Lexicase Selection for Program Synthesis: A Diversity Analysis

Chapter · December 2016

DOI: 10.1007/978-3-319-34223-8_9

3 authors, including:

Nicholas Mcphee
University of Minnesota Morris
67 PUBLICATIONS **3,182** CITATIONS
SEE PROFILE

Lee Spector
Hampshire College
197 PUBLICATIONS **5,078** CITATIONS
SEE PROFILE

Preprint: To appear in: Genetic Programming Theory and Practice XIII (GPTP 2015). Springer.

Lexicase selection for program synthesis: a diversity analysis

Thomas Helmuth, Nicholas Freitag McPhee, Lee Spector

Abstract Lexicase selection is a selection method for evolutionary computation in which individuals are selected by filtering the population according to performance on test cases, considered in random order. When used as the parent selection method in genetic programming, lexicase selection has been shown to provide significant improvements in problem-solving power. In this chapter we investigate the reasons for the success of lexicase selection, focusing on measures of population diversity. We present data from eight program synthesis problems and compare lexicase selection to tournament selection and selection based on implicit fitness sharing. We conclude that lexicase selection does indeed produce more diverse populations, which helps to explain the utility of lexicase selection for program synthesis.

Key words: Lexicase selection, diversity, tournament selection, implicit fitness sharing.

1 Introduction

Lexicase selection is a recently developed parent selection method for evolutionary computation in which individuals are selected by filtering the population according to performance on individual fitness cases, considered in random order (Spector, 2012). Lexicase selection, when used as the parent selection method in genetic programming, has been shown to provide significant improvements in terms of problem-solving power (Helmuth et al, 2014; Helmuth and Spector, 2015). In this

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

Nicholas Freitag McPhee
Division of Science and Mathematics, University of Minnesota, Morris, MN USA

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

chapter we investigate the reasons for the success of lexicase selection, focusing in particular on the ways in which lexicase selection seems to help maintain population diversity. We present data from eight program synthesis problems and compare lexicase selection, in terms of problem solving power and diversity, to tournament selection and selection based on implicit fitness sharing (IFS). IFS distributes reward among the individuals that solve a test case, giving more reward for cases solved by fewer individuals (McKay, 2000); for more detail see (Helmuth et al, 2014).

For each parent selection event lexicase selection randomly orders the test cases and then removes any individuals that do not have the best performance on the first case. If more than one individual remains then those that do not have the best performance on the second case are also removed. This continues until only one individual remains and is selected, or until all cases have been used, in which case one of the remaining individuals is selected randomly. Key properties of lexicase selection are that (a) it avoids combining all errors into a single scalar fitness value, (b) because of the random ordering of test cases, every test case will be most important (first to be considered) at least occasionally, and (c) similarly, each pair of test cases, and each triple, etc., will be most important at least occasionally.

We investigate the relations between selection methods and population diversity using two measures of diversity: error diversity and cluster counts. We find that lexicase selection runs have consistently higher error diversity than tournament selection and IFS across all generations and all problems. The cluster counts for lexicase selection are also generally higher, but less consistently. We conclude that lexicase selection does indeed produce more diverse populations, which helps to explain the utility of lexicase selection for program synthesis.

2 Diversity Measures

To evaluate a program in program synthesis, we run it on a set of test cases composed of input/output pairs, creating a behavior vector of its outputs. Then, we apply one or more error functions to each desired output and the program's output, creating an error vector for each individual. 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 (Jackson, 2010). 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 vector, but two different error vectors must come from different behavior vectors. Helmuth et al (2014) showed that lexicase selection maintained higher diversity than tournament and IFS selection on three problems.

One hypothesis we have put forth regarding the improved performance of lexicase selection is that it enables groups of specialists for solving different parts of the problem to evolve side-by-side, implicitly maintaining the kind of niches that are maintained more explicitly by island models 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 expect 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.

To examine this idea, we must be able to measure the clustering of a population with respect to the training cases. We base the clustering of the population on the individuals' error vectors across the training cases. Since we are primarily interested in whether an individual performs 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[1] to count how many clusters there are 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 a single cluster, recording at each step the distance between the clusters in each merged pair. We can then break the single cluster into smaller clusters by "cutting" the merge between any two clusters whose distance exceeds some threshold. Since we are using binary 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 training cases; for example, if a problem has 200 training cases, we count the number of clusters that differ in binary eliteness on at least 20 training cases. While this distance is somewhat arbitrary, it gives a reasonable and consistent estimate of how many groups of individuals are doing significantly different things in a given generation.

[1] We used the `agnes` (Maechler et al, 2014) implementation of agglomerative clustering in R (R Core Team, 2014), using the `average` linkage when combining clusters.

3 Experiment and Results

We collected data from 100 runs each on 8 different problems described by Helmuth and Spector (2015). All of these are basic programming problems taken from introductory programming texts; several are readily solved, while others remain unsolved using this study's tools. Table 1 lists the problems, a brief description, and the length of the error vectors[2]; other details of the runs can be found in (Helmuth and Spector, 2015). In Table 2 we've also provided the number of successes, i.e., runs in which a program was evolved with total error of 0 across all the training cases. Success rates aren't the focus of this chapter, but these numbers give a sense of the relative difficulty of the problems and illustrate the substantial improvements that lexicase selection provides over both tournament selection and IFS.

