GENERAL PROGRAM SYNTHESIS FROM EXAMPLES USING GENETIC PROGRAMMING WITH PARENT SELECTION BASED ON RANDOM LEXICOGRAPHIC ORDERINGS OF TEST CASES

A Dissertation Presented

by

THOMAS HELMUTH

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2015

College of Information and Computer Sciences

GENERAL PROGRAM SYNTHESIS FROM EXAMPLES USING GENETIC PROGRAMMING WITH PARENT SELECTION BASED ON RANDOM LEXICOGRAPHIC ORDERINGS OF TEST CASES

A Dissertation Presented

by

THOMAS HELMUTH

Approved as to style and content by:

Lee Spector, Chair

David Jensen, Member

Yuriy Brun, Member

Adam Porter, Member

James Allan, Chair of the Faculty
College of Information and Computer Sciences

ACKNOWLEDGMENTS

No research is conducted in a void. My collaborators have contributed greatly to the ideas and efforts presented here: Lee Spector, James Matheson, William La Cava, Kourosh Danai, Nicholas Freitag McPhee, David Donatucci, Krzysztof Krawiec, Paweł Liskowski, Kyle Harrington, and Brian Martin. Additionally, the members of the Hampshire College Computational Intelligence Lab and Bill Tozier have contributed enumerable conversations that have shaped my work. My committee members, David Jensen, Yuriy Brun, and Adam Porter have helped me see my work through different lenses and gave new dimensions to this dissertation.

I cannot overstate my appreciation of my advisor, Lee Spector, who has guided me throughout graduate school. I will always appreciate his creativity and ability to focus on interesting questions. Lee has done a wonderful job teaching me how to be an academic, both in research and teaching.

I'd like to specifically thank Nicholas McPhee, who over the past year has been a tremendous mentor and collaborator. Our talks over tea have had an enormous impact on both the work in this dissertation and my skills as a researcher.

Some of my most enjoyable and productive hours in grad school were spent driving to and from the curling rink with Robert Walls, who always provided enriching and entertaining conversation. Thanks to him and Jason Beers for great times on the disc golf course and playing board games.

Josiah Erikson's systems support has been irreplaceable, allowing me to generate all of the data in this dissertation using the Hampshire College computing cluster.

Most importantly, my family has been ever supportive of my goals. Fiona (and Ben!) have brightened every day of my long graduate school experience. My parents and sister have always shown interest in my work, even when it sounded like gibberish.

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.

ABSTRACT

GENERAL PROGRAM SYNTHESIS FROM EXAMPLES USING GENETIC PROGRAMMING WITH PARENT SELECTION BASED ON RANDOM LEXICOGRAPHIC ORDERINGS OF TEST CASES

SEPTEMBER 2015

THOMAS HELMUTH

B.A., HAMILTON COLLEGE

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Lee Spector

Software developers routinely create tests before writing code, to ensure that their programs fulfill their requirements. Instead of having human programmers write the code to meet these tests, automatic program synthesis systems can create programs to meet specifications without human intervention, only requiring examples of desired behavior. In the long-term, we envision using genetic programming to synthesize large pieces of software. This dissertation takes steps toward this goal by investigating the ability of genetic programming to solve introductory computer science programming problems.

We present a suite of 29 benchmark problems intended to test general program synthesis systems, which we systematically selected from sources of introductory com-

puter science programming problems. This suite is suitable for experiments with any program synthesis system driven by input/output examples. Unlike existing benchmarks that concentrate on constrained problem domains such as list manipulation, symbolic regression, or boolean functions, this suite contains general programming problems that require a range of programming constructs, such as multiple data types and data structures, control flow statements, and I/O. The problems encompass a range of difficulties and requirements as necessary to thoroughly assess the capabilities of a program synthesis system. Besides describing the specifications for each problem, we make recommendations for experimental protocols and statistical methods to use with the problems.

This dissertation's second contribution is an investigation of behavior-based parent selection in genetic programming, concentrating on a new method called lexicase selection. Most parent selection techniques aggregate errors from test cases to compute a single scalar fitness value; lexicase selection instead treats test cases separately, never comparing error values of different test cases. This property allows it to select parents that specialize on some test cases even if they perform poorly on others. We compare lexicase selection to other parent selection techniques on our benchmark suite, showing better performance for lexicase selection. After observing that lexicase selection increases exploration of the search space while also increasing exploitation of promising programs, we conduct a range of experiments to identify which characteristics of lexicase selection influence its utility.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Automatic programming of general software, long a goal of computer science [84, 44], requires a specification of the desired program. While formal specifications offer one approach, they require some understanding of the algorithmic basis of the synthesized program. A simpler alternative requires only the specification of the desired program behavior in the form of input/output pairs, or *examples*. Despite recent advances in the synthesis of specialized, domain-specific code, the synthesis of general-purpose software from examples remains unachieved.

Genetic programming (GP) borrows ideas from biological evolution to synthesize functions or programs [61]. In GP, a *population* of programs (or *individuals*) undergoes selection and variation to stochastically search for a desired program. GP has traditionally been used to solve many different types of problems, including general and domain-specific program synthesis, symbolic regression, boolean logic, design, and classification. A significant amount of recent GP research and application has focused on symbolic regression, classification and other domains that only require small, domain-specific instruction sets [107, Ch. 12], [45]. Relatively little recent work has explored general programming with large general-purpose instruction sets.

The primary contributions of this dissertation include:

- We have created a program synthesis benchmark suite based on examples as specifications. These 29 problems, along with our recommendations for experimental and statistical methods, fill a void of general program synthesis benchmark problems in both GP and other program synthesis fields.

- We show that a GP system, PushGP, can solve many of the problems in our benchmark suite. We attempt to solve the problems with other common program synthesis systems without success. This demonstrates PushGP as the first general-purpose program synthesis system.

- We investigate lexicase selection, a new parent selection technique for GP. This investigation shows that lexicase selection significantly outperforms other common parent selection mechanisms. We then present empirical evidence to help explain the differences in success that we observed, contributing to the understanding of parent selection in GP.

1.1 Applications

The ability to automatically synthesize programs from examples can enable or improve a variety of applications. For example, the narrow, domain-specific program synthesis system Flash Fill has recently been included in Microsoft Excel, enabling non-technical users to perform repetitive string manipulation tasks by providing examples. This would otherwise require them to write short macros, which are instead synthesized on the fly [25].

General program synthesis could assist advanced programmers, allowing a programmer to write the higher-level program structure while leaving the lower-level functions to be implemented by the synthesis system. Additionally, many companies need to modernize legacy code; automatic program synthesis could make use of the legacy code as an "oracle" to provide the desired outputs for given inputs, allowing code to be generated without intervention from a human programmer. General program synthesis from examples may also have applications to computer science education, where a new programmer can demonstrate the inputs and outputs they wish their program to handle, and the synthesis system can lead them toward a working program. Our longer-term goal is to enable the synthesis of large parts of software

applications entirely from examples, allowing software engineers to concentrate on creating tests for the software.

The related fields of automatic debugging and automatic program improvement have recently seen much interest, in particular systems that make use of GP. In automatic program debugging, programs that fail some test cases but pass others are altered until they pass all available test cases [74, 76, 73, 77, 133, 75, 116, 2, 5, 105, 104]. An automatic program improvement system takes a program that passes its functional test cases and improves its other characteristics such as runtime or memory requirements [69, 68, 3, 29, 134]. While we will not focus on automatic debugging or improvement, our techniques presented here may be applicable in these fields as well.

1.2 Program Synthesis System Requirements

We wish to investigate systems that can perform domain-agnostic program synthesis from examples. In order for such a system to prove useful for a variety of applications, it should meet the following requirements, which ensure that the system can synthesize arbitrary, domain-specific programs that generalize the given examples to unseen inputs:

1. **The system should not assume the existence of any code on which it can build or improve, but should synthesize code from scratch.** Although synthesis systems can perform interesting tasks in the areas of automatic bug fixing and code improvements, the applications listed above require synthesis to start from nothing and create code to meet specifications.

2. **Specifications for the desired programs are given as a set of input/ output examples without a specific order.** No other domain knowledge should be needed, unless specialized instructions are required beyond a standard general-purpose instruction set. In order to improve accessibility, we do not

want the user to have to know the form of or constraints on the program they wish to create, meaning they should not have to provide formal specifications. Some systems require examples to be given in a particular order, such as "in increasing complexity" [56, 106, 57]. This requires the user to have at least basic knowledge of the program they wish to synthesize, possibly even requiring an understanding of recursion; we wish to avoid such requirements as they restrict potential users and applications.

3. **The system must be able to find solutions to problems that require a moderate number of instructions, at least equivalent to a complex function or method written by a human programmer.** Small programs can be trivially synthesized through brute-force search. Even when using more advanced techniques, many program synthesis systems can generate relatively small programs in reasonable amounts of time, but fail after a certain point because they cannot handle the exponential nature of the search space.

4. **The system must be able to generate programs that use a large set of general-purpose instructions and operate on a variety of data types.** At minimum the instructions should operate on integers, strings, and booleans, with other options including floating-point numbers, characters, and data structures such as lists and trees. Instruction sets should be general enough to represent solution programs for a wide variety of synthesis problems, not only problems from a specific domain.

5. **The language of synthesis must be able to perform arbitrary calculations, meaning it must be Turing-complete.** This constraint means that the programming language must be able to express control flow, possibly including conditional execution, iteration, and recursion.

6. **The synthesized program must generalize to unseen example inputs.** The goal of program synthesis is to create a program that generates the correct outputs on all inputs in the input space, not just those used to synthesize the program. One can trivially synthesize a program that produces the example outputs by memorizing them; this type of overfitting would likely have no utility to a user.

To our knowledge, no existing program synthesis systems have been shown to meet all of these requirements.

Depending on the scope and complexity of the desired program, a synthesis system may create a single program that fulfills the requirements, or may need to synthesize multiple interconnected smaller programs that make up a larger program. Some recent work in software synthesis using GP suggests one way in which multiple programs can evolve while making use of each other to create a larger piece of software [6]. We could envision other viable methods, such as a hierarchical software synthesis system that identifies requirements of modules and then recursively creates them in a top-down synthesis approach. For now, we simply wish to synthesize single programs (which may create helper functions) that perform the desired behaviors, which could potentially be adopted into larger synthesis systems.

1.3 General Program Synthesis Benchmarking

Several GP researchers have highlighted the need for better benchmark problems to guide research in the field [85, 135, 138]. While various benchmark problems have been proposed, very few test for the ability to perform general program synthesis, even though this type of benchmark received high interest in a recent community survey [135]. Our survey of problems that have been used to test program synthesis systems outside of GP found no acceptable problems as well; most such problems test for the

ability to generate small programs in domain-specific fields such as list manipulation or string processing ([40] for example), failing one or more of the requirements above.

In order to enable the benchmarking of general program synthesis systems, we have developed a suite of 29 problems, which we present in Chapter 3. We systematically selected these programs from sources of introductory-level computer science course homework problems. The problems require a wide range of programming techniques, data types, and program structures representative of the types of programs solved by undergraduate programmers, and hopefully programmers more generally. We describe each problem in natural language and give recommendations for specifications in the form of input/output examples. We additionally suggest experimental protocols and statistical methods to be used with the benchmark problems. We show that this benchmark suite can help differentiate the strengths and weaknesses of general program synthesis systems.

1.4 Formalization of Program Synthesis from Examples

Before proceeding further, let us formalize some definitions we use when discussing general program synthesis from examples. The specification for a desired program is given by a set of examples $E \subset X \times Y$, where X is the space of inputs and Y is the space of outputs. Each example $(\mathbf{x}, \mathbf{y}) \in E$ consists of program input(s) $\mathbf{x}$ and expected output(s) $\mathbf{y}$[1]. The synthesis system defines a space of considered programs $\mathcal{P}$, constrained by system parameters such as instruction set, program syntax or grammar, maximum program size, etc. Each program $p \in \mathcal{P}$ can be interpreted as a function $p : X \to Y$. The objective of synthesis is to generate a program $p^* \in \mathcal{P}$ that passes all of the examples, such that for every example $(\mathbf{x}, \mathbf{y}) \in E$, $p^*(\mathbf{x}) = \mathbf{y}$.

[1] $\mathbf{x}$ may consist of one or more inputs and $\mathbf{y}$ one or more expected outputs, all of which may have different data types.

When using GP for program synthesis, we define one or more *error functions* $f_1, ..., f_n \in \mathcal{F}$. Each error function takes an example and a program, and returns a positive, real number *error value*; thus $f_i : E \times \mathcal{P} \to [0, \infty)$. The purpose of an error function is to indicate the distance[2] between a program's output $p(\mathbf{x})$ and the desired output $\mathbf{y}$, with 0 being the optimal error value. For example, one common error function used for numeric outputs is the absolute difference $|p(\mathbf{x}) - \mathbf{y}|$.

We call a pair composed of an example and an error function a *test case* in the space $E \times \mathcal{F}$. We often split the set of test cases into a *training set* T_s, used for guiding the synthesis algorithm, and an *unseen test set* T_u, used to test the generalization of a program on withheld data. In GP it is common to define the *fitness* of a program as the summed *total error* across the test cases in the training set; in particular

$$f_{TE}(p) = \sum_{((\mathbf{x},\mathbf{y}),f_i)\in T_s} f_i((\mathbf{x}, \mathbf{y}), p) \tag{1.1}$$

When using absolute difference as the error function, Equation 1.1 is equivalent to the absolute value norm[3]. Thus, the objective of program synthesis can be restated as finding a program $p^* \in \mathcal{P}$ such that $f_{TE}(p^*) = 0$. Such a program may or may not also achieve zero error on the unseen test set T_u; if it does, we call it a *solution* or *successful program.*

1.5 Parent Selection in Genetic Programming

Within our chosen synthesis paradigm of GP, we wish to investigate the effects of parent selection mechanisms on problem-solving performance. The *parent selection* algorithm chooses a program from the population to genetically vary through mutation or crossover to produce a child for the next generation. Most common parent

[2]Note that error functions do not need to satisfy the conditions of being a metric.

[3]Other norms are commonly used as fitness functions, such as Euclidean distance.

selection methods use a single scalar fitness value such as total error (Equation 1.1) to bias stochastic selection toward fitter individuals. For example, *tournament selection* with tournament size N uniformly chooses a tournament pool of N individuals from the population and then selects the one with the best fitness.

While it is convenient to define the objective of GP in terms of an aggregate fitness value such as total error, this does not necessarily make it an efficient driver for guiding search [63, 65]. An aggregate fitness value necessarily discards information about a program's performance across the test cases in the training set, which could otherwise be useful for directing selection. For example, some test cases might be easier to pass than others, yet a scalar fitness treats all tests as equally valuable. As Krawiec et al. [65] summarize, "The conventional scalar evaluation denies a search algorithm access to the more detailed information on program's behavioral characteristics, while that information could help to drive the search process more efficiently". While this view is not unique in GP, the conventional aggregate fitness approach still dominates the field.

Instead, Krawiec et al. recommend the development of *behavior-based search drivers* that utilize more information-rich sources of program performance than a scalar fitness value can provide [65]. Such sources range from a program's error values across the training set to execution traces of a program's effects on an instruction-by-instruction basis. Krawiec et al. claim that behavior-driven GP can contribute to a range of benefits, from increased population diversity to better performance [65].

In this work we explore *lexicase selection*, a behavior-based parent selection technique that only compares individuals' results on single test cases at a time, never directly comparing results on separate test cases [123]. Initial tests indicate that lexicase selection improves performance of GP on a range of problems, including one software synthesis problem [39]. In Chapter 4, we describe and extensively test lexicase selection in comparison to other parent selection techniques. In particular, we

examine ways in which lexicase selection affects the breadth and depth of the GP search, allowing it to maintain a diverse population while concentrating search effort on promising individuals. We also conduct systematic experiments that present evidence both for and against hypotheses that attempt to explain the improved performance and population diversity observed when using lexicase selection.

1.6 Comparisons to Other Systems

Our final contribution is a brief comparison of our results using GP to other program synthesis techniques. In Chapter 5, we attempt to solve problems from our benchmark suite using Flash Fill [25], MagicHaskeller [48, 46], and Sketch [120, 119], three techniques representative of the field as a whole. We hope that similar experiments using our benchmark suite can help advance the field of program synthesis.

1.7 Collaborators

All research undertaken here was conducted under the supervision of Lee Spector. The work in Chapter 2 contains contributions from Lee Spector [124, 39]. Chapter 3 presents research conducted with Lee Spector and James Matheson [36, 38, 37, 39]. Collaborators on work in Chapter 4 include Lee Spector, Nicholas Freitag McPhee, James Matheson, and David Donatucci [39, 37, 88, 33]. Karthik Kannappan contributed to the research presented in Chapter 5.

CHAPTER 2

RELATED WORK

Systems that perform automatic program synthesis require a specification for the program that the user wishes to create. Some systems use functional or logical program specifications, which require the user to know how to create one of these formal specifications for their problem and often require insight into the structure of the desired program. Other systems only require input/output examples that show correct behavior for the desired program; we concentrate on such systems, which place less burden on the user to know how the program should work since they only need to specify what the program should do.

All program synthesis systems also require a description of the programming language in which the synthesized programs will be written. A variety of options exist for specifying the language of synthesis. Many systems, including many forms of GP, define the language by providing an instruction set of the statements allowable in the programs, which may be broken into sets of functions and terminals. Other systems define the language through a grammar, which may provide more information as to the allowable structure of the synthesized programs. No matter how the language is specified, a system should be able to synthesize programs that use a wide variety of instructions over multiple data types in order to solve general-purpose programming tasks.

2.1 Program Synthesis From Examples

Inductive program synthesis systems automatically create programs based on incomplete specifications [24, 21, 55, 40]. In particular, in *program synthesis from examples* (which others have called *programming by example*), specifications are limited to desired program behaviors given as input/output examples [26, 79, 112]. The related field of programming by demonstration has users provide more informative specifications, such as demonstrating the steps required to perform the task [16, 72, 71, 70]. Another system, Sketch, requires the user to provide a partial program with pieces missing to be filled in by synthesis [120, 119]. An abundance of other techniques exist if you relax specifications further, for example by requiring formal specifications of the desired program, with [55, 24, 21, 13, 95] providing good surveys of inductive program synthesis. We will concentrate solely on program synthesis from examples, where the specification is entirely contained in the examples and does not rely on additional information.

Many program synthesis from examples systems assume that the user can only specify a small number of examples, often five or fewer. Since a small number of examples does not strongly specify a particular program, these systems must make other assumptions about the space of problems from which the problem is selected, or must require the user to provide hand-crafted clues about examples in the problem domain. On the other hand, we wish to synthesize general programs based entirely on behavioral examples, and want the resulting programs to generalize to unseen inputs, meaning we will require many more examples, often 100 or more.

Many existing inductive program synthesis systems will have difficulties with general-purpose program synthesis. In particular, many of them have been shown to exhibit prohibitively long run times if presented with the number of examples we expect to use. Additionally, some will not perform well with the large, general instruction sets we expect our synthesis systems to use. Almost all synthesis systems

perform well for problems that have solutions with small numbers of instructions, but do not scale well to more complex problems that require larger solution programs.

Two major categories of methods exist for synthesizing programs from examples: *analytic methods*, which use logic-based techniques to only synthesize programs that fulfill the example specifications, and *search-based* (or "generate-and-test") methods that create many programs to search for those that pass the examples. Genetic programming is a search-based program synthesis method but will be treated separately in Section 2.1.3.

2.1.1 Analytic Methods

IGOR2 combines analytic and generate-and-test inductive function programming into a system that generates recursive programs from examples [56, 53, 54]. This type of inductive programming performs well on structural problems such as list manipulations. Though this system can quickly synthesize programs for some simple benchmarks, it has not been tested on problems that require more general programming techniques. Its performance is greatly diminished when the given instruction set (referred to in these papers and in other inductive programming settings as "background knowledge") contains many unnecessary instructions. This means if many different instructions are likely necessary to solve the problem, but it is not clear which ones are necessary, IGOR2 will find it difficult to distinguish which instructions are important and will incur significant slowdowns. Another disadvantage of IGOR2, as well as some other analytic inductive programming systems [131, 57], is that the examples must be "complete" and presented in order of most simple to most complex. Though it is unclear what is meant by "complete," it appears that this requires the user to have some intuition about the recursive nature of the program in order to present the examples properly. IGOR2 requires enough examples to fully specify the problem, but too many more additional examples cause the system to slow

significantly, even for a version of the system that was designed to be more robust to changes in the examples [49]. These requirements of the examples greatly limit IGOR2's applications, in that the user must carefully prescribe the examples in correct order without giving too many. While these constraints on examples have proven to work for simple benchmark problems, they have not yet been demonstrated to work on more general program synthesis problems. Additionally, they limit IGOR2's users to those who can correctly specify the examples.

Flash Fill, recently added to Microsoft Excel, uses version-space algebras to perform program synthesis from examples on string manipulation tasks [25]. This research has developed methods for quickly creating simple programs for one-off repetitive tasks in spreadsheet applications for non-programmers [27, 32, 113, 114]. Building on the use of version-space algebras for programming by demonstration [72], Flash Fill assumes few example inputs and must make simplifying assumptions about the problem space. In particular, the domain-specific language used here is designed for small string manipulation tasks that an end-user may want to perform without knowing how to program them. Adapting the technique for new domains would require a different domain-specific language that is carefully crafted to meet problem requirements while restrictive enough to allow for quick searching. Each different domain-specific language would also require a new synthesis technique; it is unclear whether Flash Fill's methods could even be adapted for general program synthesis, or if they require a more restricted domain. Related work uses Bayesian inference and represents programs using combinatory logic with similar assumptions and restrictions [78]. We aim to synthesize programs from wider domains using general-purpose programming.

2.1.2 Search-Based Methods

Menon et al. [89] use machine learning to more efficiently search for programs to perform string manipulation for end-users. Here, parameters for a probabilistic context-free grammar are learned, and then the space of programs is searched starting with the most likely program based on the grammar. This work assumes that the user will provide very few examples, usually one or two. In order to find correct programs from such few examples, the parameters for the model must be learned ahead of time, based on a corpus of training data made up of correct program outputs. These correct training examples must be drawn from similar tasks to the those on which the model will run. Each training example needs to have an associated correct program, although this arduous step could possibly be avoided using bootstrapping. Additionally, a large set of hand-written clues must be provided that are used to determine which rules in the grammar may be relevant for a particular input. The hand-crafted clues and extra training data, especially the correct programs, go beyond what is necessary in our problem formulation and put a large burden on the user for any new problem domain.

Escher synthesizes recursive programs using only examples for specifications and no domain-specific knowledge beyond the instructions to use in the synthesized programs [1]. It guides an exhaustive search for programs primarily by program size, with smaller programs being preferred. Escher's assumptions about the use of examples as specification line up well with ours. It also seems to gracefully handle large instruction sets, multiple instruction and literal types, and control flow. But, it has only been tested on toy problems using artificially simple instructions. The method used for program generation raises concerns over how well the synthesized programs will generalize to unseen inputs, and no generalization data is presented. It is unclear whether Escher could handle problems that require more than 20 or so instructions, since all programs with fewer instructions are tried first. To ensure

termination, ESCHER requires that the arguments to synthesized recursive functions "decrease" based on a type-dependent ordering, meaning that some programs cannot be synthesized since the language for the programs is not Turing complete; this could cause issues with a problem such as generating the Collatz sequence. Finally, ESCHER seems to require examples showing the results of every recursive call, meaning the inputs are restricted in a similar way to systems such as IGOR2. This system shows a lot of promise, but needs to show results on more realistic problems with looser requirements on examples in order to fully evaluate it.

Test-driven synthesis [106] uses ideas from test-driven development along with component-based synthesis to build programs while incrementally adding increasingly complicated examples. Although this system only requires binary examples (right or wrong), the dependence on ordered examples places extra burden on the user. Additionally, an expert-written domain-specific language as a context-free grammar is required for each new problem domain, making it difficult for users to add new problem domains. This system performs well at doing general program synthesis when numbers of instructions and program sizes are relatively small. But since the search is mostly exhaustive using the provided CFG, it does not scale well; it cannot create programs with more than about 20 instructions or when more than about 50 instructions are provided in the domain-specific language.

MagicHaskeller [48, 46] synthesizes functional Haskell programs through an exhaustive search of programs with the correct type signatures. It uses a Monte-Carlo algorithm to remove semantically equivalent programs from the search space [47]. More recently, it has also integrated an analytic component based on IGOR2, which allows it to synthesize a greater range of programs than can be found in reasonable time using exhaustive search [50]. Additionally, a web interface is available running a time-limited version of MagicHaskeller intended as a Haskell teaching tool for new programmers [51]. These implementations make great use of Haskell's functional in-

structions, and they perform well on problems that require list manipulations and structural changes. While MagicHaskeller performs very quickly on simple problems without too many examples, like IGOR2 it has trouble with problems that require a large number of examples to illuminate the relevant edge cases. Additionally, it seems to have trouble with problems that require conditional control flow.

ADATE is a search-based synthesis system inspired by biological evolution, but is not typically considered genetic programming [98, 100, 99]. In particular, it uses specialized deterministic genetic operators to exhaustively search for programs. ADATE requires specifications similar to GP, as example inputs and outputs with an output evaluation function similar to a fitness function in GP. Since ADATE creates programs of increasing size exhaustively, it has trouble solving problems requiring larger programs. In [40] ADATE was compared to other synthesis techniques including IGOR2 and MagicHaskeller on some list manipulation problems; it had moderate success, solving one problem that no other system could but taking much longer than the other systems on most other problems while being unable to solve two problems.

2.1.3 Genetic Programming

Although much of the research in GP could be considered relevant to general program synthesis from examples, we will focus only on recent developments specifically related to general program synthesis or the creation of Turing-complete programs.

Arcuri and Yao use GP to synthesize programs based on formal specifications [6, 4]. This research works toward the long-term goal of evolving general software from scratch, and makes some advances not seen elsewhere. Their system uses co-evolution to adapt the set of examples during evolution to try to find areas to test that will make the programs fail. This is reminiscent of the co-evolution of fitness predictors [110, 109, 111], though these co-evolutionary methods have different goals and behaviors. Another interesting aspect of their system is the use of N-version programming [7,

19] to create ensembles of synthesized programs that usually perform better than individually synthesized programs. Since they are interested in evolving full software, they propose a framework for evolving multiple interrelated programs that compose a larger whole; their experiments only make rudimentary use of this system, but it lays the groundwork for the evolution of full software systems. Although this line of research makes strides toward general program synthesis using GP, it also does not meet some of the constraints we suggest as fundamental to general program synthesis from examples. In particular, this system requires formal specifications in order to create the examples they co-evolve to test their programs, where we would prefer the user to only need to provide input/output examples. They use a somewhat minimal instruction set that features booleans, integers, and array access. These instructions are not general enough to solve the majority of problems; for example, they only allow for one variable and one array. They provide only a single iteration instruction, which is highly constrained to ensure good behavior. Finally, while they test their system on a handful of problems and also check for generalization to unseen data, the problems they test on are all moderately simple array manipulation problems such as sorting; we would be very interested to see results of their system on a wider variety of problems that require larger programs.

The recent GP system FINCH evolves Java bytecode, allowing general Java programs to be synthesized by decompiling the bytecode [105, 104, 103]. The evolved programs can make use of Turing-complete instructions including iteration and recursion. FINCH requires a Java program to be provided as input, which is used to initialize the entire population; this program could be a partially completed or buggy program, but in some examples they simply provide junk code that simply exhibits all of the provided instructions. Still, it is unclear how well this system would perform if initialized with random programs, as is typically used in GP, instead of code crafted for each problem. The instruction sets used in these experiments tend to be small

and problem-specific; since FINCH is evolving low-level Java bytecode, it is unclear how well it would do when given a large set of more advanced instructions. So far FINCH has only been tested on relatively simple toy problems, so we are interested in how it would fare on more difficult problems.

In order for a GP system to perform general program synthesis, it must be able to handle multiple data types. The most common mechanism for allowing multiple data types in GP is based on strongly-typed GP [92], in which each instruction has input and output types and genetic operators ensure that type constraints are obeyed. Recent work in Cartesian genetic programming has allowed for multiple data types, though thus far has been limited to floating point numbers and lists of numbers and does not have general control flow available [28].

Grammar guided genetic programming, and in particular grammatical evolution, have become common frameworks for GP [101, 102]. Even though having the guidance of a grammar seems like it could be a boon for evolving general programs with multiple data types, little work has been done in this area, possibly because of difficulties related to recursion and iteration [87].

Various GP practitioners have added looping or recursion to their evolutionary languages in order to evolve programs in a Turing-complete language, but most of these lack other abilities required for synthesizing general software. For example, most only provide a very limited and problem-specific instruction set. Teller describes a general framework for Turing-complete GP without meeting most of the other constraints for general program synthesis [132]. Woodward presents a Turing-complete tree-based GP using a specialized crossover operator that respects modules, but uses a small instruction set and does not present substantial results [139]. Moraglio et al. [94] facilitate the evolution of recursive functions by replacing recursive calls in evolving programs with the correct value, which must be given as an example; this requires examples to be "complete" up to a certain point, resulting in similar issues

to other analytic and search-based synthesis methods described above. Withall et al. [136] use a block-based representation including restricted looping and a small set of integer and list instructions to synthesize solutions to moderately simple list manipulation problems such as summing or reversing a list, similar to problems used in many inductive programming experiments such as [40]. This work does test the generality of the evolved programs on unseen examples, though all examples use small input lists with less than ten elements. Binard and Felty develop a simple lambda calculus program representation that allows for various forms of recursion and iteration, though they only show results on simple problems [10, 9]. None of these systems have been shown to perform general program synthesis as we have defined it.

Yu allows GP to synthesize recursive programs by providing higher-order functions that implement recursion instead of allowing recursive calls directly [140]. This approach seems noteworthy and a potential direction for future research. Unfortunately, Yu uses tiny instruction and terminal sets tailored very specifically to the substring and Fibonacci problems that are presented, as opposed to the large, general-purpose instruction sets needed to solve general programming problems.

2.1.3.1 Push and PushGP[1]

Our work primarily uses the GP system PushGP, which evolves programs expressed in the Push programming language. Push is a stack-based postfix language that was designed specifically for use in GP systems [121, 128, 126]. Push is general purpose and Turing complete, with support for iteration, recursion, conditional execution, and automatically-defined subroutines through runtime code manipulation and tagging [126, 127]. Push and PushGP implementations exist in C++, Java, JavaScript, Python, Common Lisp, Clojure, Scheme, Erlang, Scala and R. Many

[1]Parts of this section originally appeared in a book chapter co-authored with Lee Spector [124].

of these are available for free download from the Push project page.[2] The results presented here were obtained using the Clojure implementation of PushGP called Clojush[3].

Push dedicates a separate stack for each data type. Instructions take their arguments from stacks of the appropriate types and they leave their results 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.

Push traditionally takes results from the top items on stacks after executing a program. For example, a program that returns an integer will take the top value on the integer stack as its output. This convention means that some evolved programs might not return a value of the correct type, if the relevant stack is empty following execution; in this scenario, we penalize the program for failing to return a result. Push also has the ability to print literals to standard output; in our implementation this simply concatenates printed material to a string of whatever has been printed so far, but this could easily be changed to print to standard output.

Many of Push's most unusual and powerful features stem from the fact that code is itself a Push data type, and from the fact that Push programs can easily (and often do) manipulate their own code as they run. Push programs may be hierarchically structured into code blocks delimited by parentheses. This hierarchical structure affects the evaluation of code-manipulation instructions. For example, the `exec_if` instruction removes the second code block on the `exec` stack if the top item on the `boolean` stack is true, and removes the first code block on the `exec` stack if it is false.

