a single value. Although maintaining higher levels of diversity may be helpful, it is also necessary to provide sufficient selection pressure to exploit good programs in order to find better ones; simply maintaining a diverse set of individuals does not single-handedly help find a solution without pressure toward the goal. This tension between exploration and exploitation is well known in evolutionary algorithms. The fact that our experiments using lexicase selection found many more solutions than other methods suggests that it at least contributes sufficient exploitative pressure toward the goal; we now wish to investigate whether or not it also maintains higher levels of diversity than the other methods.

Various measures of diversity have been proposed in the genetic programming literature. Because we are primarily interested in how a genetic programming population explores the space of output vectors, we will focus on behavioral diversity [34]. Here Jackson defines the *behavior* of a program to be the vector of outputs it produces for the test case inputs. The *behavioral diversity* of a population of programs is the percent of distinct behavior vectors in the population. Jackson shows that there is correlation, if not causation, between higher levels of behavioral diversity and higher solution rates on a variety of small benchmark problems [34].

We will explore whether the use of lexicase selection leads to higher levels of behavioral diversity than tournament selection or IFS. IFS was designed to increase population diversity by reducing the influence of a test case on an individual's fitness proportionately with the number of individuals in the current population that solve it (or do well on it in the case of non-binary IFS). We will present behavioral diversity data from our finite algebras, factorial, and wc runs[9]. Since the behavioral diversity plots for tournament selection or IFS runs with different tournament sizes follow very similar trends, we will only present a few tournament size settings from each selection method to increase clarity.

Figures 2 and 3 plot the mean behavioral diversity at each generation for the finite algebras problem on the $\mathbf{A}_1$ and $\mathbf{A}_2$ algebras respectively. We plot runs that use lexicase selection, IFS with a range of tournament sizes, and size 9 tournament selection (which performed best out of the tournament sizes on these problems). After the first 10 or so generations, we see some separation between the plots. IFS with tournament size of 3 shows the highest diversity for the first 15 to 25 generations, after which point lexicase selection maintains the highest diversity until almost all of its runs

[9]We finished our digital multiplier runs before deciding to measure behavioral diversity, and could not redo those runs because they are too computationally expensive.

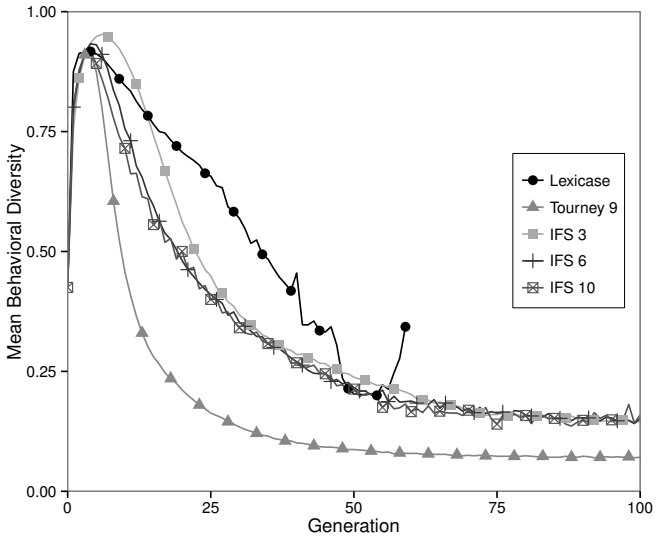

Fig. 2. Behavioral diversity for the $\mathbf{A}_1$ finite algebras problem. The numbers beside runs indicate the tournament size used. The plot of lexicase selection ends after every run has found a successful program; it becomes jagged before it ends because there it averages only the few runs that have not yet succeeded.