Table 1 Short descriptions of the 8 test problems used here, along with the number of errors in each error vector. See (Helmuth and Spector, 2015) for more details on each problem.

Problem name	Description	# errors
Replace Space With Newline	Print the input string, replacing spaces with newlines. Also, return the number of non-whitespace characters.	200
Syllables	Count the number of occurrences of vowels (a, e, i, o, u, y) in the given string and print that number as `X` in `The number of syllables is X`.	200
String Lengths Backwards	Given a vector of strings, print the length of each string in reverse order (starting with last and ending with first).	100
Negative To Zero	Given a vector of integers, return the vector where all negative integers have been replaced by 0.	200
Double Letters	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.	100
Scrabble Score	Given a string of visible ASCII characters, return the Scrabble score for that string.	200
Checksum	Given a string, compute the integer ASCII values of the characters in the string, 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). Then print `Check sum is X`, where `X` is replaced by the correct checksum character.	200
Count Odds	Return the number of odd numbers in a vector of integers.	200

We used the Clojush implementation[3] of the PushGP system (Spector and Robinson, 2002; Spector et al, 2005) for all runs. Each run used a population size of 1,000 individuals, and runs continued for either 300 generations or a until solution was found, whichever came first.

[2] For some of these problems, each test case generates multiple error values because we apply more than one error function.

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

Table 2 Number of successes (out of 100 runs) for each of the 8 test problems used here. These numbers are similar but not identical to those reported in (Helmuth and Spector, 2015) because new runs were performed for this chapter.

Problem name	Lexicase	Tournament	IFS
Replace Space With Newline	57	13	17
Syllables	24	1	2
String Lengths Backwards	75	18	12
Negative To Zero	72	15	9
Double Letters	5	0	0
Scrabble Score	0	0	0
Checksum	0	0	0
Count Odds	4	0	0

Figures 1–16 show error diversity and cluster counts over time for each of the test problems. Below each plot is a smaller sub-plot showing the number of successes over time for each selection; 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. In Figure 1, 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 or median cluster count across whichever of the 100 runs was still running 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.

In general the error diversity numbers for lexicase selection are substantially and 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 there was any substantial overlap between the range of values for lexicase and the other two selection mechanisms (see Figure 5). Typically the lexicase error diversity rises very sharply in the early generations leveling off somewhere between 0.75 and 1.0, meaning that $\frac{3}{4}$ or more of the individuals in the lexicase runs have unique 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 11 and 15), which neither ever solved, where the error diversity values approach or exceed 0.75.

The cluster count results are more mixed. Lexicase selection has clearly higher cluster counts for half of the problems (Replace Space With Newline, Syllables, Scrabble Score, and Count Odds; Figures 2, 4, 12 and 16). It also starts with much higher counts on the Double Letters problem (Figure 10), but those numbers drop again quickly, matching the other two approaches by around generation 100. On the Negative To Zero problem (Figure 8), the lexicase cluster counts remain small (about the same as for both tournament and IFS) throughout the runs. Particularly striking are lexicase cluster counts for String Lengths Backwards (Figure 6) and

Checksum (Figure 14), where the number of clusters with lexicase selection is actually lower earlier in the run.

4 Discussion

As in (Helmuth and Spector, 2015), lexicase selection produced more successes than either tournament selection or IFS on any problem in which a solution was found. 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.

For four of the eight problems, the cluster counts were also much higher for lexicase than for the other two selection mechanisms. For some of these problems (e.g., Count Odds) there are over 100 clusters, and for Syllables the median cluster count is over 400 from generation 100 forward. 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 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. Analysis of the lexicase Checksum runs suggests that the lack of clustering might be a function of structural issues with the test cases; there are 100 test cases, with two error functions per test case: 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 test cases 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. Additional test cases exploring different inputs might allow evolution to first stumble upon and then exploit code that produces actual checksums.

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 and the often high number of clusters, 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 gen-