---

[2] `http://pushlanguage.org/`

[3] `https://github.com/lspector/Clojush`

In a change from previous versions of PushGP, the most recent version of Clojush does not evolve Push programs directly, but instead uses a separate linear genome representation that we translate into Push programs prior to execution. The new *Plush*[4] *genomes* are linear sequences of instructions that may have one or more epigenetic markers attached to each instruction. The epigenetic markers affect the translation of the Plush genomes into Push programs. For example, the *silent* marker is a boolean that tells whether a particular instruction will appear in the translated program.

When evolving Push programs directly, we often found that parenthesis-delimited code blocks would rarely evolve in conjunction with instructions that made use of them. One of the motivations for moving to linear Plush genomes was that we could require that Push instructions that make use of code blocks be followed by them. With this change, every instruction that takes one or more argument from the *exec* stack implicitly opens one or more code blocks. Additionally, each instruction has a *close* epigenetic marker that tells the number of code blocks to end after that instruction. Thus, during translation from Plush genome to Push program, an open parenthesis is placed after each instruction that requires a code block, and a matching closing parenthesis is placed after a later instruction with a non-zero close marker. Note that these code blocks can create hierarchically nested Push programs. For example, if a new code block B is opened after the start of code block A, the next close marker will close block B, not block A. If not enough close markers occur by the end of a program to match all opened code blocks, all opened blocks are simply closed.

Another advantage of moving to linear Plush genomes instead of traditional tree genomes or hierarchical Push programs is that it enables simple use of uniform genetic operators. The uniform genetic operators implemented in Clojush are inspired by the operator ULTRA, which was designed for Push programs, requiring them

[4]**L**inear **Push**

to be translated into a linear form and back [124]. The main crossover operator, *alternation*, traverses two parents in parallel while copying instructions from one parent or the other to the child. While traversing the parents, copying can jump from one parent to the other with probability specified by the *alternation rate* parameter. When alternating between parents, the index at which to continue copying may be offset backward or forward some amount based on a random sample from a normal distribution with mean 0 and standard deviation set by the *alignment deviation* parameter. We also use a *uniform mutation* operator that traverses a parent and with some probability replaces each instruction with a random one. In order to manipulate the locations of closing parentheses, we include a *uniform close mutation* operator that may increment or decrement the close epigenetic marker of any given instruction. Finally, we allow genetic operator pipelines that combine multiple operators; for example, we often use alternation followed by uniform mutation, which closely resembles ULTRA [124].

We have used PushGP for a variety of program synthesis tasks, including mimicking the Unix word count utility wc [36], creating the factorial function using a large instruction set [124], creating digital multiplier programs [35], and developmentally evolving SQL queries [34]; of these, only the wc problem meets all of our constraints of general program synthesis. Some of the work in Push has been motivated by autoconstructive evolution, where the programs themselves are executed to create the children programs for the next generation [122, 128, 121, 31]. Keijzer developed pushforth as a single-stack language for evolving programs [52]. All considered, very little of this work has synthesized programs using large, general-purpose instruction sets.

2.2 Parent Selection in Genetic Programming[5]

Since this dissertation focuses on effects of parent selection on GP's ability to perform general program synthesis from examples, here we will survey related parent selection mechanisms. In particular, we concentrate on the effects of population diversity on performance, and present parent selection methods that implicitly or explicitly affect diversity preservation. This section covers work including multiobjective GP and fitness sharing.

To some extent one can consider multiple test cases that a program must pass to be the multiple objectives in a multiobjective optimization problem [97]. This approach has been called the *multi-objectivization* of a single-objective problem into a multi-objective problem [59]. 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.

A variety of multiobjective approaches have been presented in the literature. *A priori* methods express preferences prior to running an algorithm. For example, in the the lexicographic ordering method [107, p. 80] the objectives are sorted based on the user's prediction of relative importance ahead of time, and solutions are compared lexicographically on the objectives.

By contrast, *a posteriori* methods do not incorporate user input regarding the relative importance of different objectives. Several *a posteriori* multiobjective methods build on the concept of "Pareto dominance": program A is said to Pareto dominate program B if A is at least as good as B on all objectives and better on at least one. In GP, Pareto-based systems have often been used with "size" as one objective and

[5]Some of the text in this section is adapted from a journal article submission co-authored with Lee Spector [39].

"fitness" as another, where fitness is calculated using a standard method for combining individual test case errors into a scalar fitness value. Examples of Pareto-based approaches in GP include using information about Pareto dominance directly in the parent selection algorithm [18] or as the foundation for an approach that produces not a single solution but rather an archive of programs along the "Pareto front," where none of the programs are Pareto dominated by others [118, 60].

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 one example, he uses Pareto tournaments for parent selection while evolving queues [66]. This problem uses six objectives, five of which are based on the performance of the individual, and the sixth minimizes memory use. He has similarly used Pareto tournaments when evolving a list data structure [67]. 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 [42].

Modern multiobjective genetic algorithms such as NSGA-II [17] and SPEA2 [141] have been shown to perform well on genetic algorithms problems with small numbers of objectives. Additionally, SPEA2 has been used in GP for reducing code bloat by treating size as one objective and fitness as a second [11]. Kotanchek, Smits, and Vladislavleva have used other Pareto-based methods to reduce code bloat and search for an ensemble of solutions that make trade-offs between size and fitness [118, 60]. But, as far as we know, these modern multiobjective approaches have not been applied to GP problems where the test cases are treated as objectives. This may be attributed to assumptions that must be made about the objectives in many multiobjective methods that don't hold for the GP test cases—in particular, the large number of test cases used for many problems. Multiobjective algorithms are typically tested on problems with very few objectives; often two objectives are used, and rarely

more than four or five. GP 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 [141, 118]. This "curse of dimensionality" must be overcome to apply multiobjective algorithms to GP problems with many test cases.

Fieldsend and Moraglio use multi-objectivization of total fitness into separate test cases to determine which individuals cannot be replaced by a new child in steady-state GP [20]. Although designed for replacement selection as opposed to parent selection, this method has some similar motivations to lexicase selection. In particular, programs that perform poorly on some test cases may still survive if they solve test cases that other programs do not. This paper postulates that this method helps increase population diversity, though they do not give diversity measurements.

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* [23] 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 GP, researchers have used both a syntactic distance of programs themselves and a semantic distance based on the outputs of the programs. Fitness sharing using semantic 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 [81].

Implicit fitness sharing (IFS), first described in [117] and adapted for GP in [86], aims to preserve population diversity by distributing 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 weighted scalar fitnesses, with a tournament then used to select parents. The implicit fitness sharing fitness function is defined as

$$f_{IFS}(i) = \sum_{t \in T_i} \frac{1}{n(t)} \tag{2.1}$$

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 only needs to be computed 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 [64]. Here, the raw error value $f(t, i)$ of individual i on test case t falls in the range $[0, 1]$ with 0 being worst and 1 being best. This version of IFS can be used on problems with error values in $[0, \infty)$ by normalizing them to $[0, 1]$ and subtracting the normalized error from 1. Implicit fitness sharing is then redefined as

$$f_{NBIFS}(i) = \sum_{t \in T} \frac{f(t, i)}{\sum_{i' \in P} f(t, i')} \tag{2.2}$$

This non-binary implicit fitness sharing still scales errors based on the errors of the rest of the population, and even reduces to traditional implicit fitness sharing when errors 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 the error value on each test case is scaled by the success rate of the population [58].

Co-solvability fitness extends implicit fitness sharing to consider pairs of test cases instead of single test cases [62]. This method emphasizes solving subsets of the test cases, similarly to lexicase selection. 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 $O(PT^2)$.

In discovery of objectives by clustering (DOC) [63, 80], test cases are clustered by X-Means into groups based on population performance. DOC then considers each cluster as an objective, with error for the objective calculated by summing errors on test cases within the objective. Then, either a multi-objective optimizer (NSGA-II) or a fitness based on the hypervolume product of the objective errors is used on the new objectives to select parents. DOC outperformed IFS and tournament selection on a range of binary-outcome problems, though in a subsequent study was outperformed by lexicase selection [80].

In order to move semantic-based GP approaches away from genetic operators (such as geometric semantic GP [93]) and into parent selection, Galván-López et al. use *semantics in selection* to ensure that the two parents in a crossover operator have different semantics [22]. In particular, they use tournament selection to select one

parent based on total error, and then use a second tournament based on total error to select the second parent, in which any individual that has equivalent semantics to the first parent is automatically rejected. This method experimentally achieves similar performance to a crossover-based semantics technique, while requiring fewer program evaluations, since the crossover-based technique must test children to see whether they have equivalent semantics to the parents, which may require many iterations.

CHAPTER 3

GENERAL PROGRAM SYNTHESIS BENCHMARKS

In this chapter[1] we present a suite of 29 general program synthesis benchmark problems, systematically selected from sources of introductory computer science programming problems—Section 3.2 presents our selection criteria. This suite is designed to fill the void of general-purpose programming problems in GP and replace the domain-specific problems used for benchmarking in other example-based program synthesis fields. We describe each problem in natural language in Section 3.3 and present each problem's specifications in the form of input/output examples in Section 3.4, making them suitable to a wide range of program synthesis techniques. While the problems are not particularly challenging for skilled human programmers, they are reasonably challenging for beginners and many are arguably too difficult for existing program synthesis systems, including GP. As textbook problems, they are not likely representative of real general program synthesis applications, yet they should prove useful for assessing progress toward this goal.

Beyond describing the 29 problems and the general problem specification, we also discuss the system-specific implementation decisions that must be made and provide details of our reference implementation in the PushGP system in Section 3.5. Aside from the problems themselves, we include recommendations for the performance measures and statistical procedures required to use them experimentally in Section 3.1.

---

[1]Much of the work and writing in this chapter was developed in submissions to *GECCO 2014* [36] and *GECCO 2015* [38] (the latter of which was expanded into a technical report [37]), both co-authored with Lee Spector, as well as a journal article submission co-authored with Lee Spector and James Matheson [39].

3.1 Benchmark-Based Comparisons

This section describes the experimental and statistical procedures we recommend for general program synthesis benchmarking. It covers the methods we use to measure performance of a program synthesis system, our recommendations for how to limit the computation used by a system during benchmarking, and statistical procedures for comparing results. These recommendations constitute a significant portion of the contribution of this benchmark suite to GP.

3.1.1 Performance Measures

The primary goal of a benchmark problem is to compare the characteristics, in particular performance, of different methods or algorithms on problems with traits similar to actual applications of those methods. Here we explore options for measuring the performance of GP systems on general program synthesis benchmark problems.

General program synthesis problems fall into the category of "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 [39]. More formally, consider a problem defined by the set of test cases T where the set of programs in the search space is $\mathcal{P}$ and $p_j(t_i)$ is the error produced by program $p_j \in \mathcal{P}$ on test case $t_i \in T$ with lower error being better. This problem is *uncompromising* if a program $p \in \mathcal{P}$ would be considered a successful program to the problem if and only if $p(t_i) \leq p_j(t_i)$ for all $t_i \in T$ and $p_j \in \mathcal{P}$.

The most frequently used performance metrics in GP papers include success rate, computational effort, and mean best fitness. *Success rate* measures the percent of runs performed that find a successful program (sometimes called an "ideal solution" or "correct program"). *Computational effort* extends success rate to estimate the

number of program evaluations necessary to find a successful program with high probability [61]. The *best fitness* of a run is simply the best fitness achieved by a program during the run; the *mean best fitness* of a set of runs is the mean of the best fitnesses found in the runs.

Recent discussions of benchmarks in GP have criticized the use of success rate and computational effort, instead recommending either mean best fitness or testing with a withheld generalization data set [85, 135]. The main criticisms cited are that success rate and computational effort "measure how well a method solves trivial problems" and that they have "poor accuracy and statistical invalidity" [85]. 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 [82]. We have no quarrel with Luke and Panait on this point *in the context of such problems*. But for many uncompromising problems, including general program synthesis problems, programs that do not pass all tests do not count as solutions. Therefore, we are not interested in programs that are only approximately correct, as might be appropriate in the context of other problems for which GP is used, such as symbolic regression and classification.

The success rate of a set of GP runs gives more information than best fitness about how well the system performs on uncompromising problems, since it measures how often the system finds perfect programs. We do agree with White et al. [135] that programs that achieve perfection on the training set should be checked for overfitting by testing on withheld generalization data, which we call the *unseen test set*. A program should only count as a successful program if it achieves zero error on both the training set and the unseen test set, ensuring that it does not simply memorize the training set. While we agree that computational effort seems to have statistical problems, statistics for success rates are relatively straight forward.

We argue more generally that the ability to solve a traditional programming benchmark problem perfectly does not indicate the triviality of the problem. If our goal is to someday have GP evolve complex software that performs important functionality without having to be coded by humans, we would hope that such a system would pass all of its tests before going into use[2]. The purpose of program synthesis benchmark problems is to compare the performances of GP methods on program synthesis problems; as such, we would expect GP to be able to solve them in order to have a hope of solving real program synthesis problems.

3.1.2 Computational Budget

When comparing success rates of different GP methods, we advise taking care to ensure each method receives similar amounts of computation. Using CPU time or wall time as a proxy for computational requirements has many flaws, including differing greatly for the same computation based on the machine used, the machine's load, and the optimization of the GP system. Instead, since program evaluations often dominate the runtime of GP algorithms, most comparisons in the literature use a unit related to program evaluation to measure the computation used by the system. The computational effort measurement takes into account program evaluations by including population size and generation of found solutions in its calculation, but we would like to avoid using computational effort since it may have statistical issues. The approach that we prefer prescribes a budget to limit the computation of the GP system. Options for computation budgets offered by the community survey in [135] include, in increasing level of granularity: number of generations, number of program fitness evaluations, number of program executions (i.e. number of times any program

[2]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 for the most important subset of the tests in all systems.

is executed), and number of point evaluations (i.e. number of instructions or literals executed).

Using the number of generations as the computational budget is equivalent to using the number of program fitness evaluations if the population size is also prescribed, with the drawback that it does not allow for varying population sizes. Prescribing the number of program fitness evaluations provides a good level of control, assuming that each GP system tests every test case during every program evaluation. But, some recent GP techniques require a small subset of the test cases be used during each program evaluation [109, 30]. These techniques would be at a disadvantage if they must adhere to numbers of program evaluations equivalent to techniques that use every test case during evaluation. In fact, both of these papers report results relative to the number of program executions in order to make the comparisons fair. Limiting the number of point evaluations seems too fine of a measurement that could vary largely based on the GP system and how high-level the language is in which programs evolve. This method may be more accurate than using number of program executions when using a single GP system with the same instruction set, but likely includes too much variation when comparing different GP systems.

After considering all of the options, we recommend limiting computation with a budget based on the maximum number of program executions allowed in a run. This method allows for flexibility in many areas of algorithm design while ensuring systems receive similar computation. That said, it may nonetheless be difficult to justify fine-grained numerical comparisons between different systems that may involve qualitatively different kinds of costs and each may be parameterized in radically different ways. In many cases, the most interesting question to ask vis-a-vis a particular system on a particular problem may just be whether the system can solve the problem at all, and if so, whether it can solve it reasonably reliably. Nevertheless, we aim here

to describe specifications that will allow for as much cross-system comparability as possible.

If we want to compare results with non-GP program synthesis techniques, we must look for other performance measures since many other techniques are not based on generating and testing full programs. Outside of GP, other program synthesis research primarily uses execution time to measure performance of synthesis systems. Often these papers set a execution time limit, and report runs as failed if they do not find perfect solution programs within that limit. Many systems, especially those that are deterministic, use a single run of the system to collect results. As we discuss above, there are issues with using execution times in performance measures. But, when comparing GP to systems not based on program evaluations, this may be the best option.

3.1.3 Statistical Procedure

In order to determine the statistical significance and reliability of the differences in success rates between two sets of GP runs, we recommend the use of both the chi-square test and confidence intervals on the difference in success rates.

The chi-square test can be used to test the null hypothesis that there is no association between the success rates of two methods. We will reject the null hypothesis if the p-value is less than 0.05. We use the R implementation of the chi-square test (`pairwise.prop.test`) with Holm correction when more than two systems are compared on the same problem [108]. Since the chi-squared test is inaccurate if the number of successes or failures is near zero, we instead use the similar but more accurate Fisher's exact test with Holm correction if any number of successes or failures is below 5 [96].

There has recently been increasing criticism of null hypothesis significance testing in the sciences, for example [15], which instead recommends the use of confidence

intervals to indicate the reliability and precision of results. To supplement null hypothesis significance testing, we recommend reporting the difference in success rates along with a confidence interval of the difference. We will calculate confidence intervals using the R implementation in the `prop.test` function [108].

3.2 Problem Selection Criteria

Here we describe the criteria we used for selecting problems for the benchmark suite. Several of our criteria overlap with those described in the GP benchmarks papers [85, 135], such as selecting problems that are varied, relevant, realistically difficult, representation-independent, and precisely defined.

This benchmark suite is designed for systems that use example inputs and their corresponding outputs as the specifications for the desired programs. Thus, a problem must be defined on a range of inputs that have known correct outputs; it cannot simply specify the calculation of a single value. For example, a problem that requires the program to calculate the number of prime numbers less than 1000 would not qualify, since a correct program takes no inputs and always returns the same value; but, a similar problem that requires the program to calculate the number of prime numbers less than an input integer n would meet this requirement, since we could then provide example inputs for n and their corresponding outputs. This requirement also ensures that we can generated examples to fill the training set and unseen test set.

The breadth of the problems in the suite should present challenges typical of real programming tasks. This criterion leads us to choose problems that call for a range of programming constructs and data types. The problems should require a variety of sizes and shapes for solution programs, not just artificially small programs.

The benchmark suite should not be biased toward a particular method of synthesis; it should be possible to attempt to solve them using various GP systems as well as analytic and search-based program synthesis systems. Since systems generate

programs in a variety of languages, we avoid problems that require a specific language feature or non-standard data type (such as Java objects).

We take our problems from pre-existing sources of introductory programming problems. From each source, we include all problems that meet the criteria described above, aiming to avoid biasing the selection of problems. We rejected problems from other sources that did not meet our criteria, such as the inductive programming benchmark repository[3], other program synthesis and inductive programming papers, and programming competitions.

3.3 Problem Descriptions

We selected problems from two sources: iJava [91, 90], an interactive textbook for introductory computer science, and IntroClass [75, 14, 116], a set of problems originally designed as benchmarks for automatic program repair. Below we describe each of these sources in further detail and present our natural language description of each problem, paraphrased from the original source. All problems use functional arguments as inputs besides one that requires reading input from a file. Some problems require programs to return functional outputs, where others require the program to print results.

3.3.1 iJava

iJava is an interactive introductory computer science textbook that contains a number of automatically graded programming problems [91, 90]. Many of its problems are graded by testing programs against a range of inputs, making them suitable for automatic generation of input/output examples.

When systematically searching through problems in iJava, we found some groups of problems that meet our criteria but test similar programming techniques; for each of

[3] `http://www.inductive-programming.org/repository.html`

these groups, we chose one representative problem, ensuring a reasonable distribution of problem requirements. Along with each problem name and description, we provide the question or project number associated with the problem in iJava 3.1.

1. **Number IO (Q 3.5.1)** Given an integer and a float, print their sum.

2. **Small or Large (Q 4.6.3)** Given an integer n, print "small" if $n < 1000$ and "large" if $n \geq 2000$ (and nothing if $1000 \leq n < 2000$).

3. **For Loop Index (Q 4.11.7)** Given 3 integer inputs $start, end$, and $step$, print the integers in the sequence

$$n_0 = start$$

$$n_i = n_{i-1} + step$$

   for each $n_i < end$, each on their own line.

4. **Compare String Lengths (Q 4.11.13)** Given three strings $n1$, $n2$, and $n3$, return true if $length(n1) < length(n2) < length(n3)$, and false otherwise.

5. **Double Letters (P 4.1)** Given a string, print the string, doubling every letter character, and tripling every exclamation point. All other non-alphabetic and non-exclamation characters should be printed a single time each.

6. **Collatz Numbers (P 4.2)** Given a positive integer, find the number of terms in the Collatz (hailstone) sequence starting from that integer.

7. **Replace Space with Newline (P 4.3)** Given a string input, print the string, replacing spaces with newlines. Also, return the integer count of the non-whitespace characters. The input string will not have tabs or newlines.

8. **String Differences (P 4.4)** Given 2 strings (without whitespace) as input, find the indices at which the strings have different characters, stopping at the end of the shorter one. For each such index, print a line containing the index as well as the character in each string. For example, if the strings are "dealer" and "dollars", the program should print:

   ```
   1 e o
   2 a l
   4 e a
   ```

9. **Even Squares (Q 5.4.1)** Given an integer n, print all of the positive even perfect squares less than n on separate lines.

10. **Wallis Pi (P 6.4))** John Wallis gave the following infinite product that converges to $\pi/4$:

    $$\frac{2}{3} \times \frac{4}{3} \times \frac{4}{5} \times \frac{6}{5} \times \frac{6}{7} \times \frac{8}{7} \times \frac{8}{9} \times \frac{10}{9} \times ...$$

    Given an integer input n, compute an approximation of this product out to n terms. Results are rounded to 5 decimal places.

11. **String Lengths Backwards (Q 7.2.5)** Given a vector of strings, print the length of each string in the vector starting with the last and ending with the first, each on a separate line.

12. **Last Index of Zero (Q 7.7.8)** Given a vector of integers, at least one of which is 0, return the index of the last occurrence of 0 in the vector.

13. **Vector Average (Q 7.7.11)** Given a vector of floats, return the average of those floats. Results are rounded to 4 decimal places.

14. **Count Odds (Q 7.7.12)** Given a vector of integers, return the number of integers that are odd, without use of a specific `even` or `odd` instruction (but allowing instructions such as `mod` and `quotient`).

15. **Mirror Image (Q 7.7.15)** Given two vectors of integers, return `true` if one vector is the reverse of the other, and `false` otherwise.

16. **Super Anagrams (P 7.3)** Given strings x and y of lowercase letters, return true if y is a super anagram of x, which is the case if every character in x is in y (and false otherwise). To be true, y may contain extra characters, but must have at least as many copies of each character as x does.

17. **Sum of Squares (Q 8.5.4)** Given integer n, return the sum of squaring each integer in the range $[1, n]$.

18. **Vectors Summed (Q 8.7.6)** Given two equal-sized vectors of integers, return a vector of integers that contains the sum of the input vectors at each index.

19. **X-Word Lines (P 8.1)** Given an integer X and a string that can contain spaces and newlines, print the string with exactly X words per line. The last line may have fewer than X words.

20. **Pig Latin (P 8.2)** Given a string containing lowercase words separated by single spaces, print the string with each word translated to pig Latin. Specifically, if a word starts with a vowel, it should have "ay" added to its end; otherwise, the first letter is moved to the end of the word, followed by "ay".

21. **Negative To Zero (Q 9.6.8)** Given a vector of integers, return the vector where all negative integers have been replaced by 0.

22. **Scrabble Score (P 10.1)** Given a string of visible ASCII characters, return the Scrabble score for that string. Each letter has a corresponding value according to normal Scrabble rules, and non-letter characters are worth zero.

23. **Word Stats (P 10.5)** Given a file, print the number of words containing n characters for n from 1 to the length of the longest word, in the format:

```
words of length 1:  12
words of length 2:  3
words of length 3:  0
words of length 4:  5
...
```

At the end of the output, print a line that gives the number of sentences and a line that gives the average sentence length using the form:

```
number of sentences:  4
average sentence length:  7.452423455
```

A word is any string of consecutive non-whitespace characters (including sentence terminators). Every file will contain at least one sentence terminator (period, exclamation point, or question mark). The average sentence length is the number of words in the file divided by the number of sentence terminator characters.

3.3.2 IntroClass

The set of 6 problems in the IntroClass dataset [75, 14, 116] was designed for the purpose of benchmarking automatic software defect repair systems. As such, the authors of this dataset provide a number of buggy programs written by students trying to solve each problem, taken from students in an introductory computer science class. For the purposes of general program synthesis from scratch, we will use the problems themselves but not the accompanying buggy programs.

24. **Checksum** Given a string, convert each character in the string into its integer ASCII value, sum them, take the sum modulo 64, add the integer value of the space character, and then convert that integer back into its corresponding character (the checksum character). The program must print `Check sum is X`, where X is replaced by the correct checksum character.

25. **Digits** Given an integer, print that integer's digits each on their own line starting with the least significant digit. A negative integer should have the negative sign printed before the most significant digit.

26. **Grade** Given 5 integers, the first four represent the lower numeric thresholds for achieving the grades A, B, C, and D, and will be distinct and in descending order. The fifth represents the student's numeric grade. The program must print `Student has a X grade.`, where X is A, B, C, D, or F depending on the thresholds and the numeric grade.

27. **Median** Given 3 integers, print their median.

28. **Smallest** Given 4 integers, print the smallest of them.

29. **Syllables** Given a string containing symbols, spaces, digits, and lowercase letters, count the number of occurrences of vowels (a, e, i, o, u, y) in the string and print that number as X in `The number of syllables is X`.

3.4 Synthesis Specifications

The natural language descriptions of the problems in Section 3.3 do not provide all of the information needed to apply program synthesis systems to the problems. Here we provide additional specifications, aiming to do so in a technique-independent and system-independent way.

Tables 3.1 and 3.2 present recommendations regarding training and test data for each problem. While these are merely guidelines, and there may be good reasons to diverge from them when using different techniques or systems, adhering to these guidelines will clarify comparisons among techniques and systems. These tables describe the data types of the inputs and outputs and give reasonable ranges for program inputs.

Table 3.1. For each problem, we give the types of the input and output examples, and the limits imposed on the inputs. Any printed outputs should be printed by the program to standard output. The columns Train and Test indicate the recommended number of input/output examples in the training set and unseen test set respectively.

Name	Inputs	Outputs	Train	Test
Number IO	integer in $[-100, 100]$, float in $[-100.0, 100.0]$	printed float	25	1000
Small Or Large	integer in $[-10000, 10000]$	printed string	100	1000
For Loop Index	integers `start` and `end` in $[-500, 500]$, `step` in $[1, 10]$	printed integers	100	1000
Compare String Lengths	3 strings of length $[0, 49]$	boolean	100	1000
Double Letters	string of length $[0, 20]$	printed string	100	1000
Collatz Numbers	integer in $[1, 10000]$	integer	200	2000
Replace Space with Newline	string of length $[0, 20]$	printed string, integer	100	1000
String Differences	2 strings of length $[0, 10]$	printed string	200	2000
Even Squares	integer in $[1, 9999]$	printed string	100	1000
Wallis Pi	integer in $[1, 200]$	float	150	50
String Lengths Backwards	vector of length $[0, 50]$ of strings of length $[0, 50]$	printed string	100	1000
Last Index of Zero	vector of integers of length $[1, 50]$ with each integer in $[-50, 50]$	integer	150	1000
Vector Average	vector of floats of length $[1, 50]$ with each float in $[-1000.0, 1000.0]$	float	100	1000
Count Odds	vector of integers of length $[0, 50]$ with each integer in $[-1000, 1000]$	integer	200	2000
Mirror Image	2 vectors of integers of length $[0, 50]$ with each integer in $[-1000, 1000]$	boolean	100	1000
Super Anagrams	2 strings of length $[0, 20]$	boolean	200	2000
Sum of Squares	integer in $[1, 100]$	integer	50	50
Vectors Summed	2 vectors of integers of length $[0, 50]$ with each integer in $[-1000, 1000]$	vector of integers	150	1500

Table 3.2. Continuation of Table 3.1.

Name	Inputs	Outputs	Train	Test
X-Word Lines	integer in $[1, 10]$, string of length $[0, 100]$	printed string	150	2000
Pig Latin	string of length $[0, 50]$	printed string	200	1000
Negative To Zero	vector of integers of length $[0, 50]$ with each integer in $[-1000, 1000]$	vector of integers	200	2000
Scrabble Score	string of length $[0, 20]$	integer	200	1000
Word Stats	file containing $[1, 100]$ chars	printed string	100	1000
Checksum	string of length $[0, 50]$	printed string	100	1000
Digits	integer in $[-9999999999, 9999999999]$	printed integers	100	1000
Grade	5 integers in $[0, 100]$	printed string	200	2000
Median	3 integers in $[-100, 100]$	printed integer	100	1000
Smallest	4 integers in $[-100, 100]$	printed integer	100	1000
Syllables	string of length $[0, 20]$	printed string	100	1000

We also provide recommendations for numbers of examples to use in the training and unseen test sets in Tables 3.1 and 3.2[4]. For most problems, we recommend between 100 and 200 examples in the training set, depending on the difficulty of the problem as well as the dimensionality of the input space. A few problems use fewer examples, either because they have limited input spaces or are simple enough to solve with fewer cases. We usually recommend using a unseen test set ten times as large as the training set; again, there are exceptions for problems with limited input spaces. The method of producing the examples is system-specific; we recommend a combination of hand-chosen edge cases with randomly generated examples, and will describe our method in more detail in Section 3.5.

The question of which instructions to make available for a synthesis system to use for each problem is a complex one. It is important to not cherry pick a small set of instructions that are known to be sufficient to solve a problem; such a selection may be difficult for a real-world problem, where it might not be clear which instructions will be useful. On the other hand, using all available instructions for every problem expands the search space and may make problems more difficult than necessary. We recommend a compromise between these approaches in which one first determines which data types are likely to be useful for solving the problem and then uses all instructions that operate on those data types. For example, an instruction that compares the equality of two integers and returns a boolean would be included if the problem could potentially make use of integers and booleans. By specifying only the data type requirements for a problem, we can limit the number of instructions without cherry picking.

[4]Note that problems using multiple error functions will have multiple test cases per example.

3.5 System-Specific Parameters

Whereas Section 3.4 gave technique-independent recommendations for specifying the benchmark problems for a synthesis system, this section will give more detail about the system-specific parameters and decisions that must be made in order to implement these problems in a given program synthesis system. Here we will focus on our implementation in the PushGP genetic programming system, but we emphasize that this is just one possible approach and implementation, and that the problems here could be used in any system that meets the requirements in Section 3.2.

PushGP evolves programs in Push, a stack-based programming language designed specifically for GP (see Section 2.1.3.1 for more details). We provide a PushGP reference implementation[5] for others to use and to guide other implementations of these problems. In the rest of this section, we will describe some of the major decisions necessary for implementing these benchmark problems in PushGP, many of which are relevant to other implementations.

3.5.1 Generation of Example Data

It is important to experimentally test a synthesis system, regardless of whether it is deterministic or stochastic, on different sets of examples for a single problem to ensure that its measured performance is not tied to a particular set of examples. We would like to create the labeled examples for the training and unseen test sets automatically so that we do not have to create hundreds of such sets by hand. While the synthesis system itself will only use a set of examples as specifications, example generation itself requires more detailed problem specifications.

In order to facilitate the creation of training and test data, we designed a general system for automatic example generation based on data domains [36]. A *data domain* D is a set of program inputs described by either a list of inputs or a random input

[5] `http://thelmuth.github.io/GECCO_2015_Benchmarks_Materials/`

generator function. The list (`"hi"`, `"hello"`, `"howdy"`, `"hey"`) and a function that returns `"zoo"` followed by 0 to 17 random lowercase letters are examples of data domains, where the former is an enumerated list of four inputs and the latter is random input generator function of strings at most 20 characters long that start with the substring `"zoo"`. Along with each data domain D, the user must provide the integers $train(D)$ and $test(D)$ that indicate the number of training and test examples respectively to generate from D.

To generate training and test data from a set of data domains $\{D_1, D_2, ..., D_n\}$, we simply take each domain and create the required number of examples. If the domain D_i is an enumerated list of inputs, we select $train(D_i)$ and $test(D_i)$ of them at random, without replacement within the training examples or test examples. If the domain is described by a random input generator, we run it $train(D_i)$ and $test(D_i)$ times (with replacement) to create the data. This automatic data generation system allows for the generation of training and test examples for a wide range of problems.