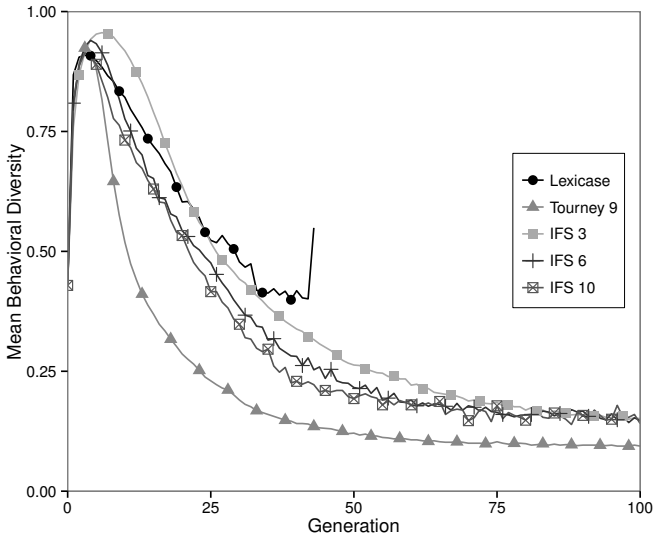

Fig. 3. Behavioral diversity for the $\mathbf{A}_2$ finite algebras problem. The numbers beside runs indicate the tournament size used. The plot of lexicase selection ends after every run has found a successful program; it becomes jagged before it ends because there it averages only the few runs that have not yet succeeded.

have found solutions. Interestingly, IFS with tournament size 3 performed worse than the IFS runs with larger tournament sizes presented in these figures. We believe this shows that IFS with tournament size 3 maintained a low level of selection pressure leading to higher diversity, but did not have enough pressure to guide evolution toward successful programs as readily. In each of these figures, size 9 tournament selection had the lowest behavioral diversity during all but the first few generations.

Figure 4 plots the mean behavioral diversity at each generation of our runs on the factorial problem. Unlike the plots for the finite algebras problem, behavioral diversity remains

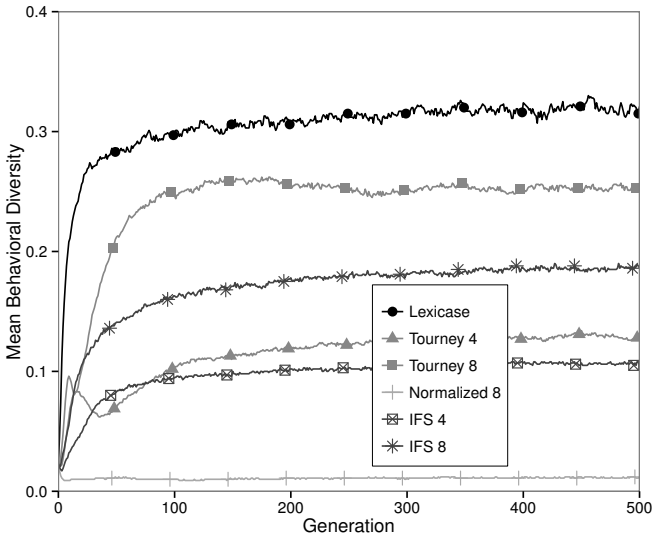

Fig. 4. Behavioral diversity for the factorial problem. The numbers beside runs indicate the tournament size used.

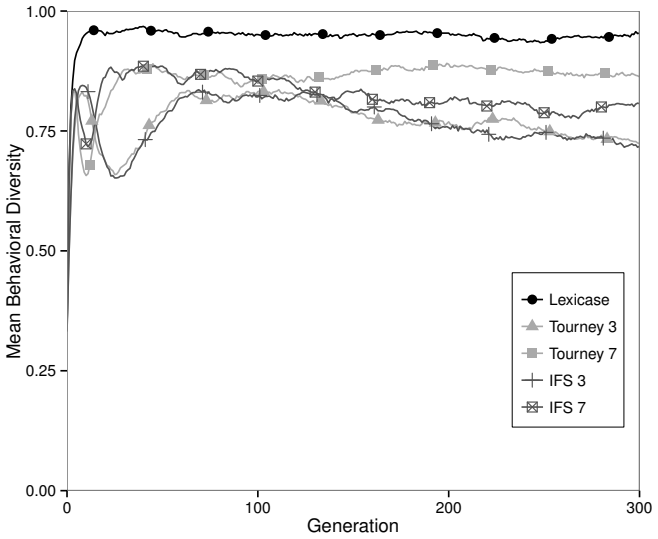

Fig. 5. Behavioral diversity for the wc problem. The numbers beside runs indicate the tournament size used.