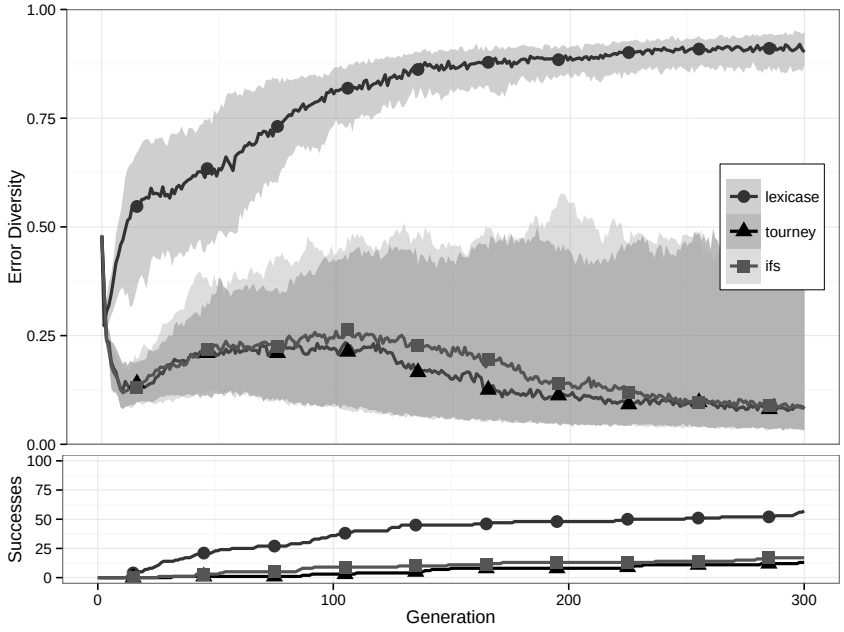

Fig. 1 Replace Space With Newline – error diversity

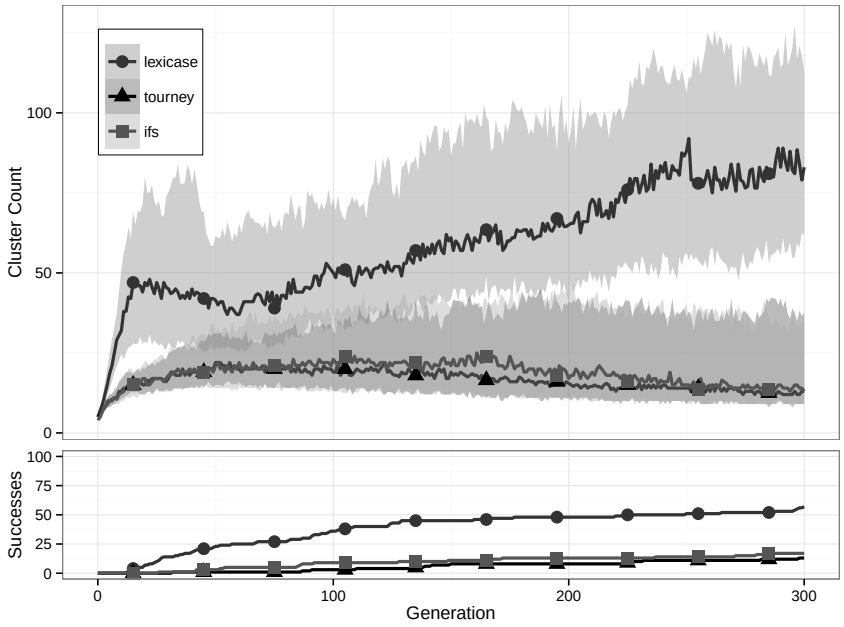

Fig. 2 Replace Space With Newline – cluster counts

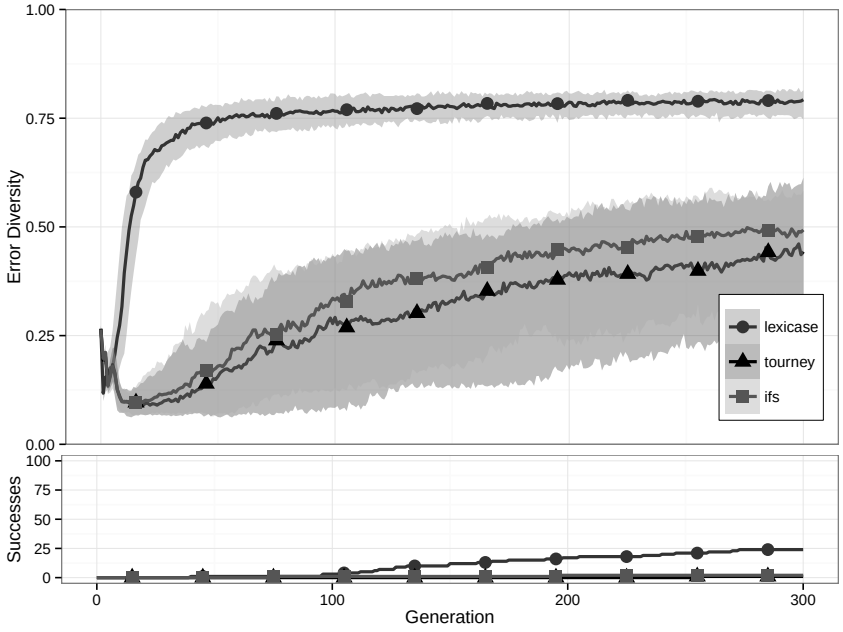

Fig. 3 Syllables – error diversity

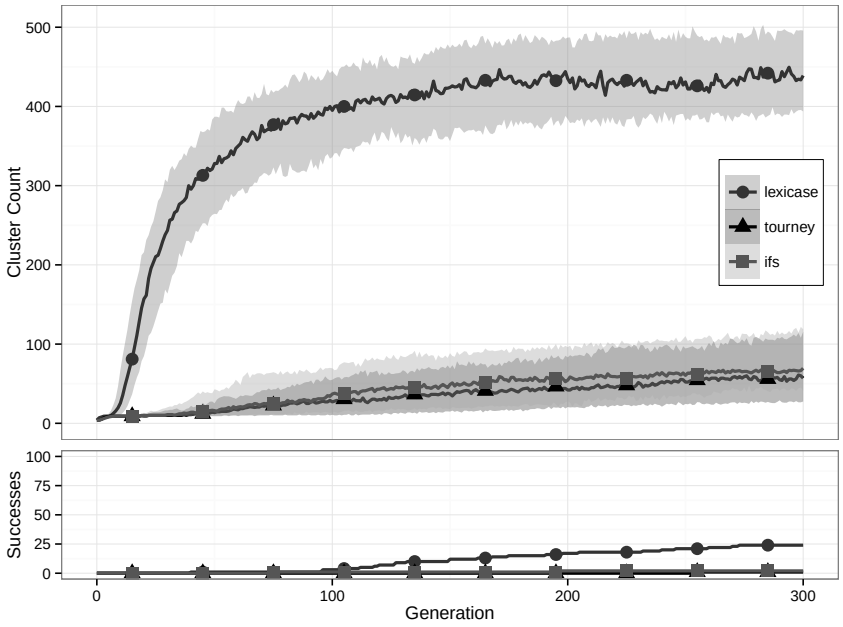

Fig. 4 Syllables – cluster counts

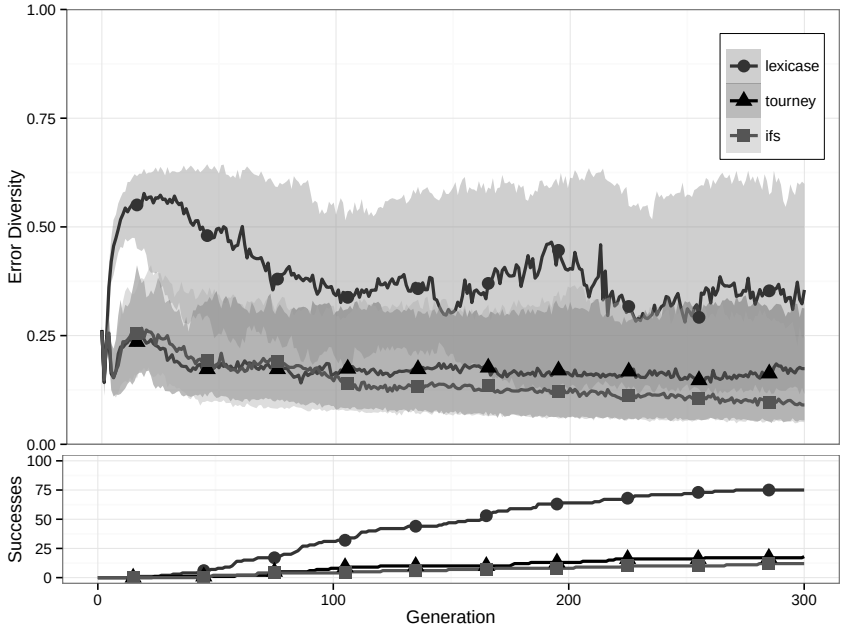

Fig. 5 String Lengths Backwards – error diversity

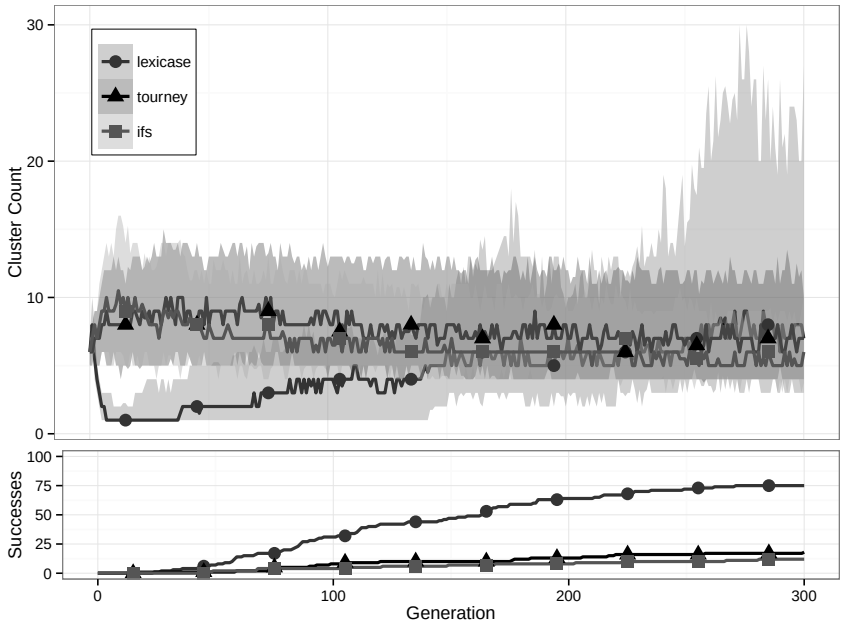

Fig. 6 String Lengths Backwards – cluster counts

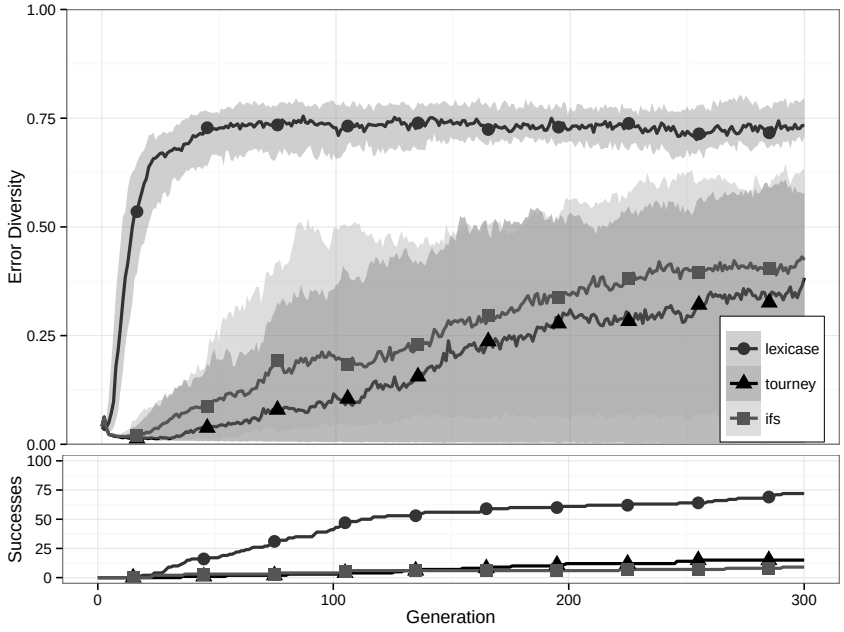

Fig. 7 Negative To Zero – error diversity

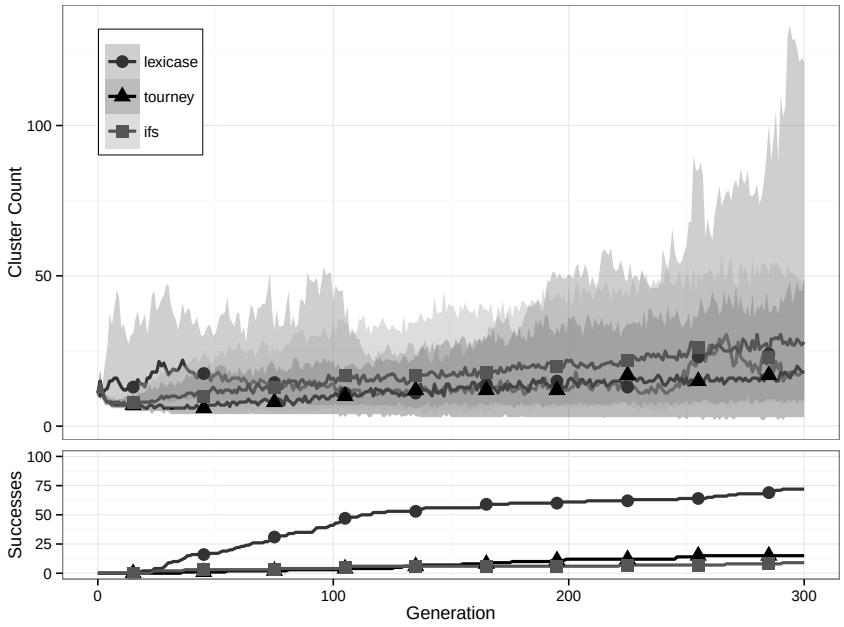

Fig. 8 Negative To Zero – cluster counts

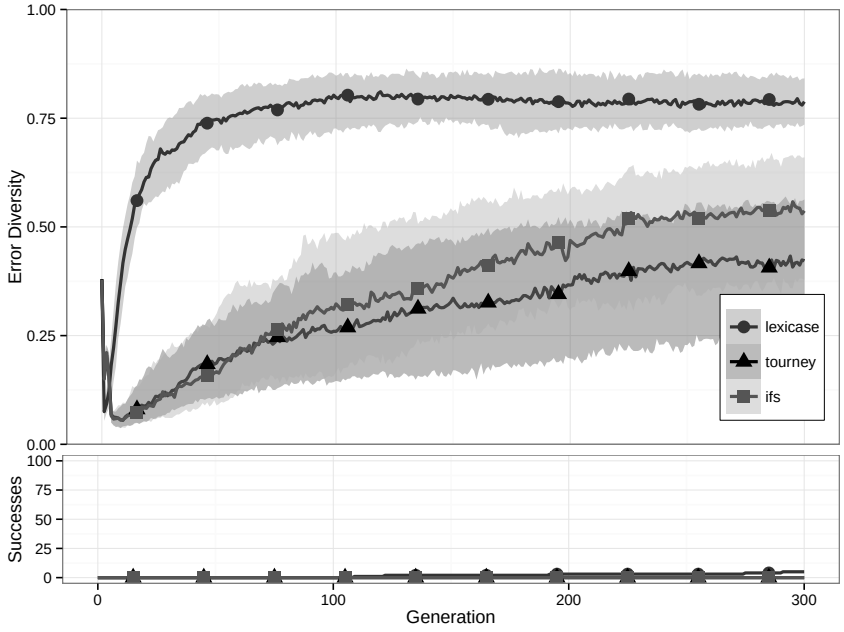

Fig. 9 Double Letters – error diversity

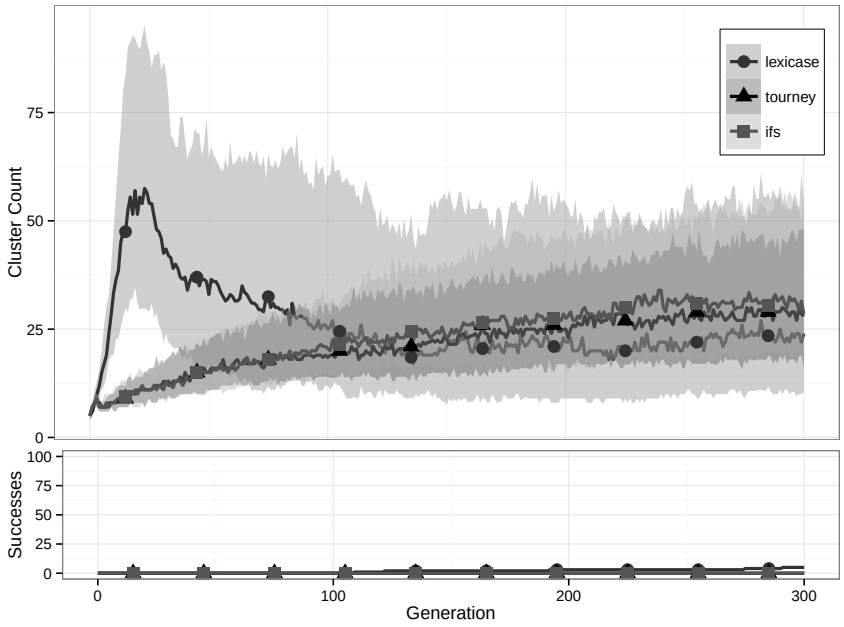

Fig. 10 Double Letters – cluster counts

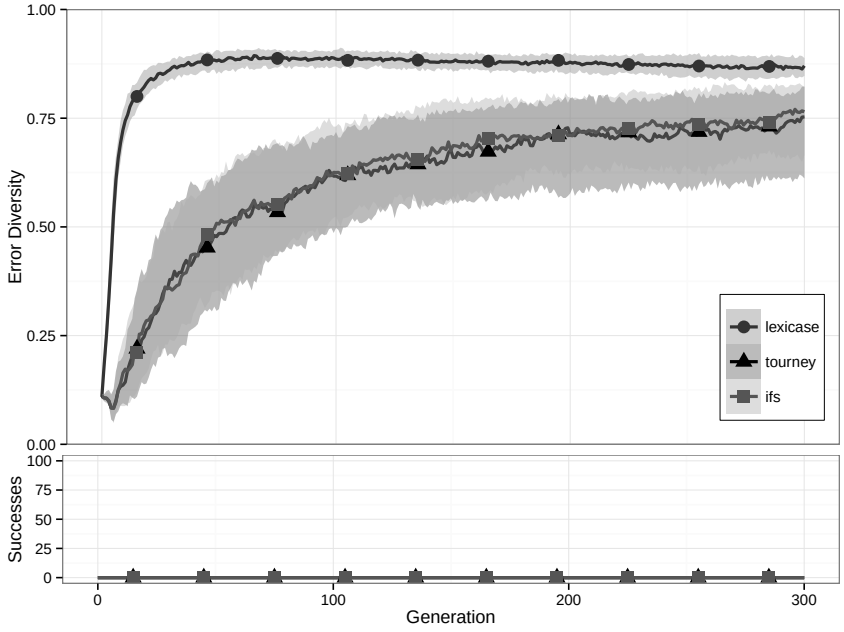

Fig. 11 Scrabble Score – error diversity