The freedom to specify hand-chosen examples as well as generate random examples gives us flexibility to define a wide range of problems. For each problem we include a small set of hard coded examples that cover edge cases on which synthesized programs may otherwise fail. These handcrafted examples are added to a larger set of randomly generated examples designed to cover the remaining space of possible inputs. This combination ensures that the training and unseen test sets provide sufficient coverage of difficult edge cases while also allowing us to use a variety of training sets.

Tables 3.3 through 3.10 present detailed descriptions of the data domains we used to generate training and test examples for each benchmark problem. These tables have two types of data domains: hard coded lists of inputs (HC) and random input generators (RNGs). For HC data domains, we give the list of inputs; for RNGs, we describe the generator. Ranges for inputs are given in Table 3.1. For integer and float RNGs, inputs are sampled uniformly across the given range; for string RNGs,

lengths are sampled uniformly between 1 and the max length given in Table 3.1, and characters are distributed uniformly across visible ASCII characters along with space, newline, and tab. If a HC domain is specified by a range such as [40, 50], it includes every integer in the range inclusive. For HC string inputs, we use `"␣"` for the space character, `"\t"` for tab, and `"\n"` for newline.

3.5.2 Error Functions

In software testing or test-driven development, each test usually only tells whether a program passes or fails. On the other hand, in GP it is common (though not strictly necessary) to also provide an *error function* that not only tests whether a program passes or fails each test case, but also gives an estimate of how badly it failed in the form of an *error value*. Error values create a richer search space with more informative search gradients compared to binary pass/fail tests. Requiring a user to provide a problem-specific error function does add additional burden on the user beyond simply providing input/output examples. As such, we have used generic error functions based simply on output type for most of our benchmark problems; these standard error functions can easily be defined for their output types. Only when necessary do we provide problem-specific error functions.

Many of the problems in this suite print results to standard output; our generic error function for these problems treats the printed outputs as strings and uses Levenshtein distance (a measure of string edit distance). Other problems produce numeric outputs, either returned or printed; for these problems we use absolute error for our generic error function, parsing printed numbers when possible. Some problems produce boolean values, or are best measured by a simple binary right or wrong; here, we use an error of 0 for right and 1 for wrong. Finally, some problems require problem-tailored error functions, such as vector edit distance or string formatting re-

Table 3.3. Data domains for each benchmark problem (part 1).

Name	Type	Domain	Train	Test
Number IO	RNG	integer, float	25	1000
Small Or Large	HC	-10000, 0, 980, 1020, 1980, 2020, 10000, [995, 1004], [1995, 2004]	27	0
	HC	integers in ranges [980, 1019] and [1980, 2019]	0	80
	RNG	integer	73	920
For Loop Index	RNG	integers: `start` $<$ 0 $<$ `end`, `start` + (20 $\times$ `step`) + 1 $>$ `end`	10	100
	RNG	integers: `start` $<$ `end`, `start` + (20 $\times$ `step`) + 1 $>$ `end`	90	900
Compare String Lengths	HC	triplet `("", "", "")`	1	0
	HC	all permutations of `("", "a", "bc")`	6	0
	RNG	(repeated twice) all permutations of 2 empty strings and a string	6	0
	RNG	(repeated 3 times) all permutations of 2 copies of a string and another string	9	0
	RNG	random string repeated 3 times	3	100
	RNG	3 strings in sorted length order	25	200
	RNG	3 strings	50	700
Double Letters	HC	`"", "A", "!", "␣", "*", "\t", "\n", "B\n", "\n\n", "CD", "ef", "!!", "q!", "!R", "!#", "@!", "!F!", "T$L", "4ps", "q\t ", "!!!", "i:!i:!i:!i:!i", "88888888888888888", "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣", "sssssssssssssssssss", "!!!!!!!!!!!!!!!!!!!", "Ha␣Ha␣Ha␣Ha␣Ha␣Ha␣Ha", "x\ny!x\ny!x\ny!x\ny!x\ny!", "1!1!1!1!1!1!1!1!1!1!1!", "G5G5G5G5G5G5G5G5G5G5", ">_=]>_=]>_=]>_=]>_=]", "k!!k!!k!!k!!k!!k!!k!"`	32	0
	RNG	string	68	1000
Collatz Numbers	HC	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 6171, 6943, 7963, 9257, 9999, 10000	16	0
	RNG	integer	184	2000

Table 3.4. Data domains for each benchmark problem (part 2).

Name	Type	Domain	Train	Test
Replace Space With Newline	HC	`"", "A", "*", "␣", "s", "B␣", "␣␣", "␣D", "ef", "!!", "␣F␣", "T␣L", "4ps", "q␣␣", "␣␣␣", "␣␣e", "hi␣", "␣␣$␣␣", "␣␣␣␣␣9", "i␣!i␣!i␣!i␣!i", "88888888888888888", "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣", "sssssssssssssssssss", "1␣1␣1␣1␣1␣1␣1␣1␣1␣", "␣v␣v␣v␣v␣v␣v␣v␣v␣v␣v", "Ha␣Ha␣Ha␣Ha␣Ha␣Ha␣Ha", "x␣y!x␣y!x␣y!x␣y!x␣y!", "G5G5G5G5G5G5G5G5G5G5", ">_=]>_=]>_=]>_=]>_=]", "^_^␣^_^␣^_^␣^_^␣^_^␣"`	30	0
	RNG	string (with ␣ as 20% of characters)	70	1000
String Differences	HC	pairs of strings: `("" ""), ("" "hi"), ("ThereWorld" ""), ("A" "A"), ("B" "C"), ("&" "#"), ("4" "456789"), ("rat" "hat"), ("new" "net"), ("big" "bag"), ("STOP" "SIGN"), ("abcde" "a"), ("abcde" "abcde"), ("abcde" "edcba"), ("2n" "nn"), ("hi" "zipper"), ("dealer" "dollars"), ("nacho" "cheese"), ("loud" "louder"), ("qwertyuiop" "asdfghjkl;"), ("LALALALALA" "LLLLLLLLLL"), ("!!!!!!" ".?."), ("9r2334" "9223d4r"), ("WellWell" "wellwell"), ("TakeThat!" "TAKETHAT!!"), ("CHOCOLATE^" "CHOCOLATE^"), ("sssssssssss" "~~~~~~~~~~"), (">_=]>_=]>_" "q_q_q_q_q_"), ("()()()()()" "pp)pp)pp)p"), ("HaHaHaHaHa" "HiHiHiHiHi")`	30	0
	RNG	pair of strings, length > 1	170	0
	RNG	pair of strings	0	2000

Table 3.5. Data domains for each benchmark problem (part 3).

Name	Type	Domain	Train	Test
Even Squares	HC	1, 2, 3, 4, 5, 6, 15, 16, 17, 18, 36, 37, 64, 65, 9600, 9700, 9999	17	0
	RNG	integer	83	1000
Wallis Pi	HC	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 198, 199, 200	15	0
	RNG	integer	135	50
String Lengths Backwards	HC	vector of strings: `[]`, `[""]`, `["" ""]`, `["" "" ""]`, `["" "" "" "" "" "" "" "" "" ""]`, `["abcde"]`, `["1"]`, `["abc" "hi there"]`, `["!@#" "\n\n\t\t" "5552\na r"]`, `["tt" "333" "1" "ccc"]`	10	0
	RNG	vector of strings	90	1000
Last Index of Zero	HC	vector of integers: `[0 1]`, `[1 0]`, `[7 0]`, `[0 8]`, `[0 -1]`, `[-1 0]`, `[-7 0]`, `[0 -8]`	8	0
	HC	every vector of zeros of length between 1 and 50	30	20
	HC	all permutations of vector `[0 5 -8 9]`	20	4
	HC	all permutations of vector `[0 0 -8 9]`	10	2
	HC	all permutations of vector `[0 0 0 9]`	4	0
	RNG	vector of integers with at least one 0	78	974
Vector Average	HC	vector of floats: `[0.0]`, `[100.0]`, `[-100.0]`, `[2.0 129.0]`, `[0.12345 -4.678]`, `[999.99 74.113]`	6	0
	RNG	length 50 vector of floats	4	50
	RNG	vector of floats	90	950
Count Odds	HC	vector of integers: `[]`, `[-10]`, `[-9]`, `[-8]`, `[-7]`, `[-6]`, `[-5]`, `[-4]`, `[-3]`, `[-2]`, `[-1]`, `[0]`, `[1]`, `[2]`, `[3]`, `[4]`, `[5]`, `[6]`, `[7]`, `[8]`, `[9]`, `[10]`, `[-947]`, `[-450]`, `[303]`, `[886]`, `[0 0]`, `[0 1]`, `[7 1]`, `[-9 -1]`, `[-11 40]`, `[944 77]`	32	0
	RNG	vector of integers, all odd	9	100
	RNG	vector of integers, all even	9	100
	RNG	vector of integers, random probability of odd per vector	150	1800

Table 3.6. Data domains for each benchmark problem (part 4).

Name	Type	Domain	Train	Test
Mirror Image	HC	pair of vectors of integers: `([] [])`, `([1] [1])`, `([0] [1])`, `([1] [0])`, `([-44] [16])`, `([-13] [-12])`, `([2 1] [1 2])`, `([0 1] [1 1])`, `([0 7] [7 0])`, `([5 8] [5 8])`, `([34 12] [34 12])`, `([456 456] [456 456])`, `([40 831] [-431 -680])`, `([1 2 1] [1 2 1])`, `([1 2 3 4 5 4 3 2 1] [1 2 3 4 5 4 3 2 1])`, `([45 99 0 12 44 7 7 44 12 0 99 45] [45 99 0 12 44 7 7 44 12 0 99 45])`, `([24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24] [24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24])`, `([33 45 -941] [33 45 -941])`, `([33 -941 45] [33 45 -941])`, `([45 33 -941] [33 45 -941])`, `([45 -941 33] [33 45 -941])`, `([-941 33 45] [33 45 -941])`, `([-941 45 33] [33 45 -941])`	23	0
	RNG	pair of vectors of integers that are mirror image	37	500
	RNG	pair of equal vectors of integers	10	100
	RNG	pair of vectors of integers that are close to mirror image, but have a few elements changed	20	200
	RNG	pair of vectors of integers	10	200

Table 3.7. Data domains for each benchmark problem (part 5).

Name	Type	Domain	Train	Test
Super Anagrams	HC	pair of strings: `("" ""), ("h" ""), ("" "i"), ("a" "a"), ("c" "b"), ("nn" "n"), ("c" "abcde"), ("abcde" "c"), ("mnbvccxz" "r"), ("aabc" "abc"), ("abcde" "aabc"), ("edcba" "abcde"), ("moo" "mo"), ("mo" "moo"), ("though" "tree"), ("zipper" "rip"), ("rip" "flipper"), ("zipper" "hi"), ("dollars" "dealer"), ("louder" "loud"), ("ccccc" "ccccccccc"), ("oldwestaction" "clinteastwood"), ("ldwestaction" "clinteastwood"), ("verificationcomplete" "verificationcomplete"), ("hhhhhhhhhhaaaaaaaaaa" "hahahahahahahahahaha"), ("aahhhh" "hahahahahahahahahaha"), ("qwqeqrqtqyquqiqoqpqs" ""), ("qazwsxedcrfvtgbyhnuj" "wxyz"), ("gggffggfefeededdd" "dddeeefffgggg"), ("dddeeefffgggg" "gggffggfefeededdd")`	30	0
	RNG	pair of strings, chosen to be close to (or actually) super anagrams	170	2000
Sum of Squares	HC	1, 2, 3, 4, 5, 100	6	0
	RNG	integer	44	50
Vectors Summed	HC	pair of vectors of integers: `([] []), ([0] [0]), ([10] [0]), ([5] [3]), ([-9] [7]), ([0 0] [0 0]), ([-4 2] [0 1]), ([-3 0] [-1 0]), ([-323 49] [-90 -6])`	10	0
	RNG	pair of length 1 vectors of integers	5	0
	RNG	pair of length 50 vectors of integers	10	100
	RNG	pairs of vectors of integers	125	1400
X-Word Lines	HC	pair of strings and integers (too long to print, see reference implementation for details)	46	0
	RNG	pair of strings and integers	104	2000

Table 3.8. Data domains for each benchmark problem (part 6).

Name	Type	Domain	Train	Test
Pig Latin	HC	`""`, `"a"`, `"b"`, `"c"`, `"d"`, `"e"`, `"i"`, `"m"`, `"o"`, `"u"`, `"y"`, `"z"`, `"hello"`, `"there"`, `"world"`, `"eat"`, `"apple"`, `"yellow"`, `"orange"`, `"umbrella"`, `"ouch"`, `"in"`, `"hello␣there␣world"`, `"out␣at␣the␣plate"`, `"nap␣time␣on␣planets"`, `"supercalifragilistic"`, `"expialidocious"`, `"u"` repeated for 50 characters, `"s"` repeated for 50 characters, `"w␣"` repeated for 50 characters, `"e␣"` repeated for 50 characters, `"ha␣"` repeated for 50 characters, `"x␣y"` repeated for 50 characters	33	0
	RNG	string	167	1000
Negative To Zero	HC	vector of integers: `[]`, `[-10]`, `[-1]`, `[0]`, `[1]`, `[10]`, `[0 0]`, `[0 1]`, `[-1 0]`, `[-90 -6]`, `[-16 33]`, `[412 111]`	12	0
	RNG	length 1 vector of integers	5	0
	RNG	vector of negative integers	9	100
	RNG	vector of positive integers	9	100
	RNG	vector of integers	165	1800
Scrabble Score	HC	each single lowercase letter	26	0
	HC	each single uppercase letter	0	26
	HC	`""`, `"*"`, `"␣"`, `"Q␣"`, `"zx"`, `"␣Dw"`, `"ef"`, `"!!"`, `"␣F@"`, `"ydp"`, `"4ps"`, `"abcdefghijklmnopqrst"`, `"ghijklmnopqrstuvwxyz"`, `"zxyzxyqQQZXYqqjjawp"`, `"h␣w␣h␣j##r##r\␣n+JJL"`, `"i␣!i␣!i␣!i␣!i"`, `"QQQQQQQQQQQQQQQQQQQQ"`, `"$$$$$$$$$$$$$$$$$$$$"`, `"wwwwwwwwwwwwwwwwwwww"`, `"1␣1␣1␣1␣1␣1␣1␣1␣1␣1␣"`, `"␣v␣v␣v␣v␣v␣v␣v␣v␣v␣v"`, `"Ha␣Ha␣Ha␣Ha␣Ha␣Ha␣Ha"`, `"x␣y!x␣y!x␣y!x␣y!x␣y!"`, `"G5G5G5G5G5G5G5G5G5G5"`	24	0
	RNG	string with at least 2 characters	150	974

Table 3.9. Data domains for each benchmark problem (part 7).

Name	Type	Domain	Train	Test
Word Stats	HC	string (too long to print, see reference implementation for details)	36	0
	RNG	string containing at least one sentence terminator	64	1000
Checksum	HC	string (too long to print, see reference implementation for details)	12	0
	RNG	string	88	1000
Digits	HC	-9495969798, -20008000, -777777, -9876, -482, -97, -20, 0, 19, 620, 24068, 512000, 8313227, 30000000, 9998887776	15	0
	RNG	integer taken from logarithmic distribution	85	1000
Grade	HC	quintuplet of integers: `(80 70 60 50 85), (80 70 60 50 80), (80 70 60 50 79), (80 70 60 50 75), (80 70 60 50 70), (80 70 60 50 69), (80 70 60 50 65), (80 70 60 50 60), (80 70 60 50 59), (80 70 60 50 55), (80 70 60 50 50), (80 70 60 50 49), (80 70 60 50 45), (90 80 70 60 100), (90 80 70 60 0), (4 3 2 1 5), (4 3 2 1 4), (4 3 2 1 3), (4 3 2 1 2), (4 3 2 1 1), (4 3 2 1 0), (100 99 98 97 100), (100 99 98 97 99), (100 99 98 97 98), (100 99 98 97 97), (100 99 98 97 96), (98 48 27 3 55), (98 48 27 3 14), (98 48 27 3 1), (45 30 27 0 1), (45 30 27 0 0), (48 46 44 42 40), (48 46 44 42 41), (48 46 44 42 42), (48 46 44 42 43), (48 46 44 42 44), (48 46 44 42 45), (48 46 44 42 46), (48 46 44 42 47), (48 46 44 42 48), (48 46 44 42 49)`	41	0
	RNG	quintuplet of integers, with the first four distinct and decreasing	159	2000
Median	RNG	triplet of integers, all equal	10	100
	RNG	triplet of integers, two of three equal	30	300
	RNG	triplet of integers	60	600

Table 3.10. Data domains for each benchmark problem (part 8).

Name	Type	Domain	Train	Test
Smallest	HC	quadruplet of integers: `(0 0 0 0), (-44 -44 -7 -13), (0 4 -99 -33), (-22 -22 -22 -22), (99 100 99 100)`	5	0
	RNG	quadruplet of integers, all equal	5	100
	RNG	quadruplet of integers, three of four equal	10	100
	RNG	quadruplet of integers in range [0, 100]	20	200
	RNG	quadruplet of integers	60	600
Syllables	HC	`"", "a", "v", "4", "o", "␣", "aei", "ouy", "chf", "quite", "a␣r␣e9j>", "you␣are␣many␣yay␣yea", "sssssssssssssssssss", "ooooooooooooooooooo", "wi␣wi␣wi␣wi␣wi␣wi", "x␣y␣x␣y␣x␣y␣x␣y␣x␣y␣", "eioyeioyeioyeioyeioy"`	17	0
	RNG	string (with each char having 20% chance of being a vowel)	83	1000

quirements. We give the details of the error functions for each problem in Tables 3.11 and 3.12.

In PushGP, it is conventional to take functional return values from the tops of stacks after a program finishes executing (see Section 2.1.3.1). But, some evolved programs finish executing with an empty return stack, resulting in no value to return. In these cases we give a penalty error value. We choose problem-specific penalties that should be larger than any reasonable error for an actual return value.

For some problems we found it appropriate to use multiple error functions per input/output example. For example, the Replace Space With Newline problem requires both a printed string and a returned integer. For problems like this, we produce multiple error values for a single example. Additionally, we find that PushGP performs better on some problems when we use more than one error function per example, even where not strictly necessary. For example, we found no solutions to the X-Word Lines problem when using Levenshtein distance as the only error function, but found

Table 3.11. The error functions used for each problem. For problems that require the program to print, we usually use Levenshtein distance on the printed string and the correct output. Additionally, we add a second error function to many problems by parsing part or all of a printed string as a different data type and comparing to the correct output. For example, for the Number IO problem, if the printed output can be parsed as a float, it is done so and used as a float error. For such problems, an output that cannot be parsed correctly receives a penalty error. Continued in Table 3.12.

Problem	**Error Function**
Number IO	printed string Levenshtein distance; printed float error
Small Or Large	printed string Levenshtein distance
For Loop Index	printed string Levenshtein distance
Compare String Lengths	boolean error
Double Letters	printed string Levenshtein distance
Collatz Numbers	integer error
Replace Space with Newline	printed string Levenshtein distance; integer error
String Differences	printed string Levenshtein distance; numeric difference in number of lines with correct format
Even Squares	printed string Levenshtein distance; numeric difference in number of lines with correct format; printed integer error on each line
Wallis Pi	float error; Levenshtein distance of string version of float
String Lengths Backwards	printed string Levenshtein distance
Last Index of Zero	integer error
Vector Average	float error
Count Odds	integer error
Mirror Image	boolean error
Super Anagrams	boolean error
Sum of Squares	integer error
Vectors Summed	integer error at each position in vector
X-Word Lines	printed string Levenshtein distance; integer error for number of newlines; numeric difference in correct words on each line summed over lines
Pig Latin	printed string Levenshtein distance
Negative To Zero	integer vector Levenshtein distance
Scrabble Score	integer error
Word Stats	printed string Levenshtein distance; integer error for printed number of sentences; float error for printed average sentence length
Checksum	printed string Levenshtein distance; for last printed char in string, ASCII value error

Table 3.12. Error functions, continued from Table 3.11.

Problem	Error Function
Digits	printed string Levenshtein distance
Grade	printed string Levenshtein distance; printed char error for grade char
Median	printed string right/wrong
Smallest	printed string right/wrong
Syllables	printed string Levenshtein distance; printed integer error

solutions after adding additional error functions calculating the number of newline characters and summed errors of differences in number of words on each line. When using multiple error functions for a single input/output example, we say that there are multiple test cases per example. We can then treat each test case separately when the parent selection method requires it.

3.5.3 Instruction Sets

As discussed in Section 3.4, we have chosen to specify the data types relevant to each problem and then include all instructions that use those data types in each problem's instruction set. Table 3.13 presents the Push data types we chose for each problem. The column "# Instructions" reports the number of instructions, terminals, and ERCs used for each problem. The remaining columns show which data types were used for each problem. For example, the Number IO problem used all instructions relevant to integers, floats, and printing. The "exec" column signifies instructions that use Push's `exec` stack, which typically perform control flow manipulations such as conditionals, iteration, and subfunctions defined through tagging [127]. The "print" column includes instructions that print data to standard output, and "file input" includes a small set of file reading instructions. The "Problems" row simply counts how many problems use each data type. The "Instructions" row shows the number of Push instructions that primarily use each data type; some use multiple types but are only counted once.

Table 3.13. Instructions and data types used in our PushGP implementation of each problem. See text for details.

Problem	# Instructions	exec	integer	float	boolean	char	string	vector of integers	vector of floats	vector of strings	print	file input
Number IO	50		x	x							x	
Small Or Large	103	x	x		x		x				x	
For Loop Index	74	x	x		x						x	
Compare String Lengths	98	x	x		x		x					
Double Letters	132	x	x		x	x	x				x	
Collatz Numbers	102	x	x	x	x							
Replace Space with Newline	135	x	x		x	x	x				x	
String Differences	135	x	x		x	x	x				x	
Even Squares	72	x	x		x						x	
Wallis Pi	103	x	x	x	x							
String Lengths Backwards	134	x	x		x		x			x	x	
Last Index of Zero	101	x	x		x			x				
Vector Average	88	x	x	x					x			
Count Odds	104	x	x		x			x				
Mirror Image	102	x	x		x			x				
Super Anagrams	129	x	x		x	x	x					
Sum of Squares	71	x	x		x							
Vectors Summed	68	x	x					x				
X-Word Lines	134	x	x		x	x	x				x	
Pig Latin	141	x	x		x	x	x				x	
Negative To Zero	102	x	x		x			x				
Scrabble Score	158	x	x		x	x	x	x				
Word Stats	281	x	x	x	x	x	x	x	x	x	x	x
Checksum	136	x	x		x	x	x				x	
Digits	133	x	x		x	x	x				x	
Grade	112	x	x		x		x				x	
Median	75	x	x		x						x	
Smallest	76	x	x		x						x	
Syllables	141	x	x		x	x	x				x	
Problems		28	29	5	26	11	15	7	2	2	17	1
Instructions		28	28	31	19	17	39	31	31	31	10	4

Table 3.14. The terminals (constants and ERCs) used for the problem; ERC ranges are given in Table 3.15. Here, char constants are represented in the Clojure style, starting with a backslash, and strings are surrounded by double quotation marks.

Problem	**Terminals (besides inputs)**
Number IO	integer ERC, float ERC
Small Or Large	“small”, “large”, integer ERC
For Loop Index	
Compare String Lengths	boolean ERC
Double Letters	\!
Collatz Numbers	0, 1, integer ERC
Replace Space with Newline	\space, \newline, string ERC, char ERC
String Differences	\space, \newline, integer ERC
Even Squares	
Wallis Pi	2 integer ERCs, 2 float ERCs
String Lengths Backwards	integer ERC
Last Index of Zero	0
Vector Average	
Count Odds	0, 1, 2, integer ERC
Mirror Image	boolean ERC
Super Anagrams	boolean ERC, char ERC, integer ERC
Sum of Squares	0, 1, integer ERC
Vectors Summed	[], integer ERC
X-Word Lines	\newline, \space
Pig Latin	“ay”, \space, \a, \e, \i, \o, \u, “aeiou”, string ERC, char ERC
Negative To Zero	0, []
Scrabble Score	vector containing Scrabble values (indexed by ASCII values)
Word Stats	\., \?, \!, \space, \tab, \newline, [], “words of length ”, “: ”, “number of sentences: ”, “average sentence length: ”, integer ERC
Checksum	“Check sum is ”, \space, 64, integer ERC, char ERC
Digits	\newline, integer ERC [-10, 10]
Grade	“Student has a ”, “ grade.”, “A”, “B”, “C”, “D”, “F”, integer ERC
Median	integer ERC
Smallest	integer ERC
Syllables	“The number of syllables is ”, “aeiouy”, \a, \e, \i, \o, \u, \y, char ERC, string ERC

Table 3.15. ERC ranges used in our problems. For char and string ERCs, "visible chars" indicates all visible ASCII characters plus space, newline, and tab.

Problem	**ERC Ranges**
Number IO	integer ERC $[-100, 100]$, float ERC $[-100.0, 100.0)$
Small Or Large	integer ERC $[-10000, 10000]$
For Loop Index	
Compare String Lengths	boolean ERC $[true, false]$
Double Letters	
Collatz Numbers	integer ERC $[-100, 100]$
Replace Space with Newline	char ERC (visible chars), string ERC (lowercase letters and spaces, with space having 20% chance at each character)
String Differences	integer ERC $[-10, 10]$
Even Squares	
Wallis Pi	integer ERC $[-10, 10]$, integer ERC $[-500, 500]$, float ERC $[-500.0, 500.0)$
String Lengths Backwards	integer ERC $[-100, 100]$
Last Index of Zero	integer ERC $[-50, 50]$
Vector Average	
Count Odds	integer ERC $[-1000, 1000]$
Mirror Image	boolean ERC $[true, false]$
Super Anagrams	boolean ERC $[true, false]$, integer ERC $[-1000, 1000]$, char ERC (visible chars)
Sum of Squares	integer ERC $[-100, 100]$
Vectors Summed	integer ERC $[-1000, 1000]$
X-Word Lines	
Pig Latin	char ERC (visible chars), string ERC (lowercase letters and spaces, with space having 20% chance at each character)
Negative To Zero	
Scrabble Score	
Word Stats	integer ERC $[-100, 100]$
Checksum	integer ERC $[-128, 128]$, char ERC (visible chars)
Digits	integer ERC $[-10, 10]$
Grade	integer ERC $[0, 100]$
Median	integer ERC $[-100, 100]$
Smallest	integer ERC $[-100, 100]$
Syllables	char ERC (visible chars), string ERC (lowercase letters, spaces, digits, and symbols, with vowels having 20% chance at each character)

Table 3.16. Push data types and instructions used in our experiments. For each combination of data types listed in the first column, we list all of the Push instructions that are included in the instruction set when those data types are present for the problem. Continued in Tables 3.17 and 3.18.

Data Types	Instructions
boolean	boolean_empty, boolean_swap, boolean_eq, boolean_invert_first_then_and, boolean_flush, boolean_rot, boolean_and, boolean_invert_second_then_and, boolean_xor, boolean_not, boolean_or, boolean_dup, boolean_pop
boolean, char	char_iswhitespace, char_empty, char_isletter, char_eq, char_isdigit
boolean, char, string	string_containschar
boolean, exec	exec_eq, exec_when, exec_if, exec_do*while, exec_while, exec_empty
boolean, float	float_lt, boolean_fromfloat, float_empty, float_lte, float_gte, float_fromboolean, float_gt, float_eq
boolean, float, vector_float	vector_float_contains
boolean, integer	integer_eq, boolean_yank, integer_gte, integer_lt, integer_lte, boolean_shove, integer_empty, integer_gt, integer_fromboolean, boolean_frominteger, boolean_stackdepth, boolean_yankdup
boolean, integer, vector_integer	vector_integer_contains
boolean, string	string_eq, string_emptystring, string_fromboolean, string_contains, string_empty
boolean, string, vector_string	vector_string_contains
boolean, vector_float	vector_float_emptyvector, vector_float_empty, vector_float_eq
boolean, vector_integer	vector_integer_eq, vector_integer_empty, vector_integer_emptyvector
boolean, vector_string	vector_string_empty, vector_string_emptyvector, vector_string_eq
char	char_dup, char_swap, char_flush, char_rot, char_pop
char, exec, string	exec_string_iterate
char, float	char_fromfloat, float_fromchar
char, integer	char_shove, char_stackdepth, integer_fromchar, char_yank, char_yankdup, char_frominteger
char, integer, string	string_occurrencesofchar, string_setchar, string_nth, string_indexofchar

Table 3.17. Push data types and instructions (part 2).

Data Types	Instructions
char, string	string_removechar, char_allfromstring, string_replacefirstchar, string_replacechar, string_conjchar, string_fromchar, string_first, string_last
exec	exec_y, exec_pop, exec_rot, exec_s, exec_k, exec_flush, exec_swap, exec_dup, exec_noop, tag, tagged
exec, float, vector_float	exec_do*vector_float
exec, integer	exec_stackdepth, exec_do*times, exec_do*count, exec_do*range, exec_yank, exec_yankdup, exec_shove
exec, integer, vector_integer	exec_do*vector_integer
exec, string, vector_string	exec_do*vector_string
file	file_readline, file_readchar, file_EOF, file_begin
float	float_rot, float_sin, float_cos, float_swap, float_div, float_inc, float_sub, float_flush, float_add, float_tan, float_mult, float_max, float_pop, float_min, float_dup, float_dec, float_mod
float, integer	float_yank, float_frominteger, float_stackdepth, float_shove, float_yankdup, integer_fromfloat
float, integer, vector_float	vector_float_indexof, vector_float_occurrencesof, vector_float_nth, vector_float_set
float, string	float_fromstring, string_fromfloat
float, vector_float	vector_float_conj, vector_float_remove, vector_float_last, vector_float_first, vector_float_replacefirst, vector_float_pushall, vector_float_replace
integer	integer_add, integer_swap, integer_yank, integer_dup, integer_yankdup, integer_flush, integer_shove, integer_mult, integer_stackdepth, integer_div, integer_inc, integer_max, integer_sub, integer_mod, integer_rot, integer_dec, integer_min, integer_pop
integer, string	string_substring, string_take, string_frominteger, string_stackdepth, integer_fromstring, string_yank, string_yankdup, string_length, string_shove
integer, string, vector_string	vector_string_indexof, vector_string_set, vector_string_nth, vector_string_occurrencesof
integer, vector_float	vector_float_shove, vector_float_length, vector_float_stackdepth, vector_float_subvec, vector_float_yank, vector_float_take, vector_float_yankdup

Table 3.18. Push data types and instructions (part 3).