relatively low throughout these runs, never reaching higher than 40%. Whereas the finite algebras runs were capped at 100 generations, these runs were allowed 500 generations; it appears that behavioral diversity did not change much in any set of runs after the first 100 generations. Lexicase selection maintained the highest diversity. Surprisingly, tournament selection maintained higher diversity than IFS. For both, runs with higher tournament sizes had higher diversity, which goes against our intuition that larger tournament sizes lead to higher selection pressure and less diversity. The runs using normalization had extremely low diversity, likely because many runs resulted in populations of essentially empty programs as discussed in Section V-B.

Figure 5 plots the mean behavioral diversity at each generation of our runs on the wc problem. All of the runs on the

wc problem maintained a relatively high behavioral diversity, with none falling below 0.65 after the first few generations. This may partially be attributed to the large length of the behavior vectors for this problem: the wc problem uses 242 test inputs, each of which produces 3 behaviors (one each for character, word, and newline outputs) for a total of 726 behaviors. Lexicase selection again maintained the highest behavioral diversity, not falling below 0.93 after the first 10 generations. Both tournament selection and IFS maintained moderate diversity with some swings up and down; for both, runs with higher tournament sizes had higher mean diversity throughout most of the generations.

The figures presented here show that runs using lexicase selection maintained higher levels of behavioral diversity than most runs using tournament selection or IFS. Only for some generations of the finite algebras problem did IFS with tournament size 3 have higher behavioral diversity; yet these runs found fewer successful programs than IFS with most other tournament sizes. We found it remarkable that while IFS was designed to maintain population diversity, in our experiments it almost never achieved higher levels of behavioral diversity than lexicase selection, and even had lower diversity than standard tournament selection on some problems. Even though lexicase selection was not designed for diversity maintenance, it consistently produced high behavioral diversity while also finding the most successful programs. This correlation suggests that lexicase selection's ability to maintain high levels of diversity while also applying strong selection pressure on random subsets of the test cases may be partially responsible for its success on the uncompromising problems presented here.

VI. Conclusion

The results presented above clearly demonstrate, using two different genetic programming systems and four different problems, that lexicase selection can perform well on at least some uncompromising problems—that is, on problems where solutions must perform optimally on each of many test cases, without compromising performance on any one test case for improved performance on others. As we have argued above, this is a broad class of problems that includes many problems to which genetic programming has traditionally been applied.

In tests of standard tree-based genetic programming on a finite algebras problem we saw that sets of runs using lexicase selection succeeded nearly 100% of the time whereas runs using tournament selection, with tournament sizes ranging from 2 to 10, succeeded between 0% and 26% of the time, depending on the specific finite algebra and the tournament size used. Runs using implicit fitness sharing did better than runs using ordinary tournaments, but nonetheless significantly worse than runs using lexicase selection.

In tests of PushGP on the 3-bit digital multiplier problem, the factorial regression problem, and the wc utility problem we also observed significant advantages from the use of lexicase selection. Indeed, the results of these experiments were even more dramatic, with lexicase selection producing many solutions (including a solution rate of 100% for the digital multiplier problem) even though these problems were almost completely unsolvable using other methods. Runs using lexicase selection were somewhat slower per generation than other methods, but this difference in execution times was minor compared to the benefits to problem solving performance. Experimental evidence suggests that lexicase selection allows genetic programming runs to maintain higher levels of population diversity, which we hypothesize contributes to the observed increases in performance.