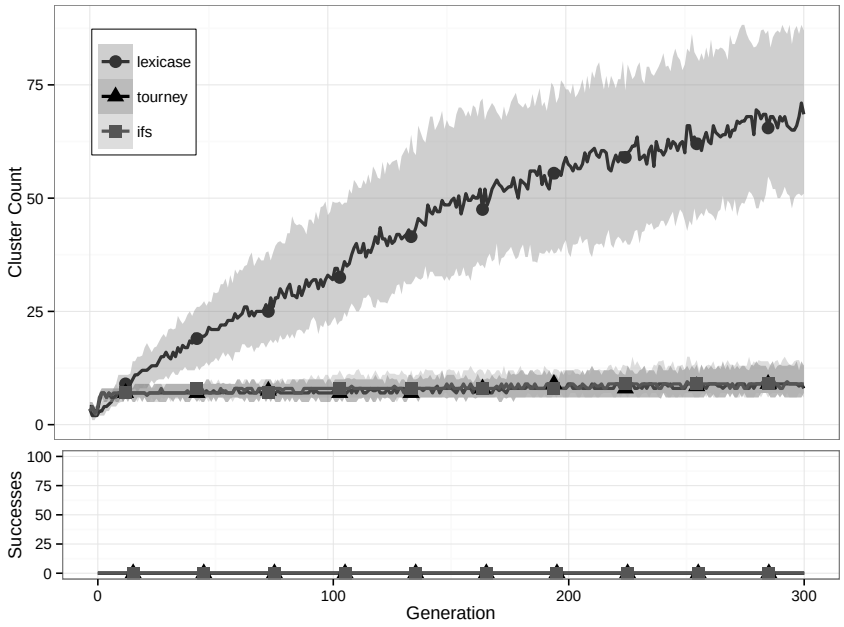

Fig. 12 Scrabble Score – cluster counts

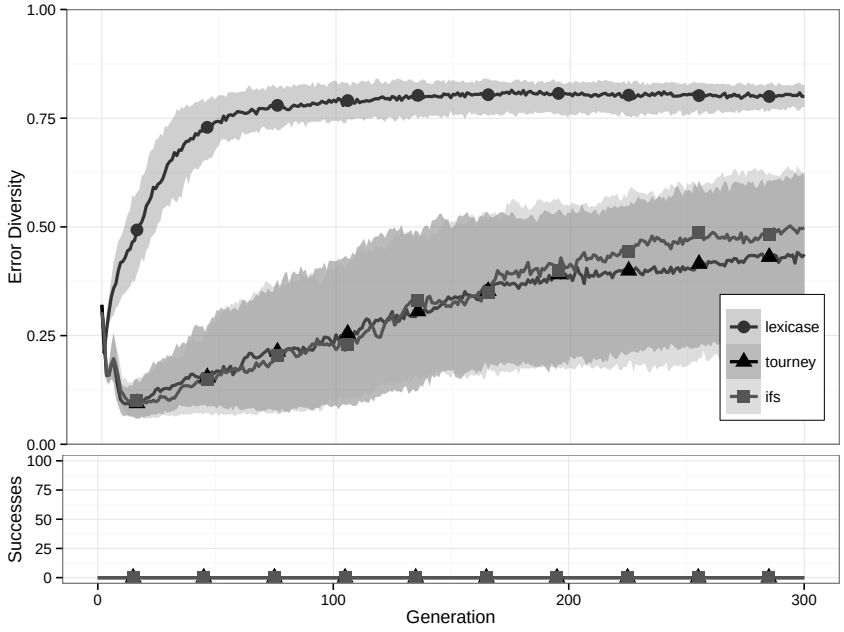

Fig. 13 Checksum – error diversity

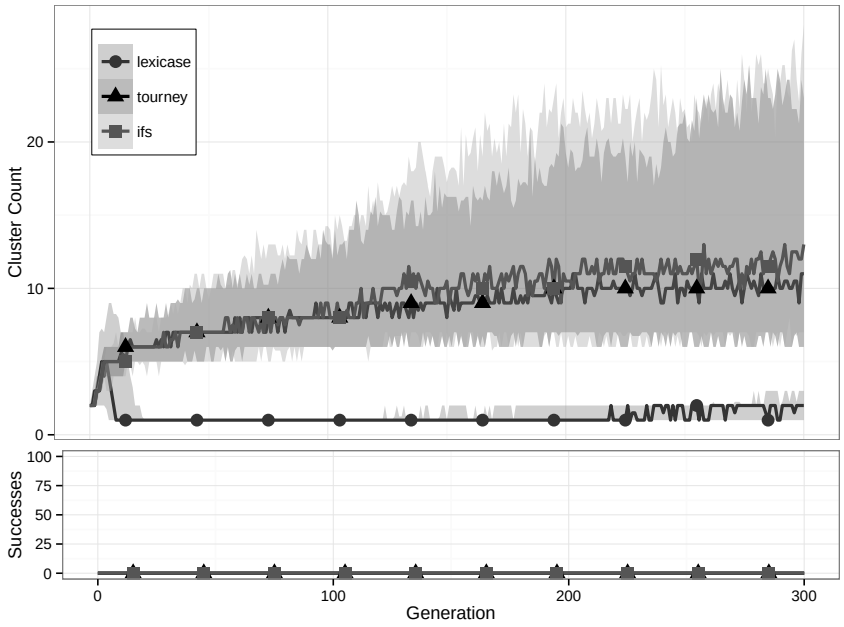

Fig. 14 Checksum – cluster counts

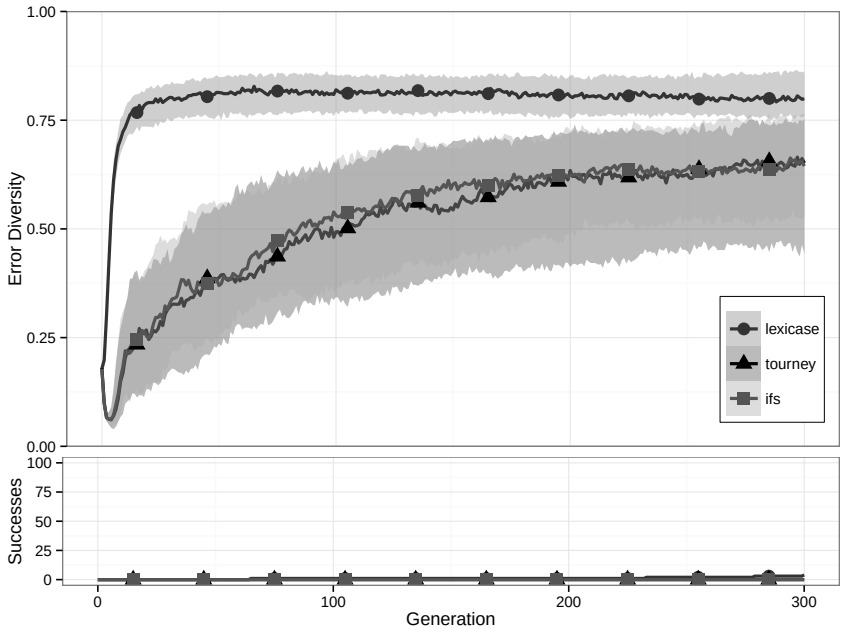

Fig. 15 Count Odds – error diversity

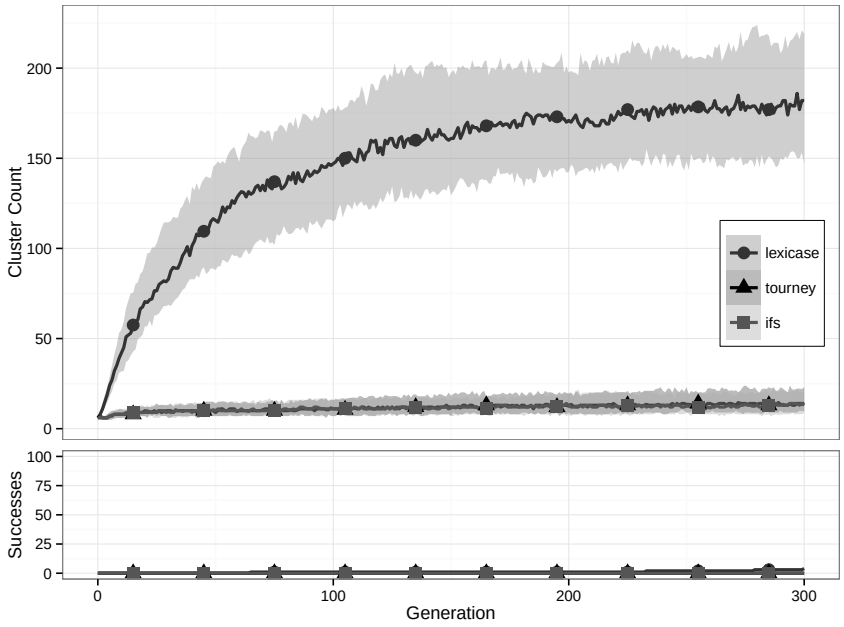

Fig. 16 Count Odds – cluster counts

erations 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 be revealing.

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 the other hand, lexicase selection consistently maintained higher error diversity than other methods, and found more solutions on every problem that was solved. 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.

5 Conclusions

In this chapter we used two different measures of diversity (error diversity and cluster counts) to try to better understand the impact of lexicase selection, and why it seems to consistently outperform tournament