Data Types	Instructions
integer, vector_integer	vector_integer_remove, vector_integer_pushall, vector_integer_yank, vector_integer_subvec, vector_integer_last, vector_integer_first, vector_integer_shove, vector_integer_indexof, vector_integer_occurrencesof, vector_integer_replace, vector_integer_replacefirst, vector_integer_take, vector_integer_stackdepth, vector_integer_nth, vector_integer_set, vector_integer_length, vector_integer_yankdup, vector_integer_conj
integer, vector_string	vector_string_stackdepth, vector_string_subvec, vector_string_take, vector_string_shove, vector_string_yank, vector_string_length, vector_string_yankdup
print	print_newline
print, boolean	print_boolean
print, char	print_char
print, exec	print_exec
print, float	print_float
print, integer	print_integer
print, string	print_string
print, vector_float	print_vector_float
print, vector_integer	print_vector_integer
print, vector_string	print_vector_string
string	string_pop, string_rot, string_rest, string_parse_to_chars, string_reverse, string_swap, string_split, string_flush, string_replacefirst, string_butlast, string_concat, string_replace, string_dup
string, vector_string	vector_string_remove, vector_string_conj, vector_string_first, vector_string_pushall, vector_string_last, vector_string_replacefirst, vector_string_replace
vector_float	vector_float_dup, vector_float_pop, vector_float_rot, vector_float_swap, vector_float_flush, vector_float_reverse, vector_float_rest, vector_float_concat, vector_float_butlast
vector_integer	vector_integer_swap, vector_integer_butlast, vector_integer_flush, vector_integer_rest, vector_integer_concat, vector_integer_rot, vector_integer_reverse, vector_integer_pop, vector_integer_dup
vector_string	vector_string_dup, vector_string_rot, vector_string_rest, vector_string_reverse, vector_string_butlast, vector_string_concat, vector_string_pop, vector_string_flush, vector_string_swap

Table 3.14 gives the terminals used for each problem, which encompass constants and ephemeral random constants (ERCs). ERCs allow for the creation of random constants in randomly generated code during initialization and mutation. We used problem-specific ERC ranges, which can be found in Table 3.15. These ranges were selected as seemed appropriate for each problem; we do not expect that changing these ranges would have significant impact on results.

Tables 3.16, 3.17, and 3.18 show every Push instruction used in our experiments and the data types that they require. For example, the `string_containschar` instruction requires that the boolean, char, and string data types be used for a problem in order to be included; this is because it must use a string and a char as inputs, and returns a boolean of whether the input string contains the input char. These tables are intended to give an idea of the scope and complexity of the instructions used in our experiments. Attempting the problems in another system would obviously require a different set of instructions specific to the programming language of the search. While we would expect such a system to use different instructions, we would also expect similar numbers of instructions that are not cherry-picked for the individual problems.

3.5.4 PushGP Parameters

In the most recent version of PushGP, genomes are represented by flat sequences of instructions that may have one or more epigenetic markers attached to each instruction (see Section 2.1.3.1). In this work, we use the default epigenetic markers, which only include a marker that tells how many pairs of parentheses to close after each instruction when translating the genome into a Push program. We initialize genomes by selecting a genome size uniformly between 0 and the maximum initial genome size, which for these runs we set to half of the maximum genome size. Each

Table 3.19. The PushGP parameters that were held constant across the problems. See Section 2.1.3.1 for more information about these parameters. The alignment deviation was set to 5 for four problems: Number IO, Small Or Large, Median, and Smallest.

Parameter	**Value**
population size	1000
alternation rate	0.01
alignment deviation	10
uniform mutation rate	0.01
uniform close mutation rate	0.1
Genetic Operator	**Prob**
alternation	0.2
uniform mutation	0.2
uniform close mutation	0.1
alternation followed by uniform mutation	0.5

gene is composed of an instruction taken uniformly from the instruction set, as well as an epigenetic marker for parentheses ranging from 0 to 3, weighted toward 0.

In our experiments, we keep most of our PushGP system parameters constant across all problems, with specific details in Table 3.19. The genetic operators in our system work on the linear Push genomes as described in Section 2.1.3.1. The only significant PushGP parameters that we vary per problem are the maximum program size, the maximum number of instruction evaluations that a program may use per execution, and the maximum number of generations per run. We varied these parameters based on expected problem difficulty and expected program size necessary to solve each problem; the exact values are given in Table 3.20. By specifying the maximum generations, the population size (1000 for all of our runs), and the size of the training set (see Table 3.1), we also specify the program execution budget, which is the product of those values.

Table 3.20. The PushGP parameters that we varied per problem. "Max Genome Size" gives the maximum number of instructions that can appear in an individual's genome. "Eval Limit" is the number of steps of the Push interpreter that are executed before stopping a program's execution; programs halted in this way may still achieve good results if they print or leave results on the appropriate stack(s) before they are stopped. "Max Gens" gives the maximum number of generations in a single PushGP run. "Program Execution Budget" is the maximum number of programs that will be executed before a run is terminated, which is the product of the maximum generations, the population size, and the size of the training set.

Problem	**Max Genome Size**	**Eval Limit**	**Max Gens**	**Program Execution Budget**
Number IO	200	200	200	5,000,000
Small Or Large	200	300	300	30,000,000
For Loop Index	300	600	300	30,000,000
Compare String Lengths	400	600	300	30,000,000
Double Letters	800	1600	300	30,000,000
Collatz Numbers	600	15000	300	60,000,000
Replace Space with Newline	800	1600	300	30,000,000
String Differences	1000	2000	300	60,000,000
Even Squares	400	2000	300	30,000,000
Wallis Pi	600	8000	300	45,000,000
String Lengths Backwards	300	600	300	30,000,000
Last Index of Zero	300	600	300	45,000,000
Vector Average	400	800	300	30,000,000
Count Odds	500	1500	300	60,000,000
Mirror Image	300	600	300	30,000,000
Super Anagrams	800	1600	300	60,000,000
Sum of Squares	400	4000	300	15,000,000
Vectors Summed	500	1500	300	45,000,000
X-Word Lines	800	1600	300	45,000,000
Pig Latin	1000	2000	300	60,000,000
Negative To Zero	500	1500	300	60,000,000
Scrabble Score	1000	2000	300	60,000,000
Word Stats	1000	6000	300	30,000,000
Checksum	800	1500	300	30,000,000
Digits	300	600	300	30,000,000
Grade	400	800	300	60,000,000
Median	200	200	200	20,000,000
Smallest	200	200	200	20,000,000
Syllables	800	1600	300	30,000,000

CHAPTER 4

LEXICASE SELECTION

In a population-based stochastic search algorithm such as GP, lexicase[1] selection provides a method for selecting individuals to serve as parents of new individuals. As a behavior-based search driver [65], it can be used any time that potential parents are assessed with respect to multiple test cases. We give details of the lexicase selection algorithm in Section 4.1.

We have previously shown that lexicase selection can effectively increase performance while also increasing behavioral diversity on a variety of GP problems [39, 36, 80, 35]. In Section 4.2, we compare the performance of GP using lexicase selection to two other common parent selection methods, tournament selection and IFS, on the general program synthesis benchmark problems given in Chapter 3. Our results again show marked performance gains by lexicase selection.

In the remainder of the chapter, we explore the properties of lexicase selection to gather insight into why it performs well compared to other methods. Looking at the details of a single run of lexicase in Section 4.3 motivated research questions that we explore further in Sections 4.4 and 4.5. In particular, we examine how lexicase selection allows GP to explore the search space of programs while concentrating effort on promising programs. These experiments help explain the performance improvements we see when using lexicase selection.

---

[1]The term "lexicase" has been used previously in unrelated work [130, 129].

To select a single parent program for use in a genetic operation:

1. `Initialize`:
   (a) Set `candidates` to be the entire population of programs.
   (b) Set `cases` to be a list of all of the test cases in the training set 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`.

Figure 4.1. Pseudocode for the lexicase selection algorithm.

4.1 Lexicase Selection Algorithm[2]

While variations of the lexicase selection algorithm have been discussed and tested, the primary version of the algorithm explored in this work is described in pseudocode in Figure 4.1; in the original publication of lexicase selection, this was called "global pool, uniform random sequence, elitist lexicase parent selection" [123]. In each parent selection event, the lexicase selection algorithm first randomly orders the test cases from the training set. 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 that do not have the

[2]Much of the text in this section is adapted from a journal article submission co-authored with Lee Spector and James Matheson [39].

[3]One variation of lexicase that has been suggested is that the retention of only "the best" could be relaxed to retain all individuals within some distance of the best, but the form of lexicase selection here is "elitist" in that it retains only the best. This idea seems particularly appealing for problems that produce floating point errors, such as symbolic regression.

best performance of the remaining individuals 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.

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, leading to better running times. 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. In practice, we have found that lexicase is about 2 to 10 times slower per generation than tournament selection [39].

Lexicase selection sometimes selects individuals that perform well on a relatively small number of test cases, even if they perform very poorly on other cases. This differs from most other selection algorithms, which select individuals based on aggregations of performance on all test cases into a single scalar fitness value. As such, lexicase often selects *specialist* individuals that solve parts of the problem extremely well, as opposed to tournament selection and IFS, which select *generalist* individuals that have good performance on average across the test cases. Although these individuals may have worse summed error across all test cases, the hope is they will be able to reproduce 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.

4.2 Performance Results[4]

A primary goal of this work is to empirically compare the performance of GP with lexicase selection to other selection methods on the general program synthesis benchmark problems described in Chapter 3. Here we present results using four different parent selection methods: lexicase selection, baseline uniform selection, tournament selection, and implicit fitness sharing. These experiments use 100 runs of PushGP per selection method (using different random seeds), with other system parameters given in Section 3.5.4.

Our first comparison method, *uniform selection*, provides a simple baseline parent selection operator[5]. Uniform selection selects parents at random without bias from the population. This approach does not take into account performance on the training set when selecting parents; as such, it conducts a random walk using the same variation operators as with other techniques without providing selection pressure. The purpose of this baseline is to explore whether or not parent selection plays any role in GP performance, or whether the other components of GP are sufficient.

We also compare lexicase selection to parent selection methods that aggregate errors on test cases into a single scalar fitness value per individual. Most common parent selection methods, including tournament selection and implicit fitness sharing, fall into this category. Implicit fitness sharing (IFS), an extension of tournament selection, makes an especially interesting comparison to lexicase selection, since it was designed specifically with the goal of increasing population diversity [86, 117] (see

[4]Much of the text in this section is adapted from a technical report co-authored with Lee Spector [37].

[5]Due to a bug, an instruction equivalent to `exec_noop` was included in the instruction set for the runs using uniform selection. We believe that this has a negligible to zero effect on our results. This instruction does nothing when evaluated, simply leaving the stack states unchanged. The instruction `exec_noop` already appeared in each instruction set once, so this simply added a second equivalent instruction. Additionally, the instruction sets we used always have 50 or more instructions, and usually more than 100 (see Table 3.13). Such large instruction sets dilute the importance of a single instruction, especially one that does nothing.

Section 2.2 for more details). Most of the problems here produce non-binary error values, for which we use the non-binary adaptation of IFS given in Equation 2.2. As required by this method, we normalize error values to $[0, 1]$ by dividing each error by a maximum allowed error value, which differs per problem based on the problem's error function. Tournament selection and IFS base selection on tournaments; we use a tournament size of 7 for both methods.

When using uniform selection, PushGP only found successful programs for two problems. It found 6 generalizing solutions on the Number IO problem, and 19 generalizing solutions on the Mirror Image problem. Considering uniform selection makes GP into a random walk, it is surprising that even as many as 19 solutions were found on any problem, no matter how easy. Still, it failed completely in 100 runs on the other 27 problems.

Table 4.1 gives the results of using lexicase selection, tournament selection, and implicit fitness sharing. Over the 29 problems, PushGP with lexicase selection produced at least one successful run on nine more problems than either tournament selection or IFS. Additionally, there were 8 problems where lexicase selection achieved a significantly higher number of successful runs than the other two using the chi-square test, where IFS showed significant improvement on just one problem and tournament selection none. Lexicase selection had more successes on 21 of the 29 problems. The confidence intervals of the difference in success rate between lexicase and tournament or IFS generally show positive or neutral effects of using lexicase. All three of these parent selection methods clearly outperform the baseline uniform selection method.

To examine aggregate performance of lexicase compared to the other selection methods, we examine the hypothesis that lexicase selection does not affect GP's performance compared to the other two methods. If we assume that all three methods have equal probability of achieving the best performance on a given problem, then we can use the binomial distribution with $p = \frac{1}{3}$ to calculate the probability of lexicase

Table 4.1. The first three columns give the number of successful runs out of 100 for each setting, where **L** is lexicase selection, **T** is tournament selection, and **I** is implicit fitness sharing. For each problem, underline indicates significant improvement over the other two selection methods (see Section 3.1.3). The columns **L−T** and **L−I** give the differences in success rate (successful runs divided by total runs) between lexicase and the other two settings. The columns **L−T CI** and **L−I CI** give 95% confidence intervals of the differences in success rate. Note that we omitted the 7 problems on which no solutions were found: Collatz Numbers, String Differences, Wallis Pi, Super Anagrams, Pig Latin, Word Stats, and Checksum.

Problem	**L**	**T**	**I**	**L−T**	**L−T CI**	**L−I**	**L−I CI**
Number IO	98	68	72	0.30	[0.19, 0.41]	0.26	[0.16, 0.36]
Small Or Large	5	3	3	0.02	[−0.04, 0.08]	0.02	[−0.04, 0.08]
For Loop Index	1	0	0	0.01	[−0.02, 0.04]	0.01	[−0.02, 0.04]
Compare String Lengths	7	3	6	0.04	[−0.03, 0.11]	0.01	[−0.07, 0.09]
Double Letters	6	0	0	0.06	[0.00, 0.12]	0.06	[0.00, 0.12]
Replace Space with Newline	51	8	16	0.43	[0.31, 0.55]	0.35	[0.22, 0.48]
Even Squares	2	0	0	0.02	[−0.02, 0.06]	0.02	[−0.02, 0.06]
String Lengths Backwards	66	7	10	0.59	[0.47, 0.71]	0.56	[0.44, 0.68]
Last Index of Zero	21	8	4	0.13	[0.02, 0.24]	0.17	[0.07, 0.27]
Vector Average	16	14	13	0.02	[−0.09, 0.13]	0.03	[−0.08, 0.14]
Count Odds	8	0	0	0.08	[0.02, 0.14]	0.08	[0.02, 0.14]
Mirror Image	78	46	64	0.32	[0.18, 0.46]	0.14	[0.01, 0.27]
Sum of Squares	6	2	0	0.04	[−0.02, 0.10]	0.06	[0.00, 0.12]
Vectors Summed	1	0	0	0.01	[−0.02, 0.04]	0.01	[−0.02, 0.04]
X-Word Lines	8	0	0	0.08	[0.02, 0.14]	0.08	[0.02, 0.14]
Negative To Zero	45	10	8	0.35	[0.23, 0.47]	0.37	[0.25, 0.49]
Scrabble Score	2	0	0	0.02	[−0.02, 0.06]	0.02	[−0.02, 0.06]
Digits	7	0	1	0.07	[0.01, 0.13]	0.06	[0.00, 0.12]
Grade	4	0	0	0.04	[−0.01, 0.09]	0.04	[−0.01, 0.09]
Median	45	7	43	0.38	[0.26, 0.50]	0.02	[−0.13, 0.17]
Smallest	81	75	98	0.06	[−0.06, 0.18]	−0.17	[−0.26, −0.08]
Syllables	18	1	7	0.17	[0.08, 0.26]	0.11	[0.01, 0.21]
Solved	22	13	13				

selection achieving the best performance on 21 of the 29 problems as 1.6×10^{-5}, an extremely unlikely event. On the other hand, if we assume that lexicase selection has a $\frac{2}{3}$ chance of achieving better performance than the other methods, we get a much more likely probability of 0.13. This shows that it is highly unlikely that lexicase selection achieved more successful runs on 21 problems simply by chance.

As another method of examining aggregate performance, we calculate the average rank of each method across the 29 problems, with 1 being best and 3 being worst (on tied problems, such as the 7 problems for which no method found a single solution, each method gets the average of the tied ranks):

Lexicase	IFS	Tournament
1.28	2.26	2.47

Lexicase achieves the lowest average rank, as it has the most or tied for the most successes on every problem except for one. The Friedman test on this data gives us a p-value of 3.4×10^{-8}, indicating that at least one method performs significantly differently from the others. A post-hoc Wilcoxon-Nemenyi-McDonald-Thompson test [41] indicates that lexicase significantly outranks both IFS and tournament ($p < 0.0001$). These results strongly indicate the utility of lexicase selection for general program synthesis problems.

The data in Table 4.1 only reflect solutions that generalize by achieving zero error on the unseen test set. Some problems seem to lend themselves to generalization more than others; for example, PushGP using lexicase selection found 14 programs with zero error on the training set for the Super Anagrams problem, none of which generalized to the unseen test set. On the other hand, 51 out of the 54 programs with zero training error on the Replace Space With Newline problem generalized to the unseen test set. For lexicase selection, five problems resulted in 20 or more runs that passed the training set that did not generalize (Small Or Large, Compare String Lengths, Last Index of Zero, Negative To Zero, and Median), and five problems

had between 10 and 20 runs that did not generalize (String Lengths Backwards, Mirror Image, Super Anagrams, Digits, and Smallest)[6]. These 10 problems show an important area for future study: how to evolve programs that generalize to unseen data for general program synthesis problems. Among these problems are the only five in the suite that give a correct/incorrect binary error as fitness in our implementation: Compare String Lengths, Mirror Image, Super Anagrams, Median, and Smallest. This shows the difficulty of evolving general programs based entirely on correctness of output, and suggests that these problems might be better tackled if they can be transformed into problems with more informative fitness functions.

4.2.1 Experimental Significance to Benchmark Suite

With regard to the problems themselves, this experiment illustrates the ability of this benchmark suite to provide useful comparisons between multiple systems or parameter settings. By looking at the number of problems solved by each technique, how often each technique showed significant improvements over the others, and the average rank of each technique across the problems, we can clearly see that lexicase selection increases PushGP's ability to solve general program synthesis problems compared to tournament selection and IFS. The main goal of a benchmark suite is to support this type of experiment. Additionally, some problems in the suite were solved frequently by each system, whereas others were solved infrequently or not at all. This range of difficulties permits the suite to be useful for a variety of experiments and allows it to remain relevant as program synthesis systems improve.

Of the seven problems on which PushGP found no generalizing solution, most are not surprising in that they involve extensive use of multiple programming constructs, the linking of many distinct steps, or a deceptive fitness space where fitness improve-

[6]Three of these problems come from the IntroClass set of problems. Work in the field of automatic program repair has also noted significant overfitting on these problems [116].

Table 4.2. The smallest size of any simplified solution program (in Push points, which includes instructions and nested parenthesis pairs) for each problem on which PushGP found at least one solution.

Problem	Size
Number IO	5
Small Or Large	27
For Loop Index	21
Compare String Lengths	11
Double Letters	20
Replace Space with Newline	9
Even Squares	37
String Lengths Backwards	9
Last Index of Zero	5
Vector Average	7
Count Odds	7
Mirror Image	4
Sum of Squares	7
Vectors Summed	11
X-Word Lines	15
Negative To Zero	8
Scrabble Score	14
Digits	20
Grade	52
Median	10
Smallest	8
Syllables	14

ments do not lead toward perfect programs. We have written solutions to each of the unsolved problems by hand to ensure that each problem is solvable within the constraints we put on the system and instruction set.

Table 4.2 gives the size (in Push points, which includes instructions and nested parenthesis pairs) of the smallest simplified solution program for each problem. Here, we've used post-run simplification to automatically reduce the sizes of solution programs without changing their semantics on the training data [125]. While this hill-climbing simplification is not guaranteed to find the smallest semantically equivalent program, it reliably removes excess code, leaving the core functionality of the program [125]. The simplified program sizes present a reasonable proxy for the smallest

solution program for each problem (using our instruction sets). While some problems can be solved with programs containing fewer than 10 instructions, this does not necessarily make them trivial; only two were solved using the baseline uniform selection, which essentially implements a random walk over the search space.

4.3 Anecdotal Example[7]

To exemplify the differences between lexicase selection and traditional tournament selection, here we present an anecdote from a single PushGP run using lexicase selection on the Replace Space With Newline problem. We analyzed this run using the open-source graph database Neo4j [8] to better understand the evolutionary dynamics introduced by lexicase selection; more details can be found in [88]. Our observations of this single run motivated some of the systematic experiments presented later in this chapter.

We will concentrate this analysis at the end of the run, when the GP system created multiple individuals that solved the problem. We used Neo4j to find all the ancestors of any "winning" individual, i.e., an individual with a total error of zero on all 200 test cases. Figure 4.2 shows the ancestry of all of the winners starting from generation 79 and ending in generation 87, when multiple winners were found. The numbers inside the nodes are Neo4j internal IDs; we will use these as "names" for the individuals. Each ID has two parts: the part before the colon is that individual's generation, and the part after is simply a random three digit identifier.

GP found 45 distinct winners in the final generation of this run, or 4.5% of the population of 1,000 individuals. All 45 winners had a single individual (86:261) as at least one of their parents, and 42 of them had 86:261 as their *only* parent, i.e.

---

[7]Much of the text in this section is adapted from a book chapter co-authored with Nicholas Freitag McPhee and David Donatucci [88].

[8]`http://neo4j.com/`

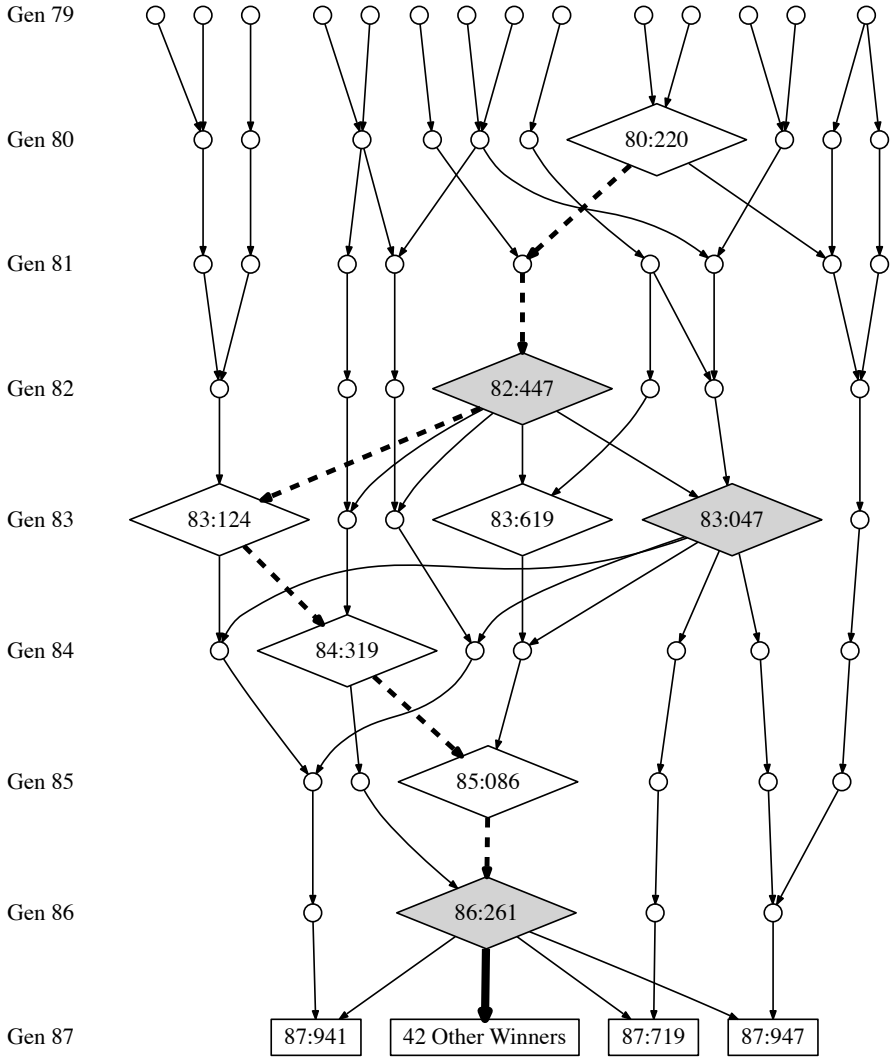

Figure 4.2. Ancestry of the 45 "winners" (individuals that achieve zero error on all test cases in the training set) from a successful run of the Replace Space With Newline problem using lexicase selection. Nodes in the graph represent individuals, and edges represent parent-child relationships, directed from parent to child. Diamond-shaped nodes had over 100 offspring each. Shaded nodes had at least five offspring that were winners or ancestors of winners.

they were mutations of 86:261 or were the result of self-crosses of 86:261. To simplify the graph, we've combined those 42 individuals into a single node labeled "42 Other Winners".

Looking at Figure 4.2 we can see that a few individuals have more offspring than others. Each diamond-shaped node in the graph represents an individual that had at least 100 children. The most remarkable is individual 86:261, which was a parent of 934 of the 1,000 individuals in generation 87, including every winner. This level of selection focus, or *hyper-selection*, would simply be impossible using almost any other common type of selection. For example, in a comparable run of tournament selection, which found a solution after 150 generations, the most prolific parent in any generation created only 24 children. For tournament selection, the maximum number of times an individual can be a parent is limited by the number of tournaments in which it participates; for tournament size 7 and a population of 1,000 individuals, an individual would only participate in about 0.7% of the tournaments on average. PushGP averages around 1,700 selections per generation[9], resulting in about 12 tournaments per individual. Thus, no matter how good an individual is, it will likely never have the opportunity to participate in enough tournaments to be parent of over 100 children, let alone 934.

If we look at the total error of the individuals in Figure 4.2, we again find some surprises that tell us quite a lot about lexicase selection. Table 4.3 presents the total error for each individual along the dashed path from 80:220 to 86:261. The first five individuals in this chain have reasonably low total error. One individual (82:447) has the best total error in its generation and all but 81:691 are in the top fifth of the population when ranked by total fitness. However, individuals 86:261 (the parent of every winner) and 85:086 (its parent and therefore the grandparent

[9]Selections per generation varies, since some genetic operators require two parents and others one, and operators are randomly selected.

Table 4.3. The total error, rank in the population (by total error; out of 1000 individuals), and number of children of the individuals along the dashed line in Figure 4.2.

Individual	Total Error	Rank in Population	Number of Children
80:220	321	147	200
81:691	441	268	17
82:447	107	1	443
83:124	157	85	170
84:319	240	188	279
85:086	100,000	971	180
86:261	4,034	765	934

of every winner) achieved considerably worse total error. Both individuals ranked in the bottom quartile of their respective generations by total error, with the 85:086 coming very near absolute worst in its population by this ranking. Yet both of these individuals had over 100 children, and as we saw previously, 86:261 was parent of over 93% of the children in the next generation.

How could individuals with such terrible total fitness end up being selected so often as parents? As a reminder, the Replace Space With Newline problem requires the program to print a string and return an integer for each input example, resulting in two different types of test cases. Exploring the specific test case errors reveals that individual 85:086 is perfect on half of the test cases (all those that involve printing), but gets a penalty error of 1,000 on the other half because it never returns an integer. None of the other programs in the population achieve zero error on all 100 of the printing test cases. So, even though it completely fails at half of the test cases, individual 85:086 was often selected by lexicase, presumably when the first few test cases in lexicase's random ordering came from the half of the cases that test for printing.

Perhaps more extremely, individual 86:261 had 934 offspring while its total error ranked in the bottom quartile of the population. This individual has zero error on

194 of the 200 test cases. On four of the remaining six test cases it fails to return a value and gets the penalty of 1,000; it has an error of 17 on the other two. When examining why 86:261 failed entirely on four test cases, we found that it is essentially a correct program, but exceeded the instruction evaluation limit on those four test cases—which were four of the longest input strings. In PushGP, when a program exceeds the execution limit, it is halted as if it terminated on its own, and return values are taken from the top of the stacks. For the two test cases on which it achieved 17 error, it also exceeded the evaluation limit, but in those cases the partial calculation left an integer on the integer stack, which was 17 away from the correct answer; for the penalized cases, the program left no output on the stack. With this in mind, this program can easily be fixed by decreasing its execution time without otherwise altering its behavior, which is likely what most or all of the 45 winners did.

The individuals 85:086 and 86:261 present extreme examples of how lexicase selection may select *specialist* individuals that perform very well on some test cases while receiving terrible error on others. Both of these individuals had total error ranking in the bottom quartiles of their populations, yet received large percentages of the parent selections in their generations. These selections were rewarded by quickly solving the problem. If we selected from these populations using tournament selection, both of these individuals would be highly unlikely to receive a single selection. Even with IFS, a parent selection technique designed to reward programs that perform well on difficult test cases, these individuals would likely not receive any selections due to their horrible performance on some test cases.

While this anecdote provides interesting observations of a single GP run using lexicase selection, it may not be representative of runs using lexicase in general. In the Sections 4.4 and 4.5 we explore some of these observations across systematic experiments to determine the important characteristics of lexicase selection.

4.4 Exploration and Exploitation

The results presented in Section 4.2 raise the question of why lexicase selection performs significantly better on general program synthesis problems than tournament selection or IFS. A large factor may be that the way lexicase selection emphasizes the selection of specialist individuals 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 help widen the evolutionary search, 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.

In our earlier work, we found that lexicase selection maintained substantially higher population diversity compared to tournament selection and IFS on a few benchmark problems, including one program synthesis problem [39]. Increasing exploration through higher diversity allows GP to locate areas of the search space with potentially useful programs and to leave local optima. In Section 4.3 we anecdotally noted high levels of exploitation when using lexicase selection in the form of individuals selected a very large percent of the time, especially compared to their rank in the population. Increased exploitation allows GP to incrementally improve good programs to refine them into better programs and find solutions.

Many search operators can increase either exploration or exploitation by sacrificing the other; for example, tournament selection can use smaller tournament sizes in order to increase exploratory diversity at the cost of reducing exploitative selection pressure, and vice versa by increasing tournament sizes. In this section we present observations that confirm lexicase selection's increased exploration (Section 4.4.1) and exploitation (Section 4.4.2) compared to tournament selection and IFS in the

context of general program synthesis problems, which help explain lexicase selection's increased performance on these problems.

4.4.1 Exploration[10]

Maintaining a diverse population allows GP to explore multiple interesting areas of the search space simultaneously, instead of limiting search to a local neighborhood. Many evolutionary techniques have been developed to try to increase diversity in the search space, including IFS and other methods discussed in Section 2.2. While these techniques have all proven useful in different settings, they all aggregate errors on test cases into a single fitness value. Lexicase selection sets itself apart by never comparing or aggregating error values from different test cases; this may allow it to explore areas of the search space unreachable using other methods that put too little pressure on selecting individuals that perform very poorly on a subset of the test cases.

Various measures of diversity have been proposed in the genetic programming literature. Because the way a genetic programming population explores the semantic space of programs, as opposed to the syntactic space, is more indicative of its ability to synthesize programs that perform different actions on a problem, we will focus on methods of measuring semantic diversity. When evaluating a program, we run it on a set of input/output examples and create a *behavior vector* of its outputs. Then, we apply one or more error functions to each of the program's outputs, comparing them to the desired output to create an *error vector*. We define *error diversity* to be the percentage of distinct error vectors in the population. Error diversity is similar to *behavioral diversity*, which is the percentage of distinct behavior vectors in the population [43]. The error diversity of a population will be less than or equal to its behavioral diversity, since two different behavior vectors may produce the same error

[10]Much of the text in this section is adapted from a book chapter co-authored with Nicholas Freitag McPhee and Lee Spector [33].

vector, but two different error vectors must come from different behavior vectors; still, we expect these two measurements to convey similar information. 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 [43]. Additionally, we have previously shown that lexicase selection maintained higher diversity than tournament and IFS selection on three problems, only one of which was a general program synthesis problem (the wc problem) [39].

To examine population diversity on general program synthesis problems, we collected data from 100 runs on each 8 of the benchmark problems, chosen to cover a range of difficulties, data types, and requirements. Since these runs were conducted after the initial experiments in Section 4.2, the numbers of successes varied slightly from those in Table 4.1. Figures 4.3–4.10 show error diversity over time for each of the problems. Below each plot is a smaller sub-plot showing the number of successes over time; since runs end when a solution is found, the successes plot gives a sense of how many runs are still being represented in the primary plot at a given generation. Additionally, since runs terminate once they find a program that passes all test cases in the training set, the success counts plotted here are on the training set, not the unseen test set.