One drawback of the form of lexicase selection used here is that it may perform poorly in contexts in which the "elite sets" for most test cases include only a single individual. In these contexts lexicase selection will base selection on single test cases rather than on combinations of cases. We have seen this problem arise when applying lexicase selection to problems that give continuous errors, such as floating-point symbolic regression problems. In preliminary tests on problems of this nature we have seen lexicase selection perform poorly in comparison to tournament selection. One option for addressing this issue is to consider non-elitist forms of lexicase selection, in which we modify the specification for which candidates are eliminated at each step of the lexicase selection algorithm. For example, we could eliminate the worst $\frac{1}{n}$ of the remaining candidates at each step until one individual remains, where n is a small number such as 2 or 3. Another option would be to retain all candidates with errors within some predefined ϵ of the elite value for the case under consideration.

Our own primary motivation for developing lexicase selection derives from our interest in using genetic programming for general program synthesis problems, in which we aim to evolve general software from high-level behavioral tests. We have applied lexicase selection to a handful of hard problems of this type, such as the problem of finding a program to control a simple calculator, where inputs are given as button presses and outputs are floating-point numbers. Another problem to which we have applied lexicase selection involves finding a program that can take ten-pin bowling scores as string inputs and return the correct total score as an integer. Both of these problems are both uncompromising and difficult. While our work on these problems is still ongoing and we have not yet solved them to our satisfaction, the runs we have performed with lexicase selection do produce results that are better than than we have been able to produce using tournament selection.

In work not presented here, we have attempted to improve lexicase selection by biasing the order of test cases based on population statistics, so that cases that one might expect to require greater attention would be more likely to appear early in case sequences and hence have a greater influence on parent selection. While we have experimented with many variations on this theme we have yet to find a biasing scheme that outperforms the standard form of lexicase selection described above. Nonetheless, we suspect that lexicase selection might be improved by a principled method for biasing the ordering of the test cases, and we suggest that further research be conducted on this issue.

Another possible avenue of future research is to improve the run time of lexicase selection, which may be possible

through algorithmic improvements. Additionally, in this work we have used the most run time intensive version of lexicase selection; it may be possible to consider fewer test cases or a smaller random subset of the population (i.e. lexicase selection within a tournament set) without decreasing its performance significantly, leading to better run time. Even though selecting a single generation using lexicase selection is slower, in the genetic programming runs presented here the lexicase selection runs often used similar amounts of total wall-clock time as tournament runs. This is because the lexicase selection runs often used fewer generations total, since they found solutions earlier in runs. Even in cases where lexicase selection is slower overall, if it enables genetic programming to find solutions to problems that tournament selection cannot, its use is obviously worthwhile.

The work presented here applies lexicase selection only to genetic programming, but there is no obvious reason that it would not also be useful in other population-based evolutionary computation systems. It is applicable in any context in which parents are selected based on performance, and in which performance is assessed relative to more than one "case." Our hypothesis is that it will be most useful in uncompromising problems, but determining its full range of applicability is a topic for future research.

Of course, we do not expect lexicase selection to provide a "free lunch" [35] over all problems (even all uncompromising problems) or all evolutionary computation systems. It would not surprise us if it were possible to specify a problem and an evolutionary computation system for which solutions could only be reached via parents that are mediocre across all test cases. But considering the dramatic benefits observed for lexicase selection on the problems and systems examined here, we are optimistic about the prospects for lexicase selection when used on other problems and with other systems as well.

ACKNOWLEDGMENT

Thanks to D. Homer, W. La Cava, and the other members of the Hampshire College Computational Intelligence Lab for helpful discussions, to J. Erikson for systems support, and to Hampshire College for support for the Hampshire College Institute for Computational Intelligence.

REFERENCES

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

[2] R. I. McKay, "Fitness sharing in genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*. Las Vegas, Nevada, USA: Morgan Kaufmann, 10-12 Jul. 2000, pp. 435–442.

[3] 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*, vol. 14, no. 1, pp. 3–29, Mar. 2013.

[4] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, Jan.-Feb. 2012.

[5] W. B. Langdon and M. Harman, "Optimising existing software with genetic programming," *IEEE Transactions on Evolutionary Computation*.