selection and implicit fitness sharing (IFS) on a range of software synthesis problems (Helmuth and Spector, 2015). The error diversity was generally *much* higher for lexicase selection than for either tournament selection or IFS, with lexicase selection maintaining a broad range of distinct behaviors. Cluster counts were typically higher with lexicase selection, and the instances in which they weren't may say more about the problem or test case structure than about the selection mechanism. This suggests that error diversity is indeed a valuable metric for studying the impact of system design decisions. The value of cluster counts is less clear, but it seems likely that understanding why the cluster counts were so low on certain problems could be informative.

Given that the lexicase selection runs maintain error diversity all across the 300 generations, it seems plausible that extending the length of the runs would generate additional solutions. It would be illuminating to extend these runs to 500 or 1,000 generations and see whether lexicase selection is able to make "better" use of those additional computational resources.

While the focus of this chapter was to better understand the behavior of lexicase selection, the results also show that tournament selection and IFS behave *very* similarly with respect to the diversity measures used here. This is unfortunate because

IFS was specifically designed to maintain diversity. Both tournament selection and IFS aggregate test case errors into a single value, with IFS just weighting the components differently; this may be partially responsible for the similar rates in diversity.

Acknowledgements Thanks to the members of the Hampshire College Computational Intelligence Lab for discussions that helped to improve the work described in this chapter, to Josiah Erikson for systems support, and to Hampshire College for support for the Hampshire College