In Figure 4.3, for example, the number of lexicase successes is nearly 25 by generation 50, and nearly 50 by generation 150. Thus there are slightly more than 75 data points still represented in the lexicase data at generation 50, but only about 50 data points represented from generations 150 to 300. Each plot includes a line indicating the median error diversity across whichever of the 100 runs is still active at that generation. We also indicate the range from the 25^{th} percentile to the 75^{th} percentile with a gray band around the median line; unfortunately the tournament and IFS results are often very similar and strongly overlap, making them difficult to differentiate.

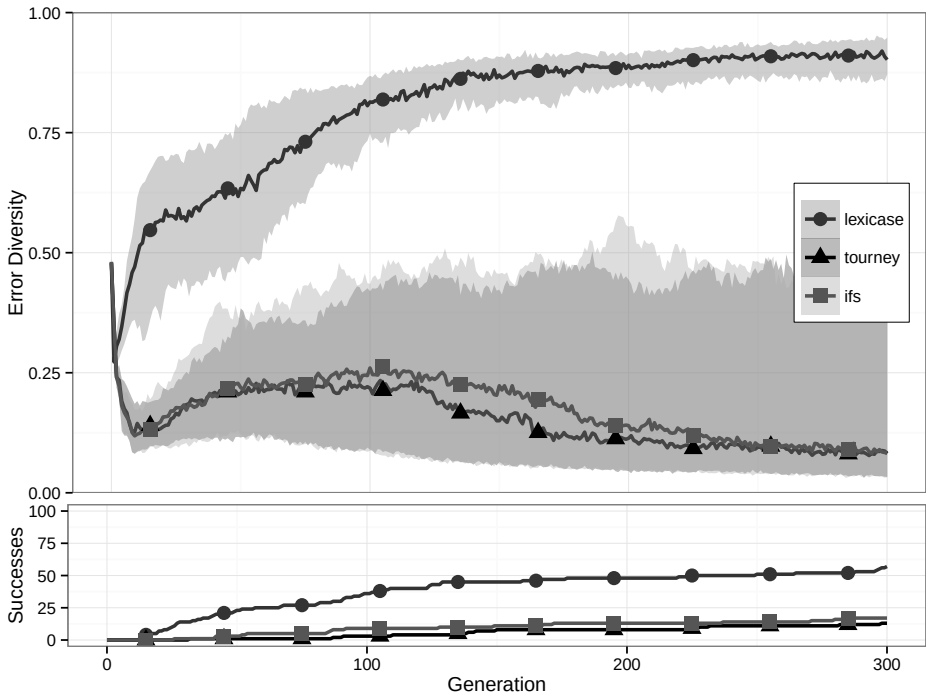

Figure 4.3. Replace Space With Newline – error diversity median (line) and quartiles (shaded)

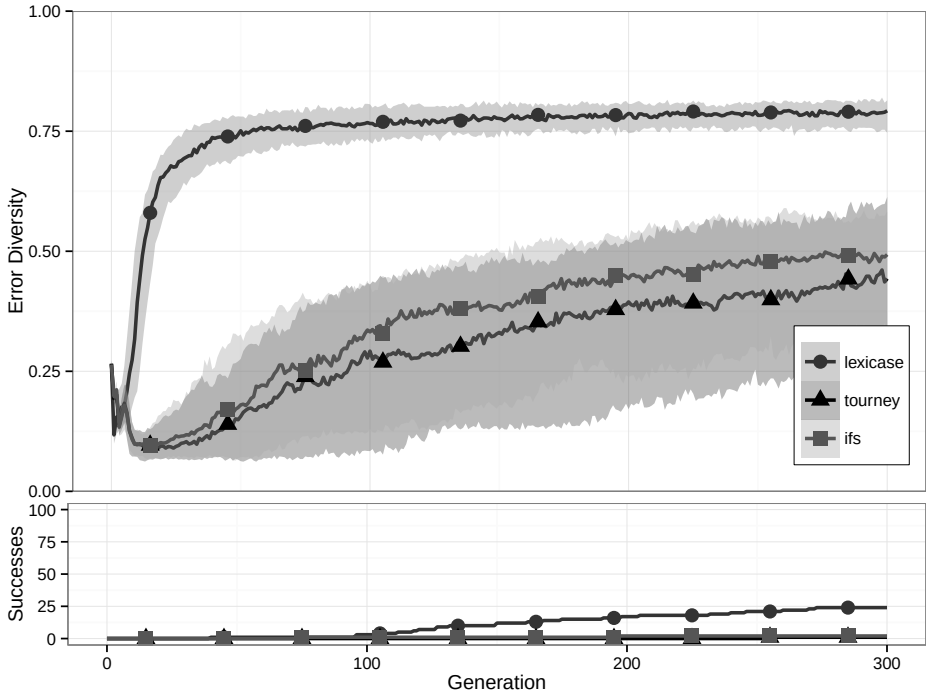

Figure 4.4. Syllables – error diversity median (line) and quartiles (shaded)

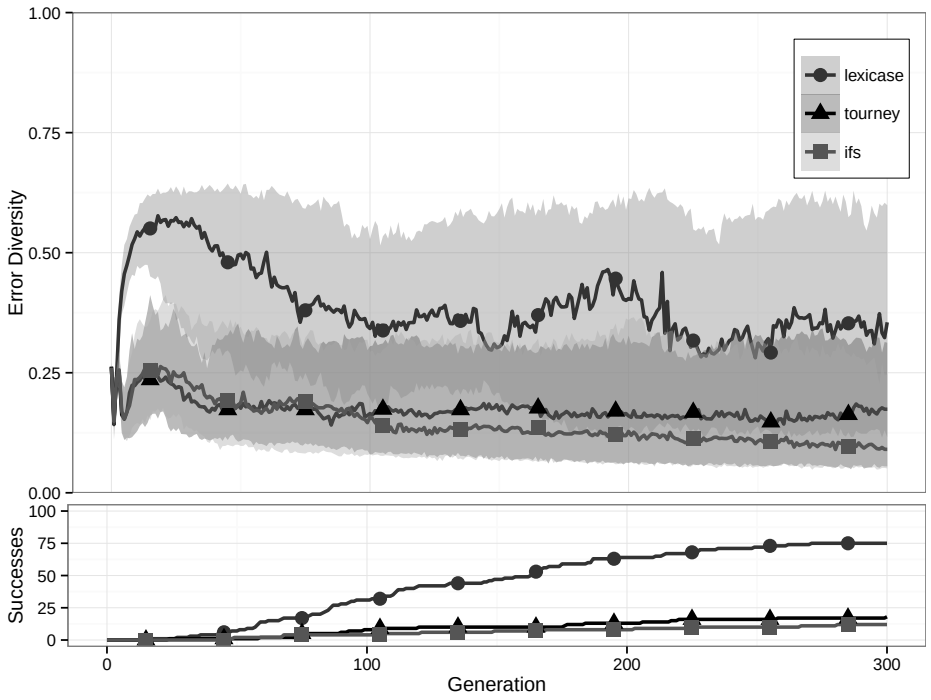

Figure 4.5. String Lengths Backwards – error diversity median (line) and quartiles (shaded)

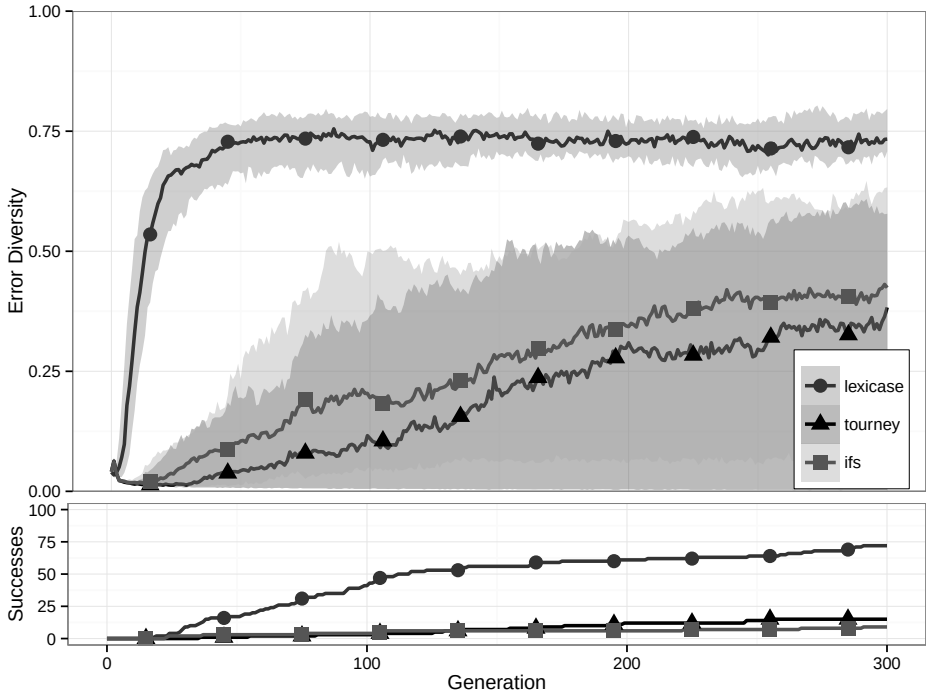

Figure 4.6. Negative To Zero – error diversity median (line) and quartiles (shaded)

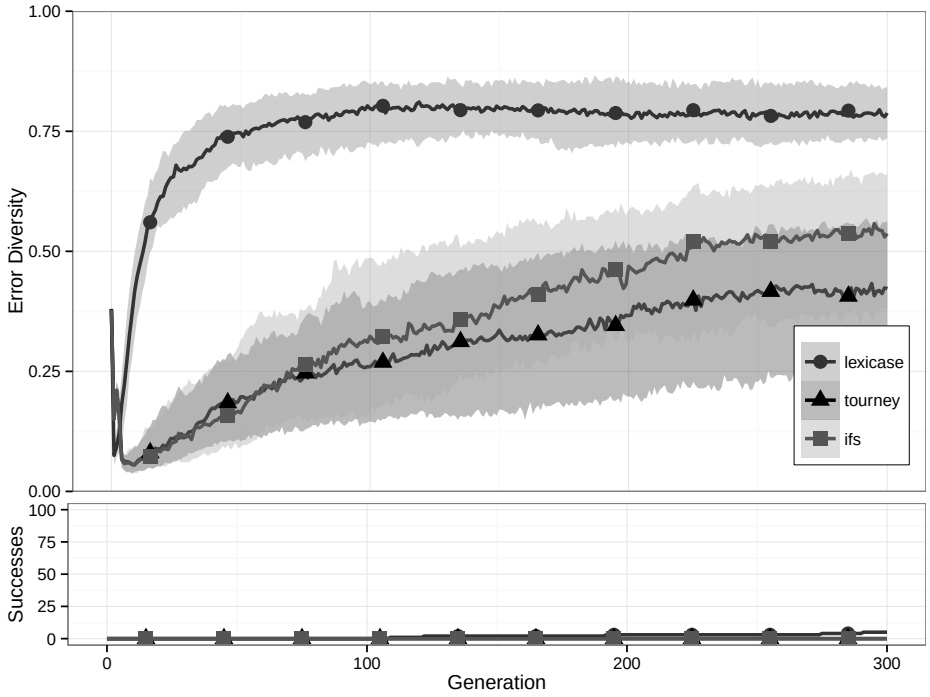

Figure 4.7. Double Letters – error diversity median (line) and quartiles (shaded)

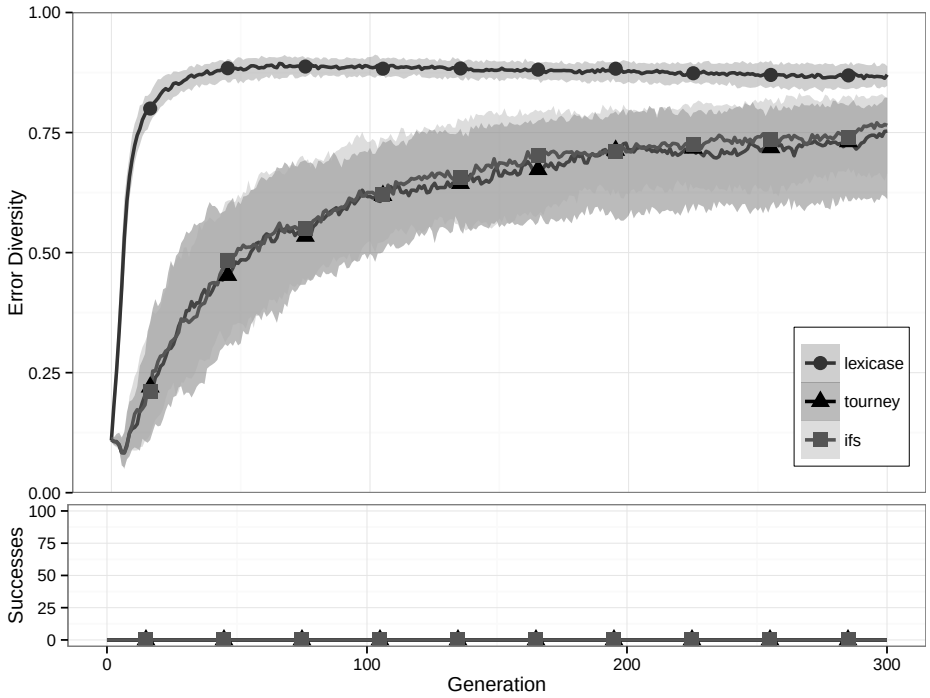

Figure 4.8. Scrabble Score – error diversity median (line) and quartiles (shaded)

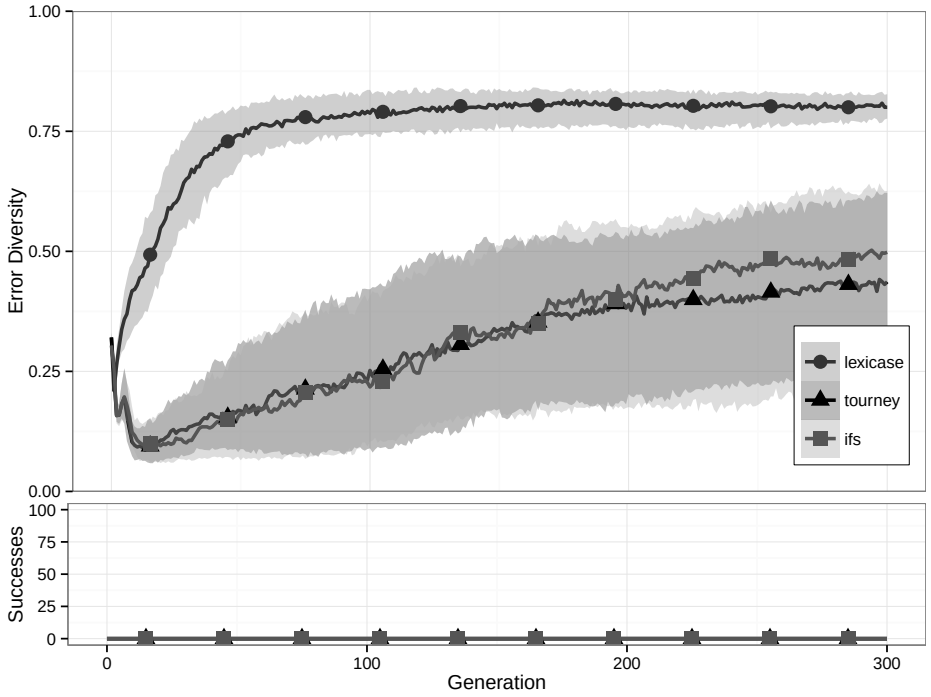

Figure 4.9. Checksum – error diversity median (line) and quartiles (shaded)

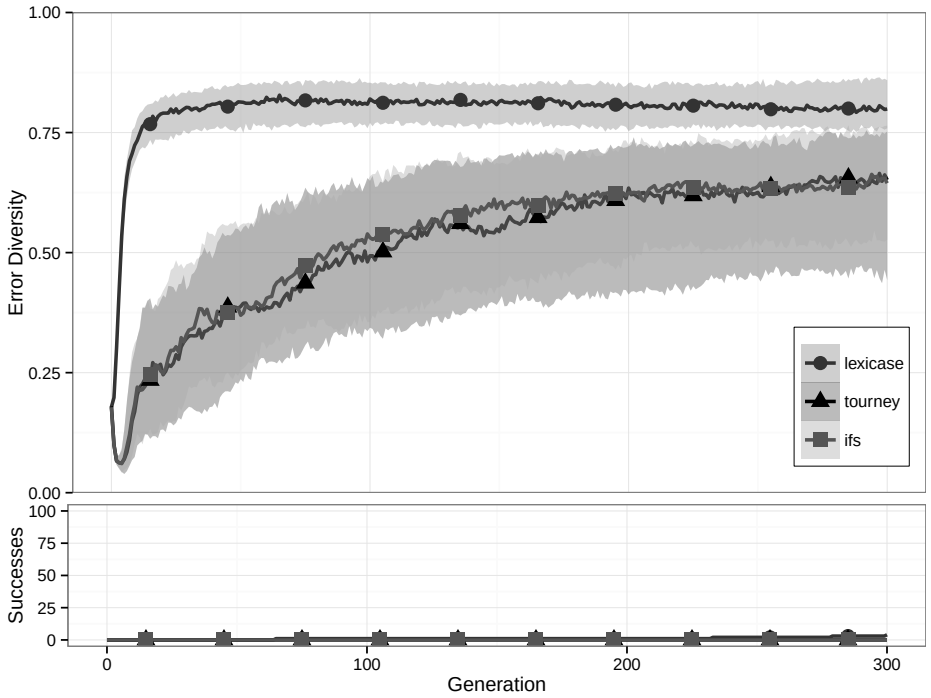

Figure 4.10. Count Odds – error diversity median (line) and quartiles (shaded)

In general the error diversity numbers for lexicase selection are significantly higher than those for either tournament selection or IFS, which tend to be extremely similar. The String Lengths Backwards problem was the only problem for which we found any substantial overlap between the range of values for lexicase and the other two selection mechanisms (see Figure 4.5). Typically the lexicase error diversity rises very sharply in the early generations and levels off somewhere between 0.75 and 1.0, meaning that $\frac{3}{4}$ or more of the individuals in the lexicase runs have distinct error vectors. This is in contrast to the tournament selection and IFS results, in which the median error diversity values rarely rise above 0.5; the two exceptions are on the Scrabble Score and Count Odds problems (Figures 4.8 and 4.10), on which the error diversity values approach or exceed 0.75, but neither problem was solved by tournament selection or IFS.

The error diversity for the lexicase runs was much higher than for tournament and IFS for most problems, which is consistent with the hypothesis that lexicase selection helps maintain diversity. The lexicase error diversity values tended to plateau at or above 0.75, meaning that in a population of $1,000$ individuals there were over 750 *distinct* error vectors. This doesn't mean that different individuals were *solving* different test cases; it could just be that many had different incorrect answers and error values. From a search perspective, though, this still seems useful, as those different error values may represent different starting points for subsequent search.

We found it remarkable that while IFS was designed to maintain population diversity, in our experiments it never achieved higher levels of error diversity than lexicase selection, but instead showed very similar levels of diversity to tournament selection on all problems. Since others ([86, 64]) have shown clear increases in diversity using IFS, it is unclear why these experiments did not replicate those results; this could be a product of testing on different types of problems, since no prior results were on general program synthesis problems. At the same time, both tournament selection and

IFS aggregate test case errors into a single value on which they base tournaments for selection, with IFS just weighting the components differently; this may be partially responsible for the similar rates in diversity.

On problems for which solutions were discovered, lexicase selection runs found solutions throughout the 300 generations. This, combined with the high levels of error diversity, gives one hope that meaningful search can still occur late in a lexicase selection run. The plots of successes over time under the primary plots typically appear to have positive slope even at generation 300, so it would be interesting to extend these runs to 500 or 1,000 generations and see how many additional solutions are discovered. If lexicase selection is indeed maintaining meaningful diversity then we would expect to see continued discovery of solutions, at a higher rate than for either tournament selection or IFS. This might be particularly interesting for problems for which solution discovery is rare but possible, such as Double Letters and Count Odds, which are solved using lexicase selection 5 and 3 times respectively, but not at all using tournament selection or IFS. Solutions for these two problems tended to be discovered later in the run (Double Letters in generations 109, 122, 192, 275, and 291; Count Odds in 65, 233, 279), so letting runs on those problems go longer might increase their successes.

4.4.2 Exploitation

Evolutionary algorithms use selection pressure to direct search, with the goal of refining promising programs into better ones. This exploitation of already-discovered individuals helps drive search toward solutions. All parent selection mechanisms select some individuals more than others, but vary in how they bias selection. The lone exception is uniform selection, which does not bias selection; in our performance trials in Section 4.2, it unsurprisingly performed very poorly.

In Section 4.3 we noted some interesting behaviors of lexicase selection, in particular that it often seemed to give many parent selections to single individuals, including some individuals that fell quite low in rankings based on total error. This extreme exploitation of specific individuals strongly contrasts to how often we would expect tournament selection to select any individual, let alone one with poor total error. In this section, we will systematically investigate the hyper-selection of single individuals in GP using lexicase selection.

Let us call an individual *hyper-selected at the X% level* if that individual receives at least X% of the parent selections in a single generation. For example, an individual that receives at least 170 selections out of 1700 total in a generation is considered hyper-selected at the 10% level (as well as any level below 10%). Examining hyper-selection events can help us characterize how often single individuals receive a large percent of the selection pressure in their generations; here, we will look at hyper-selection events at the 1%, 5%, and 10% levels.

With tournament selection, the number of times an individual can be selected is limited by the number of tournaments in which it participates. If the best member of the population participates in 1% of the tournaments for a given generation, it will be selected 1% of the time that generation, but no more. Since the expected number of tournaments in which each individual participates is constant for a particular population size P and tournament size t, the probability of an individual being selected by tournament selection is entirely determined by its rank in the population. In particular, Bäck [8, 12] shows that the probability of selecting an individual with rank $i \in [1, P]$, with $i = 1$ being the best rank, is

$$p(i) = \frac{(P - i + 1)^t - (P - i)^t}{P^t} \tag{4.1}$$

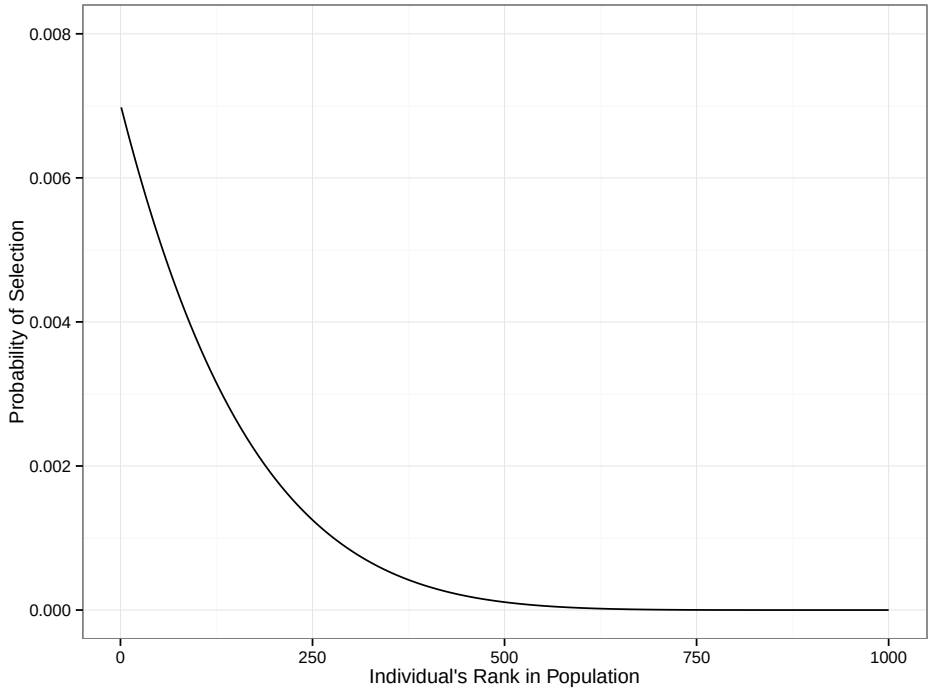

Figure 4.11. Probability mass function of selecting individual with rank i out of a population of 1000 individuals using tournament selection with tournament size 7, assuming no two individuals have the same rank. This plots Equation 4.1.

assuming no two individuals have the same fitness. With ties in the rankings, this equation does not hold exactly, but is approximately correct unless there are many tied individuals. We plot this probability mass function in Figure 4.11.

From Equation 4.1, we see that in our runs using population size 1000 and tournament size 7, the best few individuals will be selected approximately 0.7% of the time each. This also follows from the fact that every individual will participate in approximately 0.7% of the tournaments, and the best individual will win each tournament in which it participates. With tournament selection it would therefore be unlikely to hyper-select many individuals at the 1% level in a generation, and extremely unlikely for any individuals to be hyper selected at the 5% or 10% level. On the other hand, in Section 4.3 we saw an example of lexicase selection rewarding an interesting individual with over 90% of the selections in a generation. We therefore expect lexicase

Table 4.4. The average number of hyper-selected individuals at the 1%, 5%, and 10% levels per generation for both lexicase selection and tournament selection.

	Lexicase			Tournament		
Problem	**1%**	**5%**	**10%**	**1%**	**5%**	**10%**
Double Letters	12.28	0.29	0.09	0.36	0.00	0.00
Replace Space with Newline	13.39	0.38	0.11	0.28	0.00	0.00
String Lengths Backwards	6.21	0.54	0.25	0.42	0.00	0.00
Vector Average	6.99	0.02	0.01	0.91	0.00	0.00
Count Odds	0.49	0.02	0.00	0.70	0.00	0.00
Mirror Image	9.72	0.23	0.05	0.22	0.00	0.00
X-Word Lines	5.31	0.13	0.04	0.36	0.00	0.00
Negative To Zero	8.21	0.39	0.16	0.43	0.00	0.00
Syllables	5.74	0.13	0.05	0.38	0.00	0.00

selection to produce non-zero numbers of hyper-selections at the 5% and 10% levels, though without empirical data it is unclear how common these will be.

To measure hyper-selection, we gathered data from using lexicase selection and tournament selection with size 7 tournaments on nine benchmark problems, a subset chosen to exhibit a range of problem requirements and difficulties[11]. We then calculated the average number of hyper-selected individuals at the 1%, 5%, and 10% levels per generation, which we present in Table 4.4. On all problems except one, lexicase selection hyper-selects 5 or more individuals per generation on average at the 1% level; tournament selection averages less than one per generation on all problems, though always greater than 0.2. Unsurprisingly, tournament selection never hyper-selected an individual at the 5% or 10% levels. On 7 of the 9 problems, lexicase selection hyper-selected one individual at the 5% level in every 2 to 10 generations on average, and one individual at the 10% level every 4 to 25 generations on average. The two problems Vector Average and Count Odds showed much lower levels of hyper-selection with lexicase selection, indicating that selecting a single individual

[11]Due to a bug, an instruction equivalent to `exec_noop` was included in the instruction set for the runs on the problems Count Odds, Mirror Image, Vector Average, Double Letters, String Lengths Backwards, and X-Word Lines. See Footnote 5 on page 70 for more details on this bug.

to parent many of the children in a single generation was rarer on those problems. These two problems were also among the least-solved problems in this subset of the benchmark problems; one hypothesis is that most of the GP runs had trouble getting any traction on these problems, leading to more homogeneous populations and fewer hyper-selection events than on other problems.

Considering that tournament selection conforms to the probability of selection given in Equation 4.1 regardless of problem, it is at first surprising that its hyper-selections at the 1% level vary as much as they do across problems. This difference is likely explained by how often tied individuals appear near the top of the rankings for different problems, since ties make Equation 4.1 not strictly hold. Intuitively, if many individuals tie for the best rank, they will each win fewer tournaments than a single best individual would, since ties in tournaments are broken randomly. Therefore, lower hyper-selection for tournament selection on a problem likely indicates that ties happened more often on those problems, which we have observed anecdotally in a few runs.

We do not present hyper-selection results for our runs using IFS selection here. In fact, without ties, we would expect IFS to give identical results to tournament selection. IFS is a variation on tournament selection where the fitness of individuals weights some test cases more heavily than others; it then uses tournament selection over the weighted fitnesses. Even though the ranking of individuals is different than when simply using total error, the probability of selecting the individual ranked i still follows Equation 4.1. Therefore, hyper-selection results using IFS should be very close to tournament selection, only differing when the two methods produce different numbers of tied individuals.

The results in Table 4.4 clearly show that lexicase selection gives more of its parent selections to single individuals than tournament selection, both at low levels (1%) and higher levels (5% and 10%) of hyper-selection. This indicates that lexicase selection

more often concentrates its selection pressure on single individuals or small groups of individuals than tournament selection, increasing its exploitation of the individuals it selects most often. This data does not indicate whether lexicase selection is hyper-selecting the same individuals that tournament selection ranks highest (those with best total error), or if it actually selects individuals that would receive few or no selections with tournament selection.

We have previously noted lexicase selection outperforming tournament selection across many benchmark problems, and now observe that it often concentrates selection pressure into small numbers of individuals. This raises important questions: can we attribute lexicase selection's success to its ability to concentrate selection in hyper-selection events? Or, is it more important that lexicase selects different individuals than tournament selection, in some sense the "right" individuals to drive evolution toward a solution? We will investigate these questions in Section 4.5.1.

4.5 Experimental Analysis of Lexicase Selection

In this section, we present three experiments that help explain why lexicase selection performs better than tournament selection and IFS. These experiments systematically test hypotheses building upon our prior observations, shedding light upon which aspects of lexicase are crucial to its success, and which are simply byproducts of the algorithm.

In Section 4.5.1 we extend our study of exploitation through hyper-selection to see whether lexicase selection's significant exploitation of single individuals helps it steer evolution toward solutions. Next, in Section 4.5.2 we look into individuals that specialize in some test cases while performing poorly on others, and whether these contribute significantly to lexicase selection's success. Finally, we delve into the importance of the clustering of individuals that perform well on similar test cases in Section 4.5.3.

4.5.1 Hyper-Selection and Lexicase Performance

In Section 4.4.2 we saw that lexicase selection often ends up selecting the same individual many times in one generation, much more often than tournament selection does. This leads to the question of whether the hyper-selections observed in lexicase selection runs are important in driving evolution toward solutions, or if they are simply a side effect of lexicase's algorithm. The alternative is that the individuals that lexicase selection selects the most often are simply different from those that tournament selection selects most often, in particular those with poor total error. In this section we test the hypothesis that the hyper-selections we observed in runs using lexicase are integral to its success, and that without these extreme exploitative events, lexicase selection would perform significantly worse than it does with them.

To test this hypothesis, we designed a new parent selection algorithm that selects the same individuals most often that lexicase does, but has hyper-selection characteristics much closer to tournament selection. The new algorithm, *sampled lexicase-tournament selection* (SLT), starts by sampling the population, which only happens once per generation before selecting any parents. We sample k individuals from the population by running the lexicase selection algorithm and tracking how often each individual is selected. In this work we set $k = 2P$, where P is the population size (set to 1000 in our runs), guaranteeing at least as many samples as the number of parents that will be selected in that generation[12]. We then use the number of samples each individual received to rank the population from best (most samples) to worst (least samples). Next, every time we need to select a parent, we conduct a tournament, where the winner of the tournament is based on the lexicase-sampled ranking instead

[12]We observe around 1700 parent selections per generation on average, which varies since we randomly select genetic operators, and some operators require one parent where others require two. This means that at most, 2000 parents could be selected in a generation.

Table 4.5. The average number of hyper-selected individuals at the 1%, 5%, and 10% levels per generation for lexicase selection, tournament selection and SLT selection. This table adds SLT to the results in Table 4.4.