[6] S. Luke and L. Panait, "Is the perfect the enemy of the good?" in *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*. New York: Morgan Kaufmann Publishers, 9-13 Jul. 2002, pp. 820–828.

[7] L. Spector, "Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report," in *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, ser. GECCO Companion '12. New York, NY, USA: ACM, 2012, pp. 401–408.

[8] S. Starosta and H. Nomura, "Lexicase parsing: A lexicon-driven approach to syntactic analysis," in *Proceedings of the 11th Coference on Computational Linguistics*, ser. COLING '86. Stroudsburg, PA, USA: Association for Computational Linguistics, 1986, pp. 127–132.

[9] T. Helmuth and L. Spector, "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*. Amsterdam, The Netherlands: ACM, 6-10 Jul. 2013, pp. 1627–1634.

[10] J. Noble and R. A. Watson, "Pareto coevolution: Using performance against coevolved opponents in a game as dimensions for pareto selection," in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001*. Morgan Kaufmann, 2001, pp. 493–500.

[11] W. B. Langdon, "Evolving data structures with genetic programming," in *Proceedings of the 6th International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 295–302.

[12] ——, "Advances in genetic programming." Cambridge, MA, USA: MIT Press, 1996, ch. Data structures and genetic programming, pp. 395–414.

[13] J. Horn, N. Nafpliotis, and D. E. Goldberg, "Multiobjective optimization using the niched pareto genetic algorithm," University of Illinois at Urbana-Champaign, 104 South Mathews Avenue, Urbana, IL 61801, Tech. Rep. IlliGAL 93005, 1993.

[14] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *Evolutionary Computation, IEEE Transactions on*, vol. 6, no. 2, pp. 182–197, 2002.

[15] E. Zitzler, M. Laumanns, and L. Thiele, "Spea2: Improving the strength pareto evolutionary algorithm," Swiss Federal Institute of Technology (ETH) Zurich, Tech. Rep. 103, 2001.

[16] G. Smits and M. Kotanchek, "Pareto-front exploitation in symbolic regression," in *Genetic Programming Theory and Practice II*, ser. Genetic Programming. Springer US, 2005, vol. 8, pp. 283–299.

[17] D. E. Goldberg and J. Richardson, "Genetic algorithms with sharing for multimodal function optimization," in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1987, pp. 41–49.

[18] S. Luke, *Essentials of Metaheuristics*, 1st ed. lulu.com, 2009.

[19] R. Smith, S. Forrest, and A. S. Perelson, "Population diversity in an immune system model: Implications for genetic search," in *Foundations of Genetic Algorithms 2*. Morgan Kaufmann, 1992, pp. 153–166.

[20] K. Krawiec and M. Nawrocki, "Implicit fitness sharing for evolutionary synthesis of license plate detectors," in *Applications of Evolutionary Computing, EvoApplications 2012*, ser. Lecture Notes in Computer Science, vol. 7835. Vienna, Austria: Springer, 3-5 Apr. 2013, pp. 376–386.

[21] J. Klein and L. Spector, "Genetic programming with historically assessed hardness," in *Genetic Programming Theory and Practice VI*, ser. Genetic and Evolutionary Computation. Ann Arbor: Springer, 15-17 May 2008, ch. 5, pp. 61–75.

[22] K. Krawiec and P. Lichocki, "Using co-solvability to model and exploit synergetic effects in evolution," in *PPSN 2010 11th International Conference on Parallel Problem Solving From Nature*, ser. Lecture Notes in Computer Science, vol. 6239. Krakow, Poland: Springer, 11-15 Sep. 2010, pp. 492–501.

[23] L. Spector, "Autoconstructive evolution: Push, pushGP, and Pushpop," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*. San Francisco, California, USA: Morgan Kaufmann, 7-11 Jul. 2001, pp. 137–146.

[24] L. Spector and A. Robinson, "Genetic programming and autoconstructive evolution with the push programming language," *Genetic Programming and Evolvable Machines*, vol. 3, no. 1, pp. 7–40, Mar. 2002.