	Lexicase			Tournament			SLT		
Problem	**1%**	**5%**	**10%**	**1%**	**5%**	**10%**	**1%**	**5%**	**10%**
Double Letters	12.28	0.29	0.09	0.36	0.00	0.00	1.54	0.00	0.00
Replace Space with Newline	13.39	0.38	0.11	0.28	0.00	0.00	1.56	0.00	0.00
String Lengths Backwards	6.21	0.54	0.25	0.42	0.00	0.00	1.53	0.00	0.00
Vector Average	6.99	0.02	0.01	0.91	0.00	0.00	1.56	0.00	0.00
Count Odds	0.49	0.02	0.00	0.70	0.00	0.00	1.54	0.00	0.00
Mirror Image	9.72	0.23	0.05	0.22	0.00	0.00	1.55	0.00	0.00
X-Word Lines	5.31	0.13	0.04	0.36	0.00	0.00	1.55	0.00	0.00
Negative To Zero	8.21	0.39	0.16	0.43	0.00	0.00	1.55	0.00	0.00
Syllables	5.74	0.13	0.05	0.38	0.00	0.00	1.55	0.00	0.00

of total error. In this experiment we used size 7 tournaments, just like we did with tournament selection and IFS in our experiments.

SLT can be seen as a variation of tournament selection in which fitness is based on lexicase sampling instead of total error. SLT gives the highest probabilities of selection to those individuals that lexicase would select the most often in the population. But, since it uses tournaments for selection, its probability of selecting the individual ranked i in the lexicase-sampled ranking will be same as in tournament selection, as given in Equation 4.1. Therefore, we would expect the hyper-selection characteristics of SLT to mirror those of tournament selection, and differ only when the two behave differently with respect to tied individuals in the rankings, especially ties amongst the best individuals.

We conducted 100 runs of PushGP[13] using SLT on the same 9 benchmark problems from Table 4.4. We present the hyper-selection results for SLT, in addition to those for lexicase selection and tournament selection, in Table 4.5. The first thing to note

[13]Due to a bug, an instruction equivalent to `exec_noop` was included in the instruction set for the runs using SLT. See Footnote 5 on page 70 for more details on this bug.

is that SLT has higher hyper-selection at the 1% level than tournament selection. Theoretically, we would expect SLT to behave similarly to tournament selection if neither had ties in rank within the population. We believe the differences we see here are a product of ties in total error when using tournament selection, especially near the top of the rankings. The relative consistency of SLT's 1%-level hyper-selection likely comes from the fact that it rarely had large numbers of tied individuals near the top of the rankings—we expect that tournament selection without ranking ties would also average around 1.55 hyper-selections at the 1% level per generation.

In these runs, SLT usually had lower hyper-selection at the 1% level than lexicase selection, and always lower at the 5% and 10% levels, on which it never had a non-zero result. Since SLT was designed to have similar hyper-selection characteristics as tournament selection, it is unsurprising that it received no hyper-selections at the upper levels. This means that SLT succeeds in our goal of creating a lexicase-based selection mechanism that never puts as much as 5% of the parent selections in a generation on a single individual. This contrasts with lexicase selection, which often selects single individuals to make large numbers of the children for the next generation.

Since we have shown that SLT has similar hyper-selection characteristics to tournament selection, let us now examine its performance results in these runs, which we present in Table 4.6. Across these 9 problems, SLT shows very similar performance to lexicase selection, and better performance than tournament selection on every problem. Both SLT and lexicase found at least one solution on each of the 9 problems, where tournament selection only found solutions to 6 of the problems. Comparing these methods using a chi-square test with the Holm correction, SLT never has a significantly different success rate compared to lexicase selection. SLT is significantly better than tournament selection on the same problems as lexicase except for Count Odds and X-Word Lines, on which it achieved fewer than the 8 successes necessary to

Table 4.6. Number of successful runs out of 100 for each setting on each problem. Lexicase selection and tournament selection results are same as those in Table 4.1.

Problem	Lexicase	Tournament	SLT
Double Letters	6	0	4
Replace Space with Newline	51	8	61
String Lengths Backwards	66	7	79
Vector Average	16	14	30
Count Odds	8	0	5
Mirror Image	78	46	84
X-Word Lines	8	0	4
Negative To Zero	45	10	53
Syllables	18	1	13

be significantly better than tournament. SLT seems to slightly outperform lexicase selection on the easier problems where both find more solutions, and lexicase selection slightly outperforms SLT on the more difficult problems where both find fewer solutions, though the difference is never significant.

We plotted the diversity across generations from the runs using SLT, as we did for other techniques in Section 4.4.1. Interestingly, the diversity plots for SLT were virtually indistinguishable from those of lexicase selection, so we omit them here. Thus, even though the techniques produce significantly different hyper-selection rates, their populations still maintain similar abilities to search widely.

These results show that even though SLT has much lower hyper-selection characteristics than lexicase selection, never selecting a single individual to parent more than 5% of the children in a generation, it nevertheless maintains the problem-solving performance shown by lexicase selection. These results give strong evidence against the hypothesis that lexicase selection's increased exploitation of hyper-selected individuals is important in its ability to outperform tournament selection and IFS. Instead, this suggests that it is more important *which* individuals lexicase selection selects most often, which it has in common with SLT but not tournament selection.

While SLT achieved similar performance to lexicase selection in this experiment, it does not otherwise indicate that it would make a better parent selection mechanism. Notably, it will perform slightly slower than lexicase selection in practice, since it performs both lexicase sampling and then tournaments for selection. Even so, it may merit further examination on other types of problems to see if it behaves differently in other settings.

4.5.2 Specialists with Poor Total Error

By considering test cases one at a time, lexicase selection often selects an individual without considering all of the test cases; this idea explicitly influenced the design of lexicase selection. When halting before seeing all of the test cases, the lexicase algorithm will ignore the error values on all other test cases, regardless of whether they are relatively good or relatively poor compared to the rest of the population. Lexicase selection therefore has the ability to select specialist individuals that perform extremely well on some cases while having very poor error on other cases. Tournament selection very rarely selects such specialists, since it bases fitness on total error, which would include the error values on which the individuals performed poorly.

In Section 4.3 we described a single run that featured two individuals in the bottom quartile of the population (when sorted by total error) that had over 100 children each; one of those programs was the parent of 45 successful programs. While this anecdote shows that lexicase selection selects specialists, it is unclear whether such individuals receive any significant portion of parent selections, and if their selection is important in directing search when using lexicase selection. Does lexicase selection perform well because it selects specialists, or can it maintain good performance without selecting individuals with poor total error? We hypothesize that lexicase selection's ability to select specialist individuals with poor total error allows it to more effectively explore the search space than if it were limited to selecting individuals with good performance

Table 4.7. The probability of tournament selection selecting an individual that would be removed by $X\%$ elitist survival. For example, the probability of selecting an individual removed by 50% elitist survival is 0.00781, meaning that individuals with total error worse than the median make up less than 0.8% of the parents when using tournament selection.

% Elitist Survival	Probability of Selecting A Removed Individual
25	0.13348
50	0.00781
75	0.00006

when measured by total error. We do not expect to see nearly as dramatic decreases in performance by tournament selection, which does not often select individuals with poor total error. Additionally, we expect that limiting lexicase selection to individuals with better total error will decrease population diversity.

To test our hypotheses, we propose an experiment where parent selection cannot select individuals with poor total error relative to the population. We devised a new survival selection step to run before parent selection called *elitist survival selection*. During elitist survival selection, we sort the population by total error and only allow the best $X\%$ of the population to "survive" to be available to make children. We then conduct parent selection using this reduced population as normal. With 100% elitist survival we would keep the entire population (i.e. no individuals are removed); 75% elitist survival would keep three quarters of the population, etc.

We conducted runs of PushGP[14] using elitist survival to remove 75%, 50%, and 25% of the population. Based on Equation 4.1, we can calculate how often we would expect tournament selection to select the individuals excluded by elitist survival[15].

[14]Due to a bug, an instruction equivalent to `exec_noop` was included in the instruction set for the runs using elitist survival at the 75%, 50%, and 25% levels. See Footnote 5 on page 70 for more details on this bug.

[15]Figure 4.11 on page 91, which plots the probability distribution defined by Equation 4.1, is useful when visualizing these cumulative probabilities..

Table 4.8. Number of successful runs out of 100 for each setting of elitist survival. The column headers indicate what percent of the population is kept by elitist survival with each selection technique. The 100% elitist survival runs are equivalent to not using elitist survival, and as such the results are the same as those in Table 4.1. Underline indicates results that are significantly worse than the 100% column, and asterisk (*) indicates results that are significantly better than the 100% column. No tournament selection runs were significantly different from the 100% tournament selection column.

	Lexicase				Tournament			
Problem	**100%**	**75%**	**50%**	**25%**	**100%**	**75%**	**50%**	**25%**
Double Letters	6	1	1	0	0	0	0	0
Replace Space with Newline	51	46	20	24	8	13	11	9
String Lengths Backwards	66	47	17	17	7	6	12	10
Vector Average	16	*33	*49	25	14	11	5	8
Count Odds	8	3	0	1	0	0	0	0
Mirror Image	78	78	67	48	46	41	34	44
X-Word Lines	8	17	4	0	0	0	0	0
Negative To Zero	45	28	19	9	10	5	10	7
Syllables	18	13	10	8	1	2	1	3

The probabilities of tournament selection choosing an individual removed by elitist survival are given in Table 4.7. Tournament selection would select a decent proportion of the individuals removed by 25% elitist survival, at around 0.13. We can see that most of those individuals are ranked between 25% and 50%, since tournament selection selects individuals worse than the median with probability of only about 0.008. Thus, we would not expect 50% survival elitism to affect the performance of tournament selection, and certainly not 75% survival elitism. Even 25% survival elitism may have negligible effects.

Table 4.8 gives the number of successful runs on 9 benchmark problems using 75%, 50%, and 25% elitist survival with lexicase and tournament selection. We compare these results to 100% elitist survival, which is equivalent to not using elitist survival, since the entire population is kept. On 6 of the problems, the number of successes using lexicase selection was significantly lower when using either 50% or 25% elitist

survival, and for 3 problems both. On the String Lengths Backwards problem, the number of successes was significantly lower even using 75% elitist survival. On the other hand, on the Vector Average problem, lexicase selection found *more* successes with 75% and 50% elitist survival. Results using tournament selection were not significantly different with any ratio of elitist survival on any problem.

We plot the population error diversity for most of these problems in Figures 4.12–4.18. On all problems, the diversity of runs using tournament selection remains essentially the same at all levels of elitist survival. This result is consistent with the unchanged performance of tournament selection with elitist survival, both of which can be explained by the small portion of selections affected by elitist survival.

On the other hand, for most problems the median diversity of runs using lexicase selection decreases as the number of individuals removed by elitist survival increases. We see this decrease across all problems besides Vector Average, though the impact varies per problem. On the Replace Space With Newline (Figure 4.12) and String Lengths Backwards (Figure 4.14) problems, lexicase selection with 50% and 25% elitist survival grow in diversity early, but then lose diversity and finish the remainder of the run with similar diversity to tournament selection. On the other problems, the lower percents of elitist survival have similar curves to the higher percents, just with lower diversity.

One interesting finding here is that the one problem where elitist survival seems to help lexicase selection—Vector Average—is also the only problem on which lexicase selection did not perform significantly better than tournament selection. Additionally, this is the only problem for which removing individuals through elitist survival *increases* the population diversity, as shown in Figure 4.18. These results suggest that for this problem, total error is a good search driver, since removing the individuals with worst total error helped lexicase.

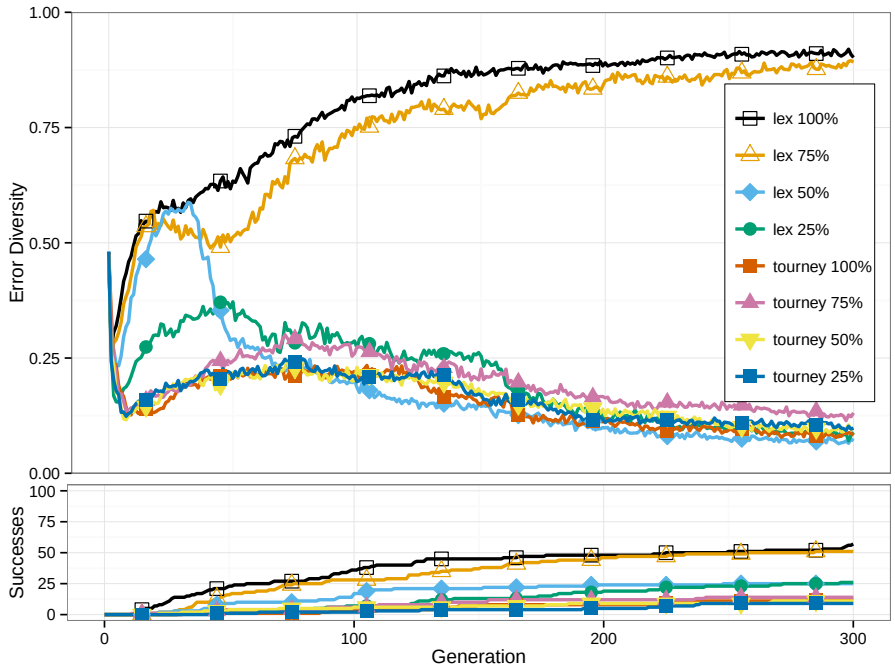

Figure 4.12. Replace Space With Newline – Median error diversity for lexicase and tournament selections using elitist survival at different percents.

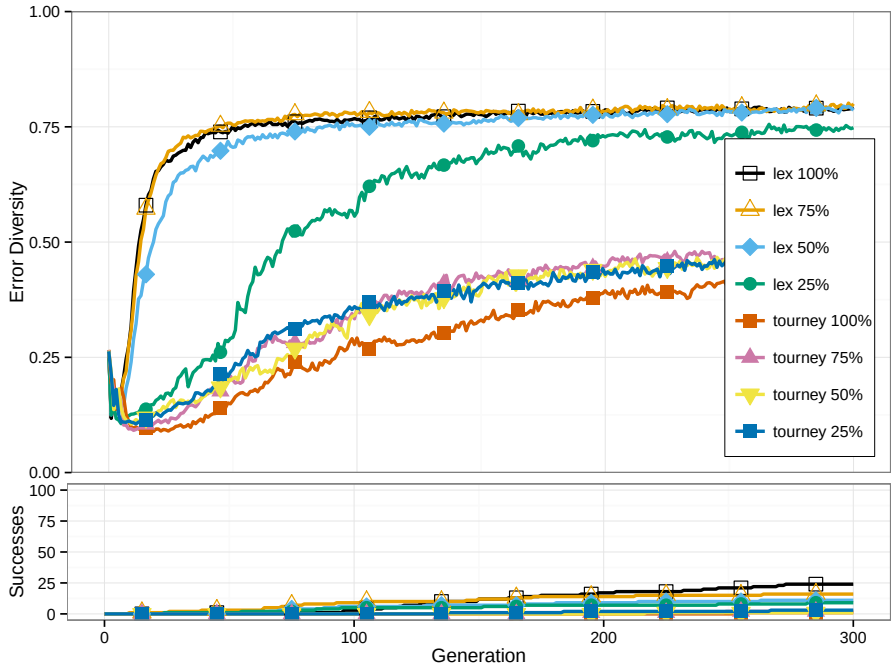

Figure 4.13. Syllables – Median error diversity for lexicase and tournament selections using elitist survival at different percents.

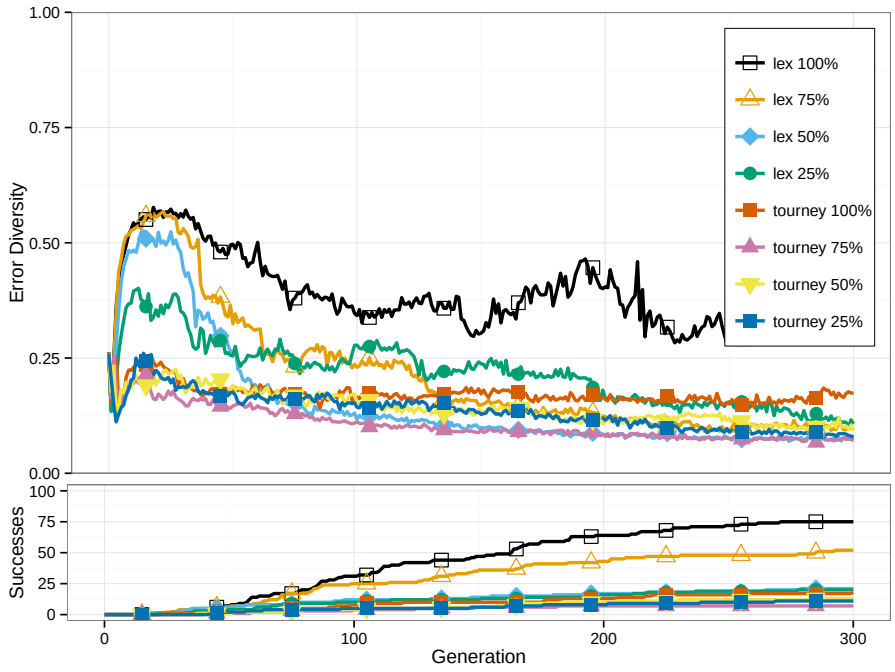

Figure 4.14. String Lengths Backwards – Median error diversity for lexicase and tournament selections using elitist survival at different percents.

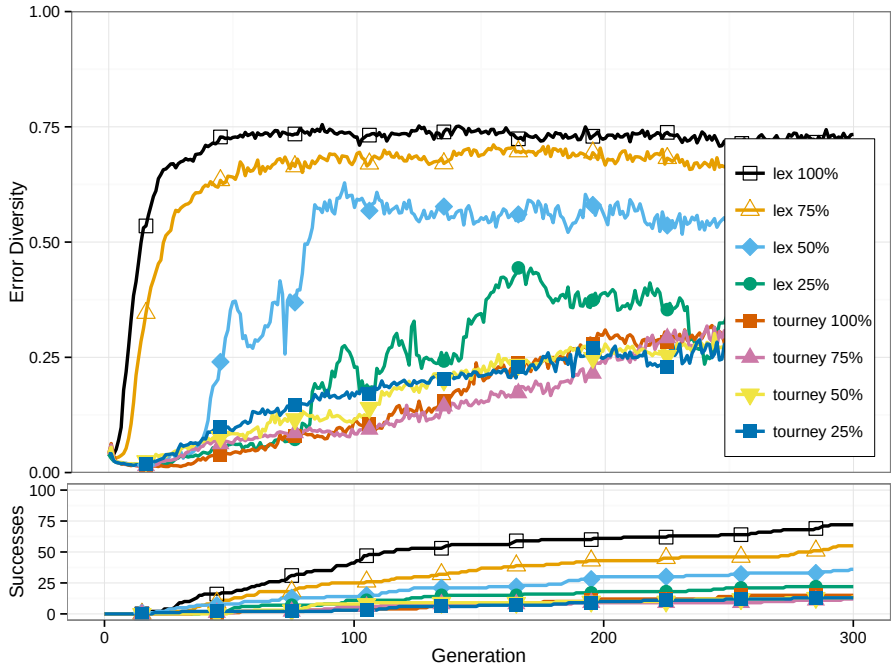

Figure 4.15. Negative To Zero – Median error diversity for lexicase and tournament selections using elitist survival at different percents.

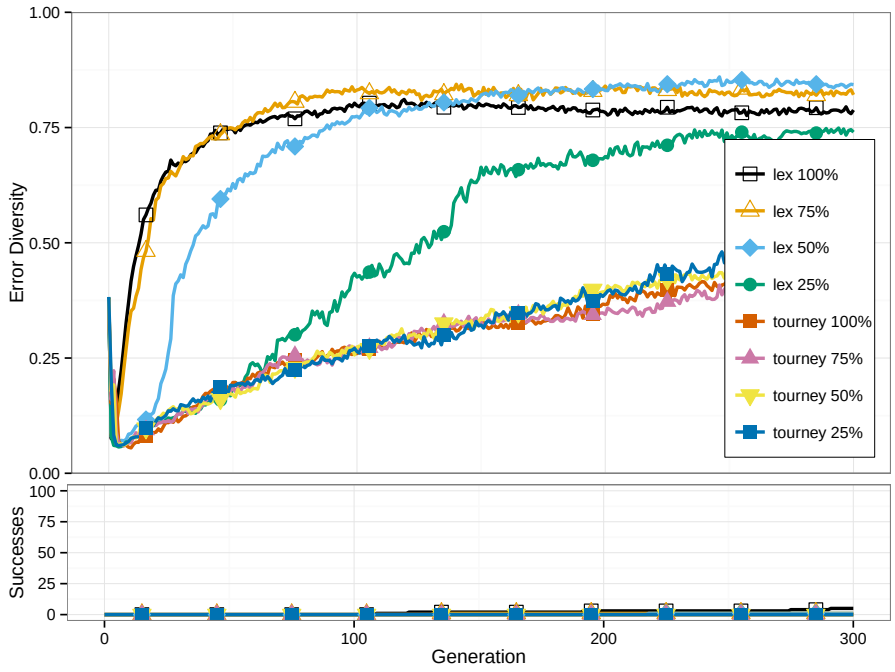

Figure 4.16. Double Letters – Median error diversity for lexicase and tournament selections using elitist survival at different percents.

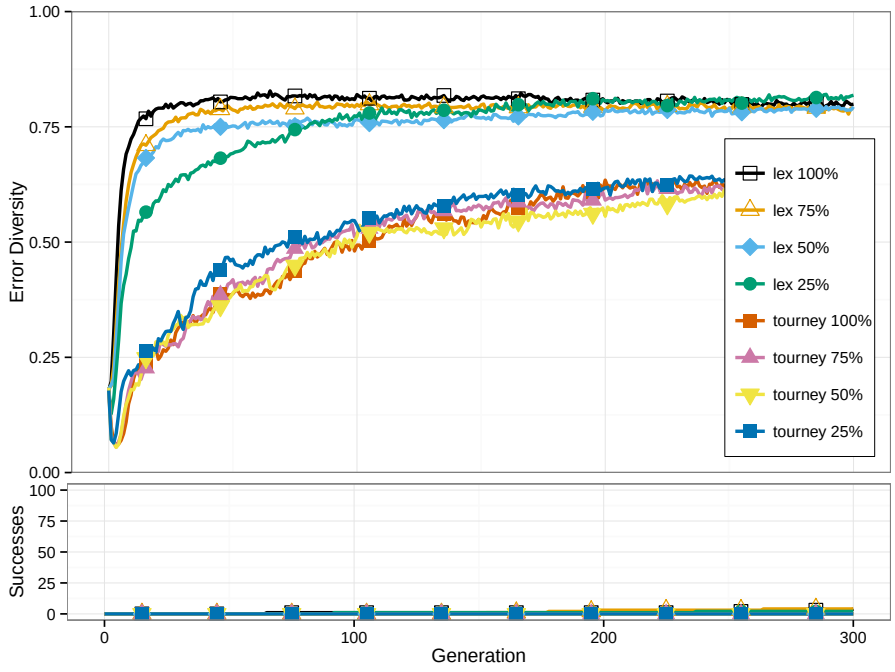

Figure 4.17. Count Odds – Median error diversity for lexicase and tournament selections using elitist survival at different percents.

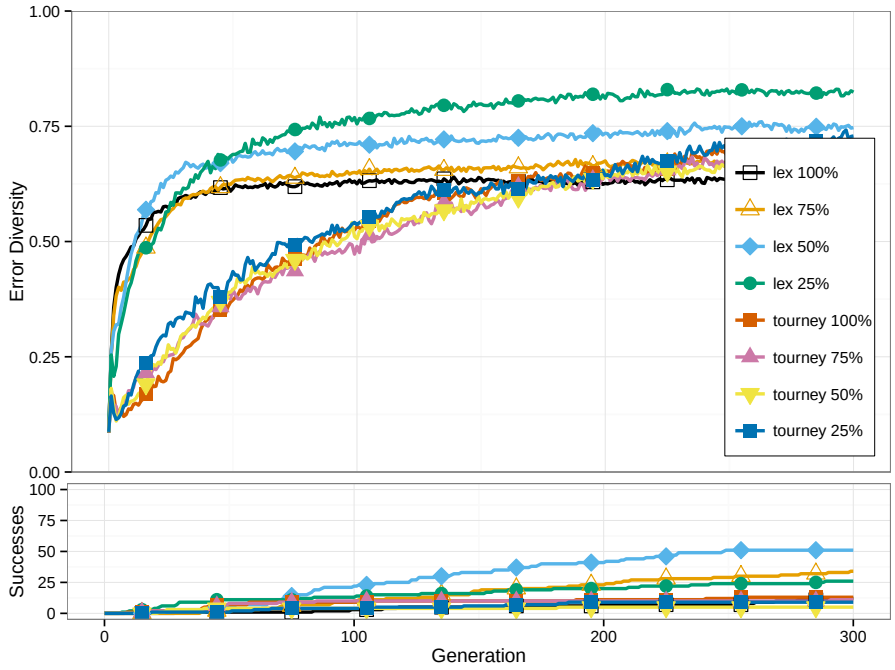

Figure 4.18. Vector Average – Median error diversity for lexicase and tournament selections using elitist survival at different percents.

These results provide evidence for our hypothesis that lexicase selection makes use of specialist individuals with poor total error relative to the rest of the population—individuals that presumably have poor errors on some test cases but good errors on others. Since lexicase performs worse when we remove the bottom half or three quarters of the population on most of the problems, it is clear that lexicase selects specialists in these parts of the population and that they help drive the direction of evolution. Even removing the bottom quarter of the population (75% elitist survival) resulted in fewer successes on 6 out of the 9 problems, albeit significantly so on only 1.

Our plots show that error diversity in lexicase runs decreases, sometimes significantly, as we decrease the number of individuals that survive elitist survival. These plots support our hypothesis that the high diversity seen in runs using lexicase selection is influenced by lexicase's ability to select individuals with relatively poor total

error. The decreased diversity in runs using lower percents of elitist survival likely contributes to the corresponding decreases in performance observed in Table 4.8.

As expected, tournament selection was not significantly affected by elitist survival selection at any level, even when removing the bottom three quarters of the population. As we saw in Table 4.7, tournament selection simply does not often select specialist parents from the bottom ranks of the population. Instead, it concentrates on the individuals with the best total error, and mostly selects from the top quarter of the population. This difference between lexicase selection and tournament selection likely explains at least part of their difference in performance.

4.5.3 Population Clustering[16]

In Section 4.5.2 we showed that lexicase selection makes use of specialist individuals to direct evolution. With this in mind, it seems possible that lexicase selection enables groups of specialists that solve different parts of the problem to evolve independently side-by-side, implicitly maintaining the kind of niches that are maintained more explicitly by island models [115] and related methods. We expect that evolution may sometimes progress when individuals from different groups mate, producing a child that combines the abilities of its parents. The hope is that this process, iterated, will eventually produce an individual that solves the entire problem.

Here we explore the effects of different parent selection methods on the development of clusters of individuals that perform similarly across the test cases. We hypothesize that using lexicase selection will result in relatively larger numbers of clusters, since it selects individuals on the basis of specific cases and groups of cases rather than on overall performance, and that the combination of these clusters through crossover helps lead evolution toward a solution.

---

[16]Much of the text in this section is adapted from a book chapter co-authored with Nicholas Freitag McPhee and Lee Spector [33].

To examine this idea, we must be able to measure the clustering of a population with respect to the test cases. We base the clustering of the population on the individuals' error vectors across the test cases in the training set. Since lexicase selection concentrates on individuals that perform at least as well as every other individual in the population, we convert the error vectors into binary "elitized" error vectors that indicate whether an individual achieved the best error on each test case in that generation. More formally, if each individual j in the population P has error vector $error_j$ containing error values on the test cases T, then the elitized error vector for individual i is defined by

$$elitized_i[t] = \begin{cases} 0, & \text{if } error_i[t] = \min_{j \in P}(error_j[t]) \\ 1, & \text{otherwise} \end{cases}$$

for $t \in T$. By elitizing the error vectors, we can ignore the differences between individuals that perform poorly on cases in different ways, and concentrate on how individuals cluster based on the cases on which they perform well.

In this work we use agglomerative clustering[17] to count the number of clusters in the population at each generation. Agglomerative clustering creates a hierarchical clustering model by first placing each individual into its own cluster. It then iteratively combines the two closest clusters into a single cluster, until all clusters have been combined into one cluster, recording at each step the distance between the clusters in each merged pair. We can then find the number of clusters separated by a specified distance by counting the number of clusters merged beyond that distance. For example, in Figure 4.19 we have plotted a dendrogram generated through agglomerative clustering, where the joining of clusters is represented by two vertical

[17]We used the `agnes` [83] implementation of agglomerative clustering in R [108], using the `average` linkage when combining clusters.

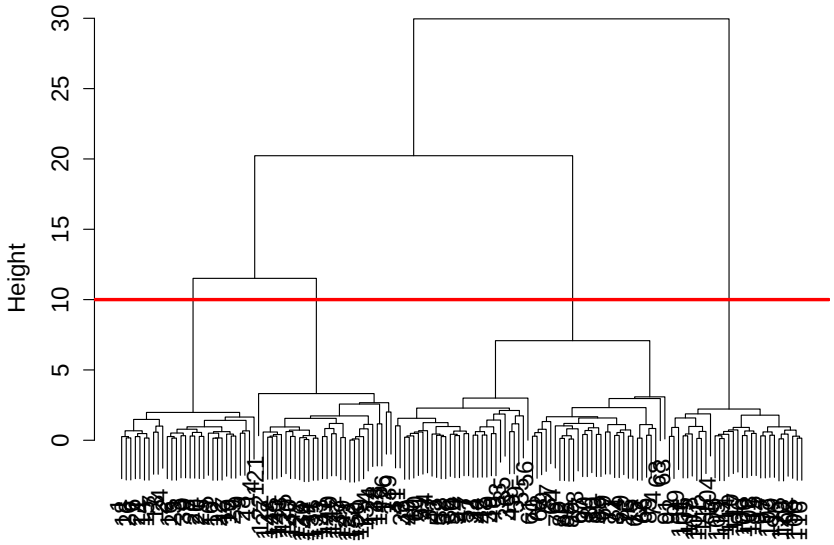

Figure 4.19. Example of a dendrogram created by agglomerative clustering. The red line at height 10 shows that this data has 4 clusters that are at least distance 10 apart from each other.

lines connected by a horizontal line. On this dendrogram, we have drawn a red line at height 10. We can see that their are 4 clusters that are at least distance 10 from each other, represented by the 4 vertical lines that the red line crosses.

Since we are using binary elitized error vectors, we use the Manhattan distance as our distance metric, which makes the distance between two error vectors a count of the number of test cases on which those two individuals have different "eliteness" results. We chose to count the number of clusters that differed on at least 10% of the test cases in the training set; for example, if a problem has 200 test cases, we count the number of clusters that differ in binary eliteness on at least 20 test cases. While somewhat arbitrary, this distance gives a reasonable and consistent estimate

of how many groups of individuals are doing significantly different things in a given generation.

To plot counts of clusters, we used data from the same PushGP runs that produced the diversity figures in Section 4.4.1. Figures 4.20–4.27 show cluster counts over time for each of the test problems. As in Section 4.4.1, below each plot is a smaller sub-plot showing the number of successes over time for each selection to give an idea of how many data points are represented in each generation. We again aggregate data per generation across runs, indicating the range of cluster counts from the 25^{th} percentile to the 75^{th} percentile with a gray band around the median line.

The cluster count results show more variation than the corresponding error diversity figures. Lexicase selection has clearly higher cluster counts for half of the problems (Replace Space With Newline, Syllables, Scrabble Score, and Count Odds; Figures 4.20, 4.21, 4.25 and 4.27). It also starts with much higher counts on the Double Letters problem (Figure 4.24), but those numbers drop again quickly, matching tournament selection and IFS by around generation 100. On the Negative To Zero problem (Figure 4.23), the lexicase cluster counts remain small (about the same as tournament and IFS) throughout the runs. Particularly striking are lexicase cluster counts for String Lengths Backwards (Figure 4.22) and Checksum (Figure 4.26), where the number of clusters with lexicase selection is actually lower than tournament selection or IFS for significant parts of the run.

For the Count Odds problem the median is over 100 clusters for much of the run, and for Syllables the median cluster count is over 400 from generation 100 forward. For the Replace Space With Newline and Scrabble Score problems, cluster counts were also much higher for lexicase selection than tournament selection or IFS, though not as high as for the other problems. This suggests that lexicase selection is maintaining large numbers of sub-groups of the population that are capable of solving different parts of the problem. For problems with no solutions found, this might indicate that

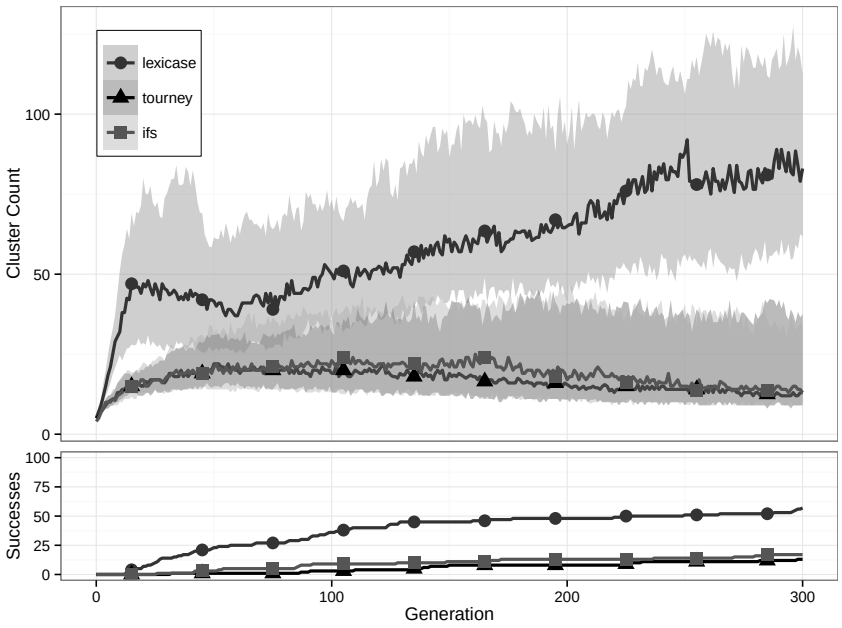

Figure 4.20. Replace Space With Newline – cluster counts median (line) and quartiles (shaded)

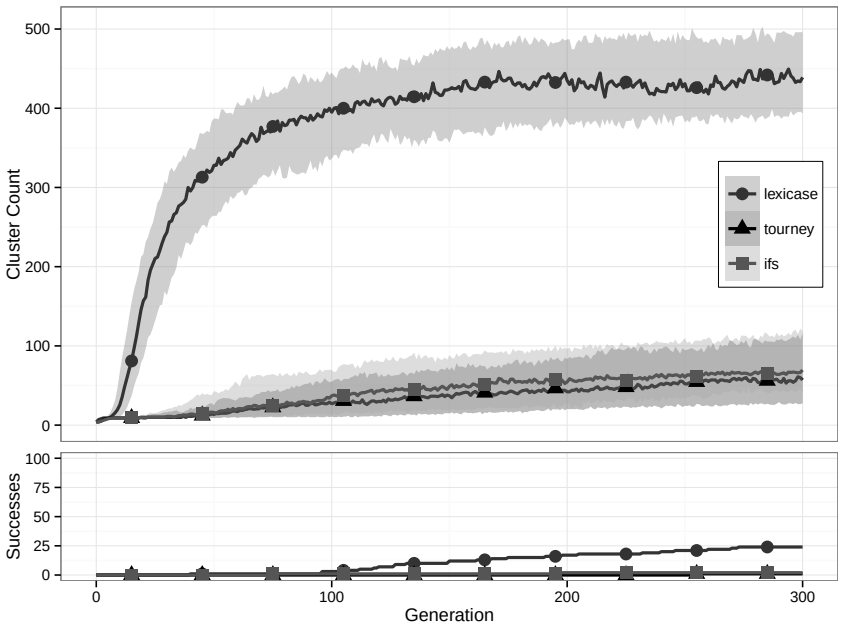

Figure 4.21. Syllables – cluster counts median (line) and quartiles (shaded)

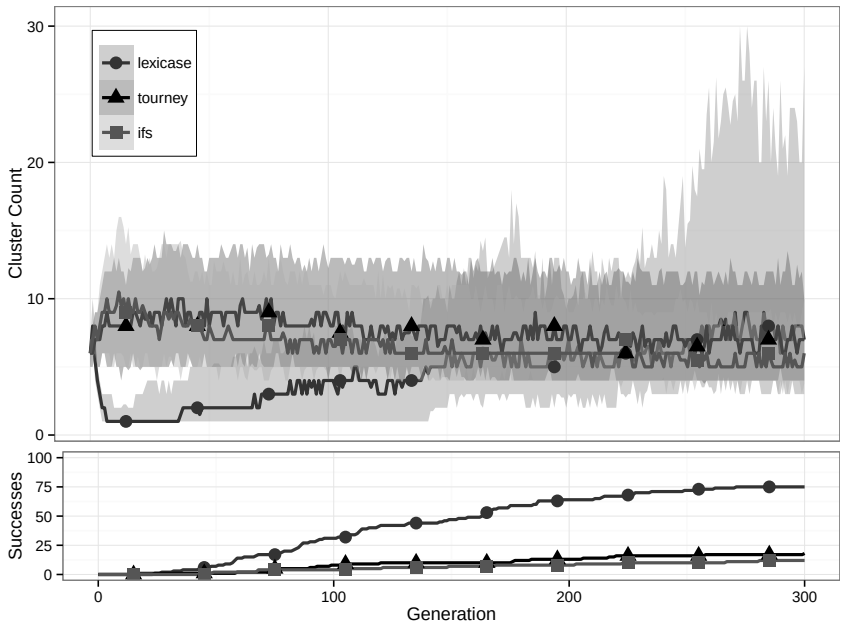

Figure 4.22. String Lengths Backwards – cluster counts median (line) and quartiles (shaded)

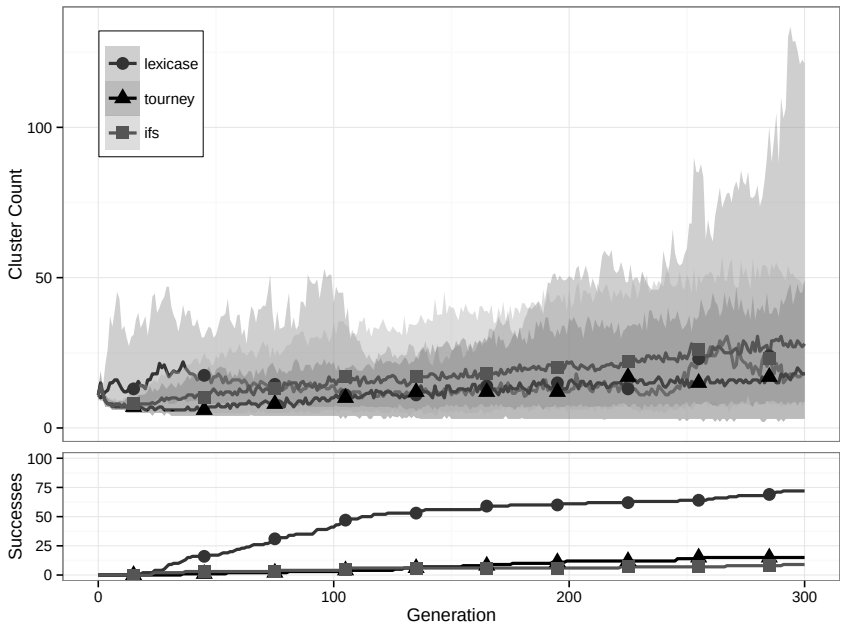

Figure 4.23. Negative To Zero – cluster counts median (line) and quartiles (shaded)

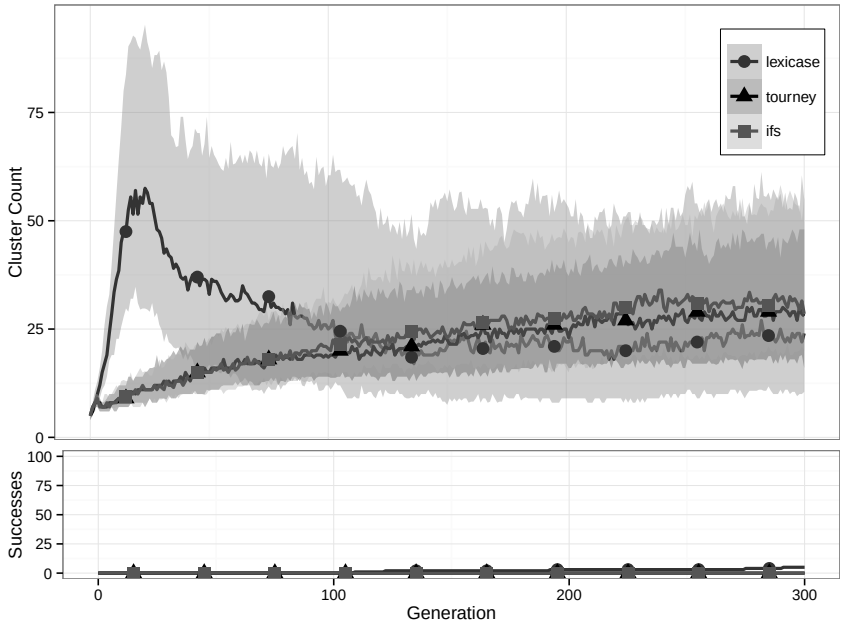

Figure 4.24. Double Letters – cluster counts median (line) and quartiles (shaded)

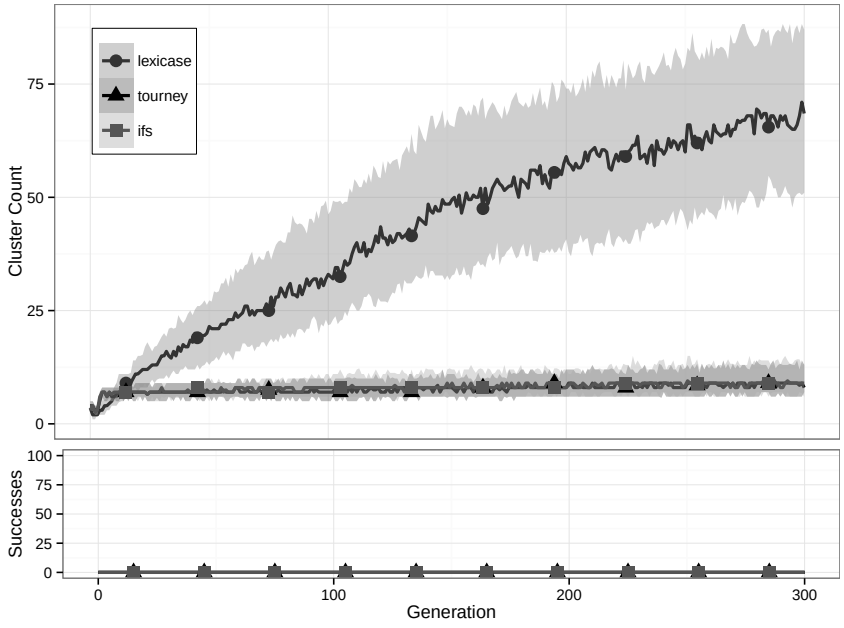

Figure 4.25. Scrabble Score – cluster counts median (line) and quartiles (shaded)

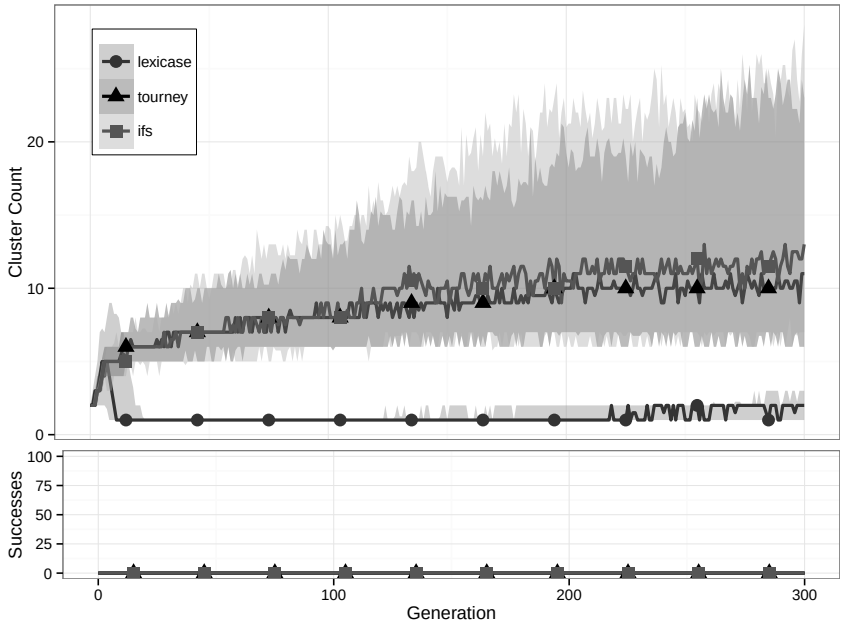

Figure 4.26. Checksum – cluster counts median (line) and quartiles (shaded)

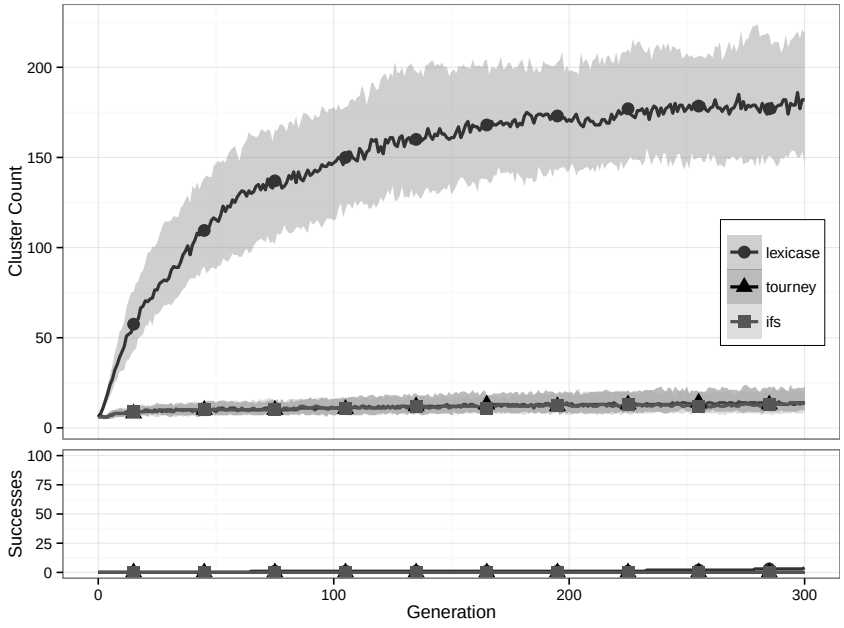

Figure 4.27. Count Odds – cluster counts median (line) and quartiles (shaded)

the genetic operators are not able to act on the structure of the programs in those sub-populations in ways that allow progress.

Interpretation of the cluster count results on the other four problems is more difficult. For example, analysis of the lexicase Checksum runs suggests that the lack of clustering might be a function of structural issues with the test cases. For this problem, there are 100 input/output examples, with two error functions per example: the Levenshtein edit distance on the printed string, and the integer difference between the ASCII values of the last character of the printed string and the correct checksum character. It appears that populations quickly evolve the ability to print `Check sum is`, but then stall, with each program printing different final characters. This allows for fairly high error diversity (over 0.75), but any given program tends to get at most two or three examples right by guessing. This means that the Manhattan distance between any two elitized error vectors is typically only 5 or 6 at most, shy of the 10% threshold of 20 for this problem, resulting in only one or two clusters. Adding examples that explore different inputs might allow evolution to first stumble upon and then exploit code that produces actual checksums for this problem.

On the set of problems explored here, error diversity seems to be a better predictor of performance than cluster counts. In fact, on two of the problems for which solutions were found in over half the runs (String Lengths Backwards and Negative To Zero), lexicase selection maintained very small numbers of clusters, similar to tournament and IFS. On problems for which lexicase selection created many more clusters than the other methods (Syllables, Scrabble Score, and Count Odds), it found many fewer solutions than it did on three of the other problems (Replace Space With Newline, String Lengths Backwards, and Negative To Zero). On the other hand, lexicase selection consistently maintained higher error diversity than other methods. This may indicate that the ability to form clusters on a problem is more indicative of the problem itself than the parent selection method and its ability to solve the problem.

This provides evidence against our hypothesis that lexicase selection performs better because it maintains clusters of individuals that genetic operators can combine to solve increasingly large numbers of test cases. This result might say more about the genetic operators we employ than about parent selection; using a different crossover operator may more readily combine programs in ways so that their children acquire both of their parents' skill sets.

4.6 Summary and Conclusions

In this chapter we described the lexicase selection algorithm and compared its utility to two common parent selection techniques. We found that lexicase selection allowed PushGP to find many more solutions than tournament selection and IFS on a range of general program synthesis benchmark problems. After anecdotally exploring a single GP run, we observed that lexicase selection seems to increase both exploration and exploitation of the evolutionary algorithm, as evidenced by increased diversity and hyper-selections of single individuals. We then conducted experiments to identify the important characteristics of lexicase selection.

Our experiments confirm our hypothesis that lexicase selection often selects specialist individuals that perform very well on some cases but poorly on others, and that these individuals, which tournament selection would very rarely select, are important in driving GP toward solutions. While lexicase sometimes selects such individuals very often as evidenced by our hyper-selection observations, it seems to matter more *which* individuals lexicase selects, and *not* that it selects them so often. These individuals contribute significantly to lexicase selection's ability to maintain high population diversity, allowing it to more widely explore the search space without sacrificing selection pressure when evolution finds promising programs.

By clustering populations into groups of individuals that perform similarly across the training set, we saw that lexicase sometimes maintains many clusters in one

population, and other times very few. As such, this clustering behavior does not seem like an important attribute of lexicase selection across all problems, though it might help in some cases. While the exploration of different genetic operators is beyond the scope of this research, future work in this area might lead to better recombination of programs specializing in different test cases and gains in performance.

CHAPTER 5

COMPARISONS TO OTHER PROGRAM SYNTHESIS SYSTEMS

In this chapter we make some initial comparisons of our results using PushGP with other program synthesis systems on our benchmark suite from Chapter 3. While we cannot exhaustively assess every program synthesis system, we hope to provide some comparisons to common synthesis systems that represent the field as a whole. We hope that others, especially those with more experience using other synthesis systems, will adopt our benchmark suite and report on their results. We will concentrate on FlashFill, MagicHaskeller, and Sketch, all popular synthesis systems.

From our survey of other program synthesis systems in Section 2.1, we expect that these systems will achieve, at best, mixed results on our benchmark problems. We presume that some of these systems will not be able to solve any of our problems simply because they cannot handle a large instruction set in a reasonable amount of time. Some of these systems recommend or require the test cases to be given in a specific order, where we provide an unordered set of test cases. It will prove interesting to see how well different systems perform with a large set of unordered test cases.

5.1 Flash Fill

Flash Fill, as discussed in Section 2.1.1, is a analytic program synthesis technique available in Microsoft Excel [25]. It was designed to help non-programmers perform repetitive tasks that would otherwise require them to write Excel macro programs.

It specializes in tasks that require string manipulations. Flash Fill is designed to work with only one or a few examples of the desired behavior; since the programs in our suite would be under-specified with such few examples, we will use the full-sized training and test sets we use with other systems. It is unclear how much use Flash Fill can make of this many examples.

We attempted to solve problems from our benchmark suite using Flash Fill. To do so, we created Excel spreadsheets that had one column per input and one column per output for each problem. Each spreadsheet included training data, which was filled in for Flash Fill to use, and unseen testing data to evaluate the results of Flash Fill. These data sets were generated using the same methods we used to generate data for our PushGP runs, described in Section 3.5.1. Since Flash Fill is deterministic and analytic, and since we lacked an automated way to import and test many data sets, we ran Flash Fill only once per problem on a single data set.

Since Excel does not include a native vector or list data structure, we were not able to find an easy way to run Flash Fill on problems that require vectors for inputs or outputs. We considered using a string representation of vectors, which seemed unsuitable. We also considered putting each item from each vector in a separate cell, but it was unclear how to handle vectors of different sizes. So, we have only tested it on the 22 problems from out suite that do not require vectors for inputs or outputs.

Of the 22 problems we tested, Flash Fill was unable to produce a program that passed the training data on 18 of them. When Flash Fill cannot find a program, it creates an error box that states:

> We looked at all the data next to your selection and didn't see a pattern for filling in values for you. To use Flash Fill, enter a couple of examples of the output you'd like to see, keep the active cell in the column you want filled in, and click the Flash Fill button again.

Flash Fill synthesized a program that passed the training data for the remaining 4 problems. On these 4 problems, we then looked at how it performed on the unseen

Table 5.1. For the four problems that Flash Fill synthesized a program from the training data, we tested the program on the unseen test set. This table gives the percent of the cases in the unseen test set that the synthesized program passed for each problem.

Problem	Test Set Percent Correct
Digits	35.4
Pig Latin	2.5
Smallest	47.8
X-Word Lines	3.6

test set to test its generalization. Flash Fill did not pass every example in the unseen test set for any of the 4 problems, and therefore had a success rate of 0 for all 22 problems on which we were able to test it. As a comparison, PushGP with lexicase selection found generalizing solutions to 22 of the 29 benchmark problems.

We additionally looked at what percent of the test set the Flash Fill program passed, which we give in Table 5.1. On two problems, Flash Fill creates a program that passes at least 35% of the unseen test set. On these problems, it seems that it was able to identify potentially relevant parts of the problem, but was not able to entirely pass the task. On the other two problems, the synthesized program passed less than 5% of the test set; on these problems, it seems that Flash Fill only is able to pass test cases with the most basic input.

Interestingly, Flash Fill performed best on Smallest, which is a number-based problem, where Flash Fill is designed to work on string manipulation tasks. We tried and failed to find a pattern in Flash Fill's outputs for this problem; it is possible that Flash Fill is interpreting the integer inputs as strings. Note that the correct output for this problem is always one of the four inputs, meaning that randomly guessing one of the four inputs to output should solve approximately 25% of the cases.

5.2 MagicHaskeller

In our survey of relevant program synthesis systems, MagicHaskeller seemed to show more promise for general-purpose synthesis than most other systems, since it makes use of a wide range of Haskell instructions [48, 46]. It can synthesize programs for a variety of input and output data types, unlike Flash Fill, which concentrates on string manipulation tasks. Still, it was unclear from the literature if MagicHaskeller would have trouble with problems that require many examples to specify a program.

Unfortunately, we were unable to install and use the stand alone version of MagicHaskeller. It appears that the system's website[1] has not been updated since 2013, and based on the documentation, we were unable to run MagicHaskeller on OS X or Windows. MagicHaskeler has a web interface[2] that is sufficient for synthesizing simple programs, but is time-limited and did not allow us to specify many examples at once.

MagicExceller, inspired by Flash Fill, uses MagicHaskeller's engine to synthesize macros in Microsoft Excel[3]. We were able to test this version on the Excel spreadsheets we created to test Flash Fill.

Of the 22 problems we ran MagicExceller on, it was not able to synthesize a program for any of them. For a few problems, it simply printed "Could not synthesize." The remaining problems encountered one of two runtime errors. On problems that use string inputs or outputs, it printed "Run-time error '28': Out of stack space." For the remaining problems that used numeric or boolean inputs and outputs, it printed "Run-time error '424': Object required". It is unclear why these errors occurred. We were able to get MagicExceller to work on the simple examples given with its docu-

[1] `http://nautilus.cs.miyazaki-u.ac.jp/~skata/MagicHaskellerLib.html`

[2] `http://nautilus.cs.miyazaki-u.ac.jp/~skata/MagicHaskeller.html`

[3] `http://nautilus.cs.miyazaki-u.ac.jp/~skata/MagicExceller.html`

mentation; it might simply not be able to handle the 50 to 200 examples we provide to specify the desired program.

5.3 Sketch[4]

Sketch is a program synthesis system that expects users to provide partially-written programs, or *sketches*, that omit some details of the program, such as integer constants or single instructions [120, 119]. Sketch can then fill in those holes with the correct details, allowing the user to skip working out the complex, error prone parts of the program. Thus, Sketch's intended use is as a programmer assistant, enabling a programmer to concentrate on the overall structure of a program instead of the small details. While Sketch is not generally intended for synthesis of programs from scratch, it may also be able to synthesize correct programs using minimal sketches.

In our experience, we were unable to get Sketch to perform synthesis based on extremely minimal sketches, such as an empty program. Instead, we found that we needed to provide significantly more structure for it to work with. For example, the sketch below represents the minimum structure we were able to provide Sketch in order to get it to synthesize a program that solves the Number IO problem:

```
float numberIOSketch(int a, float b) {
    float a2 = (float)a;
    float expression = {|((a2|b) (+|-|*|/) (a2|b))|};
    return expression;
}
```

This information greatly constrains the space of possible programs. We have specified that the integer input must be cast into a float, that the two floats must then undergo a basic arithmetic operator, and the result returned. This narrows the search space

[4]The work presented in this section was conducted with Karthik Kannappan.

significantly, down to only $2 * 4 * 2 = 16$ possible programs, which can easily be tested using brute force. Providing more generic specification, such as saying that any number of arbitrary operations may be performed on the input variables to the function, does not seem to be currently supported by Sketch. This highlights the difference in intended use between Sketch and program synthesis from scratch: a programmer writing a complex function could leave open 30 holes, leading to billions of possible programs, and Sketch should be able to fill them in, beyond what could be done quickly with brute force.

We ran into other difficulties while running Sketch, even on the simple Number IO problem, which make it not user-friendly and not viable for many of our potential applications. For instance, while running a Sketch on the Number IO problem with the default settings, we encountered the following error:

> [ERROR] [SKETCH] Error at node: You can't cast from ints to doubles/floats if you are using –fe-fpencoding AS_BIT.((float)a_5)

This issue is solvable by using a different floating-point encoding as suggested in the Sketch manual, but we consider this beyond the abilities of a standard user.

Overall, we found Sketch to not be useful for general program synthesis from examples for a variety of reasons, including that it requires a mostly-finished sketch of the desired program. This experience shows how Sketch is designed to assist programmers, not to synthesize code from scratch. It simply does not seem suitable for synthesis based entirely on input/output examples, as it requires significant additional guidance in the form of the sketch.

CHAPTER 6

SUMMARY, CONCLUSIONS, AND FUTURE WORK

To help fill a void of general program synthesis from examples problems, we have developed a benchmark suite composed of 29 problems taken from introductory computer science homework sets. These problems exhibit a range of requirements and difficulties, but all specify the desired program entirely through input/output examples of expected behavior. Our descriptions and reference implementation of these problems allows them to be useful for assessing the abilities of a wide range of systems on general program synthesis tasks. Results show that PushGP is able to solve many of the problems in this suite and meets the six requirements for general program synthesis from examples that we set out in in Section 1; to our knowledge this is the first program synthesis system to meet all six requirements.

Our exploration of lexicase selection, a parent selection technique for evolutionary computation, shows that it outperforms two conventional techniques based on aggregate fitness functions, tournament selection and IFS. Lexicase selection sets itself apart by considering one test case at a time, potentially terminating before seeing all test cases. This allows it to select individuals that specialize their efforts in some test cases while potentially performing poorly on others. Compared to the other approaches, lexicase selection manages to maintain higher population diversity while sometimes focusing many selections on promising individuals. Our systematic experiments confirm some of our hypotheses about lexicase's important characteristics while invalidating others, leading to a fuller picture of behavior-based parent selection.

We have also provided first attempts at solving problems from our benchmark suite using other, non-GP program synthesis techniques. We were unable to generate programs that generalized to unseen data using Flash Fill. While unable to run standalone versions of MagicHaskeller, we were able to use its cousin MagicExceller, which did not solve any of the benchmark problems. We found Sketch to not be the right tool for synthesizing programs from scratch based entirely on input/output examples. These representative systems had trouble with our benchmark suite; we hope others attempt these problems in other systems, and that the benchmarks help drive the creation of stronger synthesis systems.

While GP has conventionally used scalar fitness functions both as objective functions and search drivers, this work adds to increasing evidence to the benefit of alternate search drivers that take into account program behavior instead of aggregate outcomes. This lesson even applies to most multiobjective approaches in GP, in which fitness across all tests is used as a single objective alongside qualitatively different objectives such as program size. We also believe methods such as lexicase selection could prove beneficial in other forms of evolutionary computation.

Our exploration of lexicase selection has highlighted the need for approaches that take into account good performance on combinations of test cases while potentially ignoring others. Our experiments with elitist survival showed that the performance of lexicase selection usually diminishes when we remove its ability to select specialists that perform very well on some test cases but poorly on others. By allowing lexicase to select and propagate such individuals, it maintains higher population diversity and finds more solutions. We also showed that even though lexicase selection more often selects single individuals to create extreme numbers of children than other methods, this property of lexicase is not paramount to its success.

In this work we have concentrated on the effects of parent selection for general program synthesis. Our initial experiments on problems in other domains have also

shown the utility of lexicase selection [39, 36, 80, 35], though we have had mixed results when applying lexicase selection to some popular domains for genetic programming, such as symbolic regression and classification. Even if lexicase selection does not prove a panacea[1], we hope that the lessons learned here can transfer to other behavior-based parent selection techniques for these fields. For example, while other methods might not use random orderings of test cases, we would recommend that they allow for the selection of specialists.

While our clustering of the population based on test cases did not definitively show that lexicase more readily forms clusters of similar individuals, we did see that different problems showed different clustering properties. Further exploration of clustering could prove a useful tool for assessing problem difficulty and guiding the design of test cases and fitness functions to improve GP performance.

While we have provided sufficient information for others to implement our method of data generation, this obstacle might discourage others from experimenting with our benchmark problems. In the future we would like to provide a web interface from which others could download randomly generated example data for each of the benchmark problems. This would likely enable wider adoption of these problems as benchmarks for GP and other program synthesis techniques.

In our anecdotal looks into individual GP runs, we noticed that tournament selection seems to be much more susceptible to the effects of penalty error values than lexicase selection. In PushGP we have conventionally given large penalties to programs that do not return an output value, as discussed in Section 3.5.2. It is unclear how detrimental these penalties are to the progress of GP using tournament selection; on the other hand, we saw lexicase select individuals that had many penalties, leading to a successful program. We found no obvious alternative to penalties, since total

[1]Which would not be surprising considering the "no free lunch" theorem [137].

error fitness expects an error value from every test case. This may simply be one of the unintended and unavoidable consequences of aggregate fitness measures and another argument for using behavior-based parent selection. Still, finding a suitable alternative, if possible, may increase the utility of aggregate fitness approaches to parent selection, especially in problem domains where such approaches show more promise.