[25] L. Spector, J. Klein, and M. Keijzer, "The Push3 execution stack and the evolution of control," in *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, vol. 2. Washington DC, USA: ACM Press, 25-29 Jun. 2005, pp. 1689–1696.

[26] L. Spector and T. Helmuth, "Uniform linear transformation with repair and alternation in genetic programming," in *Genetic Programming Theory and Practice XI*. Springer, 2013, p. In preparation.
[27] L. Spector, D. M. Clark, I. Lindsay, B. Barr, and J. Klein, "Genetic programming for finite algebras," in *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*. Atlanta, GA, USA: ACM, 12-16 Jul. 2008, pp. 1291–1298.
[28] R. McKenzie, G. McNulty, and W. Taylor, *Algebras, Lattices and Varieties*. Belmont, CA: Wadsworth and Brooks/Cole, 1987, vol. 1.
[29] 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 *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*. Philadelphia, Pennsylvania, USA: ACM, 7-11 Jul. 2012, pp. 791–798.
[30] J. A. Walker and J. F. Miller, "The automatic acquisition, evolution and reuse of modules in cartesian genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 12, no. 4, pp. 397–417, Aug. 2008.
[31] A. Agapitos and S. M. Lucas, "Learning recursive functions with object oriented genetic programming," in *Proceedings of the 9th European Conference on Genetic Programming*, ser. Lecture Notes in Computer Science, vol. 3905. Budapest, Hungary: Springer, 10 - 12 Apr. 2006, pp. 166–177.
[32] T. Helmuth and L. Spector, "Word count as a traditional programming benchmark problem for genetic programming," in *GECCO '14: Proceedings of the 2014 conference on Genetic and evolutionary computation*. Vancouver, BC, Canada: ACM, 12-16 Jul. 2014, pp. 919–926.
[33] L. Spector, B. Martin, K. Harrington, and T. Helmuth, "Tag-based modules in genetic programming," in *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*. Dublin, Ireland: ACM, 12-16 Jul. 2011, pp. 1419–1426.
[34] D. Jackson, "Promoting phenotypic diversity in genetic programming," in *PPSN 2010 11th International Conference on Parallel Problem Solving From Nature*, ser. Lecture Notes in Computer Science, vol. 6239. Krakow, Poland: Springer, 11-15 Sep. 2010, pp. 472–481.
[35] D. Wolpert and W. Macready, "No free lunch theorems for optimization," *Evolutionary Computation, IEEE Transactions on*, vol. 1, no. 1, pp. 67–82, 1997.

Thomas Helmuth received the B.A. degree in computer science and mathematics from Hamilton College, Clinton, NY, USA in 2009, and the M.S. degree in computer science from the University of Massachusetts, Amherst, MA, USA in 2012.

He is currently working towards the Ph.D. degree at University of Massachusetts, Amherst, MA, USA. His research interests focus on automatic program synthesis for the creation of general software by genetic programming.

Lee Spector received the B.A. degree in philosophy from Oberlin College, Oberlin, OH, USA in 1984, and the Ph.D. degree in computer science from the University of Maryland, College Park, MD, USA in 1992.

He is currently a Professor of Computer Science in the School of Cognitive Science at Hampshire College in Amherst, MA, USA, and an Adjunct Professor in the Department of Computer Science at the University of Massachusetts, Amherst, MA, USA. He conducts research in artificial intelligence, artificial life, and a variety of areas at the intersections of computer science with cognitive science, physics, evolutionary biology, and the arts.

Dr. Spector is the Editor-in-Chief of the journal *Genetic Programming and Evolvable Machines*, a member of the editorial board of the journal *Evolutionary Computation*, and a member of the Executive Committee of the ACM Special Interest Group on Evolutionary Computation (ACM SIGEVO).

James Matheson received the B.A. degree in genetic programming and behavioral economics from Hampshire College, Amherst, MA, USA in 2014.

He is a Co-Founder of a communications software start-up called Trext. His research interests include neuroscience, economics, and modeling human behavior with evolutionary algorithms.