When a single individual receives a very large portion of the selections in a generation, diversity tends to drop since most children have that individual as a parent. We wondered if such events could catastrophically decrease population diversity when using lexicase selection. But, we have anecdotally observed rapid diversification following such events under lexicase selection. We would like to systematically investigate whether these rapid drops and climbs in diversity are common when using lexicase selection. This could give even stronger evidence to lexicase's ability to both explore and exploit the search space.

There remain unanswered questions about lexicase selection that could further differentiate which situations it will have the highest utility. For example, we have long wondered whether lexicase might be susceptible to large numbers of test cases that essentially duplicate the testing of the same functionality. Lexicase selection might then place more selection pressure on individuals that perform well on that functionality, since a representative of those test cases will be more likely to come early in the random test case ordering. Early results on this question indicate that lexicase is more robust to duplicate test cases than we had hypothesized, but more systematic testing is warranted. We have started exploring methods to neutralize potentially problematic duplication of test cases, such as by clustering similar test cases into a single case for the purposes of lexicase selection [80]; early results suggest that even these modifications might not be helpful.

Similarly, we hope to investigate whether the uniform random test case ordering used in our lexicase algorithm leads to the best results, or whether some type of biased ordering could lead to better performance. For example, we could imagine using a very different (or very similar) test case ordering for selecting the two mates for a crossover operator. We have also considered biasing the test case ordering to have easier (or harder) test cases come near the front of the list. Such efforts could potentially encourage the selection of individuals that lead to a solution without forfeiting lexicase selection's other beneficial characteristics.

BIBLIOGRAPHY

[1] Albarghouthi, Aws, Gulwani, Sumit, and Kincaid, Zachary. Recursive program synthesis. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith, Eds., vol. 8044 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 934–950.

[2] Arcuri, Andrea. On the automation of fixing software bugs. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering* (Leipzig, Germany, 2008), ACM, pp. 1003–1006. Doctoral symposium session.

[3] Arcuri, Andrea, White, David Robert, Clark, John, and Yao, Xin. Multi-objective improvement of software using co-evolution and smart seeding. In *Proceedings of the 7th International Conference on Simulated Evolution And Learning (SEAL '08)* (Melbourne, Australia, Dec. 7-10 2008), vol. 5361 of *Lecture Notes in Computer Science*, Springer, pp. 61–70.

[4] Arcuri, Andrea, and Yao, Xin. Coevolving programs and unit tests from their specification. In *IEEE International Conference on Automated Software Engineering (ASE)* (Atlanta, Georgia, USA, Nov. 5-9 2007).

[5] Arcuri, Andrea, and Yao, Xin. A novel co-evolutionary approach to automatic software bug fixing. In *2008 IEEE World Congress on Computational Intelligence* (Hong Kong, 1-6 June 2008), Jun Wang, Ed., IEEE Computational Intelligence Society, IEEE Press, pp. 162–168.

[6] Arcuri, Andrea, and Yao, Xin. Co-evolutionary automatic programming for software development. *Information Sciences 259* (2014), 412–432.

[7] Avizienis, A. The N-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng. 11*, 12 (Dec. 1985), 1491–1501.

[8] Bäck, Thomas. Selective pressure in evolutionary algorithms: a characterization of selection mechanisms. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on* (Jun 1994), pp. 57–62 vol.1.

[9] Binard, Franck, and Felty, Amy. An abstraction-based genetic programming system. In *Late breaking paper at Genetic and Evolutionary Computation Conference (GECCO'2007)* (London, United Kingdom, 7-11 July 2007), Peter A. N. Bosman, Ed., ACM Press, pp. 2415–2422.

[10] Binard, Franck, and Felty, Amy. Genetic programming with polymorphic types and higher-order functions. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation* (Atlanta, GA, USA, 12-16 July 2008), ACM, pp. 1187–1194.

[11] Bleuler, S., Brack, M., Thiele, L., and Zitzler, E. Multiobjective genetic programming: reducing bloat using spea2. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on* (2001), vol. 1, pp. 536–543 vol. 1.

[12] Blickle, Tobias, and Thiele, Lothar. A mathematical analysis of tournament selection. In *Proceedings of the 6th International Conference on Genetic Algorithms* (San Francisco, CA, USA, 1995), Morgan Kaufmann Publishers Inc., pp. 9–16.

[13] Bodik, Rastislav, and Jobstmann, Barbara. Algorithmic program synthesis: introduction. *International Journal on Software Tools for Technology Transfer 15*, 5-6 (2013), 397–411.

[14] Brun, Yuriy, Barr, Earl, Xiao, Ming, Le Goues, Claire, and Devanbu, Prem. Evolution vs. intelligent design in program patching. Tech. Rep. `https://escholarship.org/uc/item/3z8926ks`, UC Davis: College of Engineering, 2013.

[15] Cumming, Geoff. The new statistics: Why and how. *Psychological Science 25*, 1 (2014), 7–29.

[16] Cypher, A., and Halbert, D.C. *Watch what I Do: Programming by Demonstration*. MIT Press, 1993.

[17] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *Evolutionary Computation, IEEE Transactions on 6*, 2 (2002), 182–197.

[18] Ekart, Aniko, and Nemeth, S. Z. Selection based on the pareto nondomination criterion for controlling code growth in genetic programming. *Genetic Programming and Evolvable Machines 2*, 1 (Mar. 2001), 61–73.

[19] Feldt, Robert. Generating multiple diverse software versions with genetic programming. In *Proceedings of the 24th EUROMICRO Conference, Workshop on Dependable Computing Systems* (Vaesteraas, Sweden, 25-27th Aug. 1998), pp. 387–396.

[20] Fieldsend, Jonathan E., and Moraglio, Alberto. Strength through diversity: Disaggregation and multi-objectivisation approaches for genetic programming. In *GECCO '15: Proceedings of the 2015 conference on Genetic and evolutionary computation* (2015), ACM.

[21] Flener, Pierre, and Schmid, Ute. An introduction to inductive programming. *Artificial Intelligence Review 29*, 1 (2008), 45–62.

[22] Galván-López, Edgar, Cody-Kenny, Brendan, Trujillo, Leonardo, and Kattan, Ahmed. Using semantics in the selection mechanism in genetic programming: a simple method for promoting semantic diversity. In *2013 IEEE Conference on Evolutionary Computation* (Cancun, Mexico, June 20-23 2013), Luis Gerardo de la Fraga, Ed., vol. 1, pp. 2972–2979.

[23] Goldberg, David E., and Richardson, Jon. 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, 1987), L. Erlbaum Associates Inc., pp. 41–49.

[24] Gulwani, Sumit. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming* (2010), ACM, pp. 13–24.

[25] Gulwani, Sumit. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2011), POPL '11, ACM, pp. 317–330.

[26] Gulwani, Sumit. Synthesis from examples: Interaction models and algorithms. In *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (2012). Invited talk paper.

[27] Gulwani, Sumit, Harris, William R., and Singh, Rishabh. Spreadsheet data manipulation using examples. *Commun. ACM 55*, 8 (Aug. 2012), 97–105.

[28] Harding, Simon, Graziano, Vincent, Leitner, Juergen, and Schmidhuber, Juergen. MT-CGP: mixed type cartesian genetic programming. In *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference* (Philadelphia, Pennsylvania, USA, 7-11 July 2012), ACM, pp. 751–758.

[29] Harman, Mark, Langdon, William B., Jia, Yue, White, David R., Arcuri, Andrea, and Clark, John A. The GISMOE challenge: Constructing the Pareto program surface using genetic programming to find better programs. In *The 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 12)* (Essen, Germany, Sept. 3-7 2012), ACM, pp. 1–14.

[30] Harper, Robin. Spatial co-evolution: quicker, fitter and less bloated. In *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference* (Philadelphia, Pennsylvania, USA, 7-11 July 2012), ACM, pp. 759–766.

[31] Harrington, Kyle I., Spector, Lee, Pollack, Jordan B., and O'Reilly, Una-May. Autoconstructive evolution for structural problems. In *GECCO 2012 2nd Workshop on Evolutionary Computation for the Automated Design of Algorithms* (Philadelphia, Pennsylvania, USA, 7-11 July 2012), Gisele L. Pappa, John Woodward, Matthew R. Hyde, and Jerry Swan, Eds., ACM, pp. 75–82.

[32] Harris, William R, and Gulwani, Sumit. Spreadsheet table transformations from examples. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 317–328.

[33] Helmuth, Thomas, McPhee, Nicholas Freitag, and Spector, Lee. Lexicase selection for program synthesis: a diversity analysis. In *Genetic Programming Theory and Practice XIII*, Genetic and Evolutionary Computation. Springer, 2015.

[34] Helmuth, Thomas, and Spector, Lee. Evolving SQL queries from examples with developmental genetic programming. In *Genetic Programming Theory and Practice X*, Rick Riolo, Ekaterina Vladislavleva, Marylyn D. Ritchie, and Jason H. Moore, Eds., Genetic and Evolutionary Computation. Springer, Ann Arbor, USA, 12-14 May 2012, ch. 1, pp. 1–14.

[35] Helmuth, Thomas, and Spector, Lee. 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, 6-10 July 2013), ACM, pp. 1627–1634.

[36] Helmuth, Thomas, and Spector, Lee. 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, 12-16 July 2014), ACM, pp. 919–926.

[37] Helmuth, Thomas, and Spector, Lee. Detailed problem descriptions for general program synthesis benchmark suite. Technical Report UM-CS-2015-006, Computer Science, University of Massachusetts, Amherst, June 2015.

[38] Helmuth, Thomas, and Spector, Lee. General program synthesis benchmark suite. In *GECCO '15: Proceedings of the 2015 Conference on Genetic and Evolutionary Computation* (July 2015).

[39] Helmuth, Thomas, Spector, Lee, and Matheson, James. Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation* (2014). Accepted for future publication.

[40] Hofmann, Martin, Kitzelmann, Emanuel, and Schmid, Ute. A unifying framework for analysis and evaluation of inductive programming systems. In *Proceedings of the Second Conference on Artificial General Intelligence* (2009), Citeseer, pp. 55–60.

[41] Hollander, M., and Wolfe, D.A. *Nonparametric Statistical Methods*. Wiley Series in Probability and Statistics. Wiley, 1999.

[42] Horn, Jeffrey, Nafpliotis, Nicholas, and Goldberg, David E. Multiobjective optimization using the niched pareto genetic algorithm. Tech. Rep. IlliGAL 93005, University of Illinois at Urbana-Champaign, 104 South Mathews Avenue, Urbana, IL 61801, 1993.

[43] Jackson, David. Promoting phenotypic diversity in genetic programming. In *PPSN 2010 11th International Conference on Parallel Problem Solving From Nature* (Krakow, Poland, 11-15 Sept. 2010), Robert Schaefer, Carlos Cotta, Joanna Kolodziej, and Guenter Rudolph, Eds., vol. 6239 of *Lecture Notes in Computer Science*, Springer, pp. 472–481.

[44] Jazayeri, Mehdi. Formal specification and automatic programming. In *Proceedings of the 2Nd International Conference on Software Engineering* (Los Alamitos, CA, USA, 1976), ICSE '76, IEEE Computer Society Press, pp. 293–296.

[45] Kannappan, Karthik, Spector, Lee, Sipper, Moshe, Helmuth, Thomas, La Cava, William, Wisdom, Jake, and Bernstein, Omri. Analyzing a decade of human-competitive (HUMIE) winners:what can we learn? In *Genetic Programming Theory and Practice XII*, Genetic and Evolutionary Computation. Springer, Ann Arbor, USA, May 2014.

[46] Katayama, Susumu. Systematic search for lambda expressions. In *Trends in Functional Programming* (2005), Marko C. J. D. van Eekelen, Ed., vol. 6 of *Trends in Functional Programming*, Intellect, pp. 111–126.

[47] Katayama, Susumu. Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In *PRICAI 2008: Trends in Artificial Intelligence*, Tu-Bao Ho and Zhi-Hua Zhou, Eds., vol. 5351 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 199–210.

[48] Katayama, Susumu. Recent improvements of magichaskeller. In *Approaches and Applications of Inductive Programming*, Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, Eds., vol. 5812 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 174–193.

[49] Katayama, Susumu. An analytical inductive functional programming system that avoids unintended programs. In *Proceedings of AAIP 2011 4th International Workshop on Approaches and Applications of Inductive Programming* (2011).

[50] Katayama, Susumu. Magichaskeller: System demonstration. In *Proceedings of AAIP 2011 4th International Workshop on Approaches and Applications of Inductive Programming* (2011).

[51] Katayama, Susumu. Magichaskeller on the web: Automated programming as a service. In *Haskell '13: Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell* (2013), ACM.

[52] Keijzer, Maarten. Push-forth: a light-weight, strongly-typed, stack-based genetic programming language. In *GECCO '13 Companion: Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion* (Amsterdam, The Netherlands, 6-10 July 2013), ACM, pp. 1635–1640.

[53] Kitzelmann, Emanuel. Data-driven induction of recursive functions from input/output-examples. In *Proceedings of the ECML/PKDD 2007 Workshop on Approaches and Applications of Inductive Programming (AAIP 2007)* (2007), pp. 15–26.

[54] Kitzelmann, Emanuel. Analytical inductive functional programming. In *Logic-Based Program Synthesis and Transformation*, Michael Hanus, Ed., vol. 5438 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, pp. 87–102.

[55] Kitzelmann, Emanuel. Inductive programming: A survey of program synthesis techniques. In *Approaches and Applications of Inductive Programming*, Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, Eds., vol. 5812 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 50–73.

[56] Kitzelmann, Emanuel. Two new operators for IGOR2 to increase synthesis efficieny. In *Proceedings of AAIP 2011 4th International Workshop on Approaches and Applications of Inductive Programming* (2011).

[57] Kitzelmann, Emanuel, and Schmid, Ute. Inductive synthesis of functional programs: An explanation based generalization approach. *J. Mach. Learn. Res. 7* (Dec. 2006), 429–454.

[58] Klein, Jon, and Spector, Lee. Genetic programming with historically assessed hardness. In *Genetic Programming Theory and Practice VI*, Rick L. Riolo, Terence Soule, and Bill Worzel, Eds., Genetic and Evolutionary Computation. Springer, Ann Arbor, 15-17 May 2008, ch. 5, pp. 61–75.

[59] Knowles, Joshua D., Watson, Richard A., and Corne, David. Reducing local optima in single-objective problems by multi-objectivization. In *Proceedings of the First International Conference on Evolutionary Multi-Criterion Optimization* (London, UK, UK, 2001), EMO '01, Springer-Verlag, pp. 269–283.

[60] Kotanchek, Mark, Smits, Guido, and Vladislavleva, Ekaterina. Pursuing the pareto paradigm: Tournaments, algorithm variations and ordinal optimization. In *Genetic Programming Theory and Practice IV*, Rick Riolo, Terence Soule, and Bill Worzel, Eds., Genetic and Evolutionary Computation. Springer US, 2007, pp. 167–185.

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

[62] Krawiec, Krzysztof, and Lichocki, Pawel. Using co-solvability to model and exploit synergetic effects in evolution. In *PPSN 2010 11th International Conference on Parallel Problem Solving From Nature* (Krakow, Poland, 11-15 Sept. 2010), Robert Schaefer, Carlos Cotta, Joanna Kolodziej, and Guenter Rudolph, Eds., vol. 6239 of *Lecture Notes in Computer Science*, Springer, pp. 492–501.

[63] Krawiec, Krzysztof, and Liskowski, Pawel. Automatic derivation of search objectives for test-based genetic programming. In *18th European Conference on Genetic Programming* (Copenhagen, 8-10 Apr. 2015), Penousal Machado, Malcolm I. Heywood, James McDermott, Mauro Castelli, Pablo Garcia-Sanchez, Paolo Burelli, Sebastian Risi, and Kevin Sim, Eds., vol. 9025 of *LNCS*, Springer, pp. 53–65.

[64] Krawiec, Krzysztof, and Nawrocki, Mateusz. Implicit fitness sharing for evolutionary synthesis of license plate detectors. In *Applications of Evolutionary Computing* (Vienna, Austria, 3-5 Apr. 2013), vol. 7835 of *Lecture Notes in Computer Science*, Springer, pp. 376–386.

[65] Krawiec, Krzysztof, Swan, Jerry, and O'Reilly, Una-May. Behavioral program synthesis: Insights and prospects. In *Genetic Programming Theory and Practice XIII*, Genetic and Evolutionary Computation. Springer, 2015.

[66] Langdon, William B. Evolving data structures with genetic programming. In *Proceedings of the 6th International Conference on Genetic Algorithms* (San Francisco, CA, USA, 1995), Morgan Kaufmann Publishers Inc., pp. 295–302.

[67] Langdon, William B. Advances in genetic programming. MIT Press, Cambridge, MA, USA, 1996, ch. Data structures and genetic programming, pp. 395–414.

[68] Langdon, William B., and Harman, Mark. Genetically improving 50000 lines of C++. Research Note RN/12/09, Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK, 19 Sept. 2012.

[69] Langdon, William B., and Harman, Mark. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation 19*, 1 (Feb. 2015), 118–135.

[70] Lau, Tessa, Bergman, Lawrence, Castelli, Vittorio, and Oblinger, Daniel. Programming shell scripts by demonstration. In *Workshop on Supervisory Control of Learning and Adaptive Systems, AAAI* (2004), vol. 4.

[71] Lau, Tessa, Domingos, Pedro, and Weld, Daniel S. Learning programs from traces using version space algebra. In *Proceedings of the 2Nd International Conference on Knowledge Capture* (New York, NY, USA, 2003), K-CAP '03, ACM, pp. 36–43.

[72] Lau, Tessa, Wolfman, Steven A, Domingos, Pedro, and Weld, Daniel S. Programming by demonstration using version space algebra. *Machine Learning 53*, 1-2 (2003), 111–156.

[73] Le Goues, Claire, Dewey-Vogt, Michael, Forrest, Stephanie, and Weimer, Westley. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *34th International Conference on Software Engineering (ICSE 2012)* (Zurich, June 2-9 2012), Martin Glinz, Ed., pp. 3–13.

[74] Le Goues, Claire, Forrest, Stephanie, and Weimer, Westley. Current challenges in automatic software repair. *Software Quality Journal 21* (Sept. 2013), 421–443.

[75] Le Goues, Claire, Holtschulte, Neal, Smith, Edward K., Brun, Yuriy, Devanbu, Premkumar, Forrest, Stephanie, and Weimer, Westley. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering (TSE), in press, 22 pages* (2015).

[76] Le Goues, Claire, Nguyen, ThanhVu, Forrest, Stephanie, and Weimer, Westley. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering 38*, 1 (Jan.-Feb. 2012), 54–72.

[77] Le Goues, Claire, Weimer, Westley, and Forrest, Stephanie. Representations and operators for improving evolutionary software repair. In *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference* (Philadelphia, Pennsylvania, USA, 7-11 July 2012), ACM, pp. 959–966.

[78] Liang, Percy, Jordan, Michael I, and Klein, Dan. Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)* (2010), pp. 639–646.

[79] Lieberman, Henry. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[80] Liskowski, Paweł, Krawiec, Krzysztof, Helmuth, Thomas, and Spector, Lee. Comparison of semantic-aware selection methods in genetic programming. In *GECCO 2015 workshop on Semantic Methods in Genetic Programming* (2015), ACM.

[81] Luke, Sean. *Essentials of Metaheuristics*, first ed. lulu.com, 2009. Available at http://cs.gmu.edu/∼sean/books/metaheuristics/.

[82] Luke, Sean, and Panait, Liviu. Is the perfect the enemy of the good? In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference* (New York, 9-13 July 2002), Morgan Kaufmann Publishers, pp. 820–828.

[83] Maechler, Martin, Rousseeuw, Peter, Struyf, Anja, Hubert, Mia, and Hornik, Kurt. *cluster: Cluster Analysis Basics and Extensions*, 2014. R package version 1.15.3.

[84] Manna, Zohar, and Waldinger, Richard J. Toward automatic program synthesis. *Commun. ACM 14*, 3 (Mar. 1971), 151–165.

[85] McDermott, James, White, David R., Luke, Sean, Manzoni, Luca, Castelli, Mauro, Vanneschi, Leonardo, Jaskowski, Wojciech, Krawiec, Krzysztof, Harper, Robin, De Jong, Kenneth, and O'Reilly, Una-May. Genetic programming needs better benchmarks. In *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference* (Philadelphia, Pennsylvania, USA, 7-11 July 2012), ACM, pp. 791–798.

[86] McKay, Robert I. Fitness sharing in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)* (Las Vegas, Nevada, USA, 10-12 July 2000), Darrell Whitley, David Goldberg, Erick Cantu-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer, Eds., Morgan Kaufmann, pp. 435–442.

[87] McKay, Robert I., Hoai, Nguyen Xuan, Whigham, Peter Alexander, Shan, Yin, and ONeill, Michael. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines 11*, 3-4 (2010), 365–396.

[88] McPhee, Nicholas Freitag, Donatucci, David, and Helmuth, Thomas. Using graph databases to explore the dynamics of genetic programming runs. In *Genetic Programming Theory and Practice XIII*, Genetic and Evolutionary Computation. Springer, 2015.

[89] Menon, Aditya Krishna, Tamuz, Omer, Gulwani, Sumit, Lampson, Butler W., and Kalai, Adam. A machine learning framework for programming by example. In *ICML (1)* (2013), vol. 28 of *JMLR Proceedings*, JMLR.org, pp. 187–195.

[90] Moll, Robert. iJava—an online interactive textbook for elementary Java instruction: Demonstration. *Journal of Computing Sciences in Colleges 26*, 6 (June 2011), 55–57.

[91] Moll, Robert. iJava. `http://ijava.cs.umass.edu/index.html`, 2014. Edition 3.1. Online; accessed September 2015.

[92] Montana, David J. Strongly typed genetic programming. *Evolutionary computation 3*, 2 (1995), 199–230.

[93] Moraglio, Alberto, Krawiec, Krzysztof, and Johnson, Colin G. Geometric semantic genetic programming. In *Parallel Problem Solving from Nature, PPSN XII (part 1)* (Taormina, Italy, Sept. 1-5 2012), vol. 7491 of *Lecture Notes in Computer Science*, Springer, pp. 21–31.

[94] Moraglio, Alberto, Otero, Fernando, Johnson, Colin, Thompson, Simon, and Freitas, Alex. Evolving recursive programs using non-recursive scaffolding. In *Proceedings of the 2012 IEEE Congress on Evolutionary Computation* (Brisbane, Australia, 10-15 June 2012), Xiaodong Li, Ed., pp. 2242–2249.

[95] Muggleton, Stephen, De Raedt, Luc, Poole, David, Bratko, Ivan, Flach, Peter, Inoue, Katsumi, and Srinivasan, Ashwin. Ilp turns 20. *Machine Learning 86*, 1 (2012), 3–23.

[96] Nakazawa, Minato. *fmsb: Functions for medical statistics book with some demographic data*, 2014. R package.

[97] Noble, Jason, and Watson, Richard A. 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* (2001), Morgan Kaufmann, pp. 493–500.

[98] Olsson, J Roland. Inductive functional programming using incremental program transformation. *Artificial intelligence 74*, 1 (1995), 55–81.

[99] Olsson, J Roland. The art of writing specifications for the adate automatic programming system. In *Proceedings of the 3rd Annual Conference on Genetic Programming* (1998), Citeseer, pp. 278–283.

[100] Olsson, J Roland. Population management for automatic design of algorithms through evolution. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on* (1998), IEEE, pp. 592–597.

[101] O'Neill, Michael, and Ryan, Conor. Grammatical evolution. *IEEE Transactions on Evolutionary Computation 5*, 4 (Aug. 2001), 349–358.

[102] O'Neill, Michael, and Ryan, Conor. *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, vol. 4 of *Genetic programming*. Kluwer Academic Publishers, 2003.

[103] Orlov, Michael, and Sipper, Moshe. Genetic programming in the wild: Evolving unrestricted bytecode. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation* (Montreal, 8-12 July 2009), ACM, pp. 1043–1050.

[104] Orlov, Michael, and Sipper, Moshe. FINCH: A system for evolving Java (bytecode). In *Genetic Programming Theory and Practice VIII*, Rick Riolo, Trent McConaghy, and Ekaterina Vladislavleva, Eds., vol. 8 of *Genetic and Evolutionary Computation*. Springer, Ann Arbor, USA, 20-22 May 2010, ch. 1, pp. 1–16.

[105] Orlov, Michael, and Sipper, Moshe. Flight of the FINCH through the Java wilderness. *IEEE Transactions on Evolutionary Computation 15*, 2 (Apr. 2011), 166–182.

[106] Perelman, Daniel, Gulwani, Sumit, Grossman, Dan, and Provost, Peter. Test-driven synthesis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), ACM, p. 43.

[107] Poli, Riccardo, Langdon, William B., and McPhee, Nicholas Freitag. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[108] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. ISBN 3-900051-07-0.

[109] Schmidt, Michael D., and Lipson, Hod. Co-evolving fitness predictors for accelerating and reducing evaluations. In *Genetic Programming Theory and Practice IV*, vol. 5 of *Genetic and Evolutionary Computation*. Springer, Ann Arbor, 11-13 May 2006, pp. 113–130.

[110] Schmidt, Michael D., and Lipson, Hod. Coevolution of fitness predictors. *IEEE Transactions on Evolutionary Computation 12*, 6 (Dec. 2008), 736–749.

[111] Schmidt, Michael D., and Lipson, Hod. Predicting solution rank to improve performance. In *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation* (Portland, Oregon, USA, 7-11 July 2010), ACM, pp. 949–956.

[112] Shaw, David Elliot, Swartout, William R, and Green, C Cordell. Inferring lisp programs from examples. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence* (1975), Morgan Kaufmann.

[113] Singh, Rishabh, and Gulwani, Sumit. Learning semantic string transformations from examples. *Proc. VLDB Endow. 5*, 8 (Apr. 2012), 740–751.

[114] Singh, Rishabh, and Gulwani, Sumit. Synthesizing number transformations from input-output examples. In *Computer Aided Verification* (2012), Springer, pp. 634–651.

[115] Skolicki, Zbigniew. An analysis of island models in evolutionary computation. In *Proceedings of the 7th Annual Workshop on Genetic and Evolutionary Computation* (New York, NY, USA, 2005), GECCO '05, ACM, pp. 386–389.

[116] Smith, Edward K, Barr, Earl T, Le Goues, Claire, and Brun, Yuriy. Is the cure worse than the disease? overfitting in automated program repair. In *ESEC/FSE* (2015), ACM.

[117] Smith, Robert, Forrest, Stephanie, and Perelson, Alan S. Population diversity in an immune system model: Implications for genetic search. In *Foundations of Genetic Algorithms 2* (1992), Morgan Kaufmann, pp. 153–166.

[118] Smits, Guido F., and Kotanchek, Mark. Pareto-front exploitation in symbolic regression. In *Genetic Programming Theory and Practice II*, Una-May OReilly, Tina Yu, Rick Riolo, and Bill Worzel, Eds., vol. 8 of *Genetic Programming*. Springer US, 2005, pp. 283–299.

[119] Solar-Lezama, Armando, Arnold, Gilad, Tancau, Liviu, Bodik, Rastislav, Saraswat, Vijay, and Seshia, Sanjit. Sketching stencils. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 167–178.

[120] Solar-Lezama, Armando, Jones, Christopher Grant, and Bodik, Rastislav. Sketching concurrent data structures. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2008), PLDI '08, ACM, pp. 136–148.

[121] Spector, Lee. Autoconstructive evolution: Push, pushGP, and pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)* (San Francisco, California, USA, 7-11 July 2001), Morgan Kaufmann, pp. 137–146.

[122] Spector, Lee. Towards practical autoconstructive evolution: Self-evolution of problem-solving genetic programming systems. In *Genetic Programming Theory and Practice VIII*, Rick Riolo, Trent McConaghy, and Ekaterina Vladislavleva, Eds., vol. 8 of *Genetic and Evolutionary Computation*. Springer, Ann Arbor, USA, 20-22 May 2010, ch. 2, pp. 17–33.

[123] Spector, Lee. 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* (New York, NY, USA, 2012), GECCO Companion '12, ACM, pp. 401–408.

[124] Spector, Lee, and Helmuth, Thomas. Uniform linear transformation with repair and alternation in genetic programming. In *Genetic Programming Theory and Practice XI*, Rick Riolo, Jason H. Moore, and Mark Kotanchek, Eds., Genetic and Evolutionary Computation. Springer, Ann Arbor, USA, 9-11 May 2013, ch. 8, pp. 137–153.

[125] Spector, Lee, and Helmuth, Thomas. Effective simplification of evolved Push programs using a simple, stochastic hill-climber. In *GECCO Companion '14* (Vancouver, BC, Canada, 12-16 July 2014), ACM, pp. 147–148.

[126] Spector, Lee, Klein, Jon, and Keijzer, Maarten. The Push3 execution stack and the evolution of control. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation* (Washington DC, USA, 25-29 June 2005), vol. 2, ACM Press, pp. 1689–1696.

[127] Spector, Lee, Martin, Brian, Harrington, Kyle, and Helmuth, Thomas. Tag-based modules in genetic programming. In *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation* (Dublin, Ireland, 12-16 July 2011), ACM, pp. 1419–1426.

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

[129] Starosta, S. *The Case for Lexicase: An Outline of Lexicase Grammatical Theory*. Open linguistics series. Pinter Publishers, 1988.

[130] Starosta, Stanley, and Nomura, Hirosato. Lexicase parsing: A lexicon-driven approach to syntactic analysis. In *Proceedings of the 11th Coference on Computational Linguistics* (Stroudsburg, PA, USA, 1986), COLING '86, Association for Computational Linguistics, pp. 127–132.

[131] Summers, Phillip D. A methodology for lisp program construction from examples. *J. ACM 24*, 1 (Jan. 1977), 161–175.

[132] Teller, A. Turing completeness in the language of genetic programming with indexed memory. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on* (Jun 1994), pp. 136–141 vol.1.

[133] Weimer, Westley, Nguyen, ThanhVu, Le Goues, Claire, and Forrest, Stephanie. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE) 2009* (Vancouver, May 16-24 2009), Stephen Fickas, Ed., pp. 364–374.

[134] White, David R., Arcuri, Andrea, and Clark, John A. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation 15*, 4 (Aug. 2011), 515–538.

[135] White, David R., Mcdermott, James, Castelli, Mauro, Manzoni, Luca, Goldman, Brian W., Kronberger, Gabriel, Jaśkowski, Wojciech, O'Reilly, Una-May, and Luke, Sean. Better GP benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines 14*, 1 (Mar. 2013), 3–29.

[136] Withall, M. S., Hinde, C. J., and Stone, R. G. An improved representation for evolving programs. *Genetic Programming and Evolvable Machines 10*, 1 (Mar. 2009), 37–70.

[137] Wolpert, D.H., and Macready, W.G. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on 1*, 1 (1997), 67–82.

[138] Woodward, John, Martin, Simon, and Swan, Jerry. Benchmarks that matter for genetic programming. In *GECCO 2014 4th workshop on evolutionary computation for the automated design of algorithms* (Vancouver, BC, Canada, 12-16 July 2014), ACM, pp. 1397–1404.

[139] Woodward, John R. Evolving turing complete representations. In *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on* (2003), vol. 2, IEEE, pp. 830–837.

[140] Yu, Tina. A higher-order function approach to evolve recursive programs. In *Genetic Programming Theory and Practice III*, Tina Yu, Rick L. Riolo, and Bill Worzel, Eds., vol. 9 of *Genetic Programming*. Springer, Ann Arbor, 12-14 May 2005, ch. 7, pp. 93–108.

[141] Zitzler, Eckart, Laumanns, Marco, and Thiele, Lothar. Spea2: Improving the strength pareto evolutionary algorithm. Tech. Rep. 103, Swiss Federal Institute of Technology (ETH) Zurich, 2001.