Institute for Computational Intelligence. This material is based upon work supported by the National Science Foundation under 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.

References

Helmuth T, Spector L (2015) General program synthesis benchmark suite. In: GECCO '15: Proceedings of the 2015 Conference on Genetic and Evolutionary Computation

Helmuth T, Spector L, Matheson J (2014) Solving uncompromising problems with lexicase selection. IEEE Transactions on Evolutionary Computation DOI doi:10.1109/TEVC.2014.2362729

Jackson D (2010) Promoting phenotypic diversity in genetic programming. In: Schaefer R, Cotta C, Kolodziej J, Rudolph G (eds) PPSN 2010 11th International Conference on Parallel Problem Solving From Nature, Springer, Krakow, Poland, Lecture Notes in Computer Science, vol 6239, pp 472–481

Maechler M, Rousseeuw P, Struyf A, Hubert M, Hornik K (2014) cluster: Cluster Analysis Basics and Extensions. R package version 1.15.3

McKay RI (2000) Fitness sharing in genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference, pages 435442, Las Vegas, Morgan Kaufmann, pp 10–12

R Core Team (2014) R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria, URL http://www.R-project.org/

Spector L (2012) Assessment of problem modality by differential performance of lexicase selection in genetic programming: A preliminary report. In: 1st workshop on Understanding Problems (GECCO-UP), ACM, Philadelphia, Pennsylvania, USA, pp 401–408, DOI doi:10.1145/2330784.2330846

Spector L, Robinson A (2002) Genetic programming and autoconstructive evolution with the push programming language. Genetic Programming and Evolvable Machines 3(1):7–40, DOI doi:10.1023/A:1014538503543

Spector L, Klein J, Keijzer M (2005) The push3 execution stack and the evolution of control. In: GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, ACM Press, Washington DC, USA, vol 2, pp 1689–1696, DOI doi:10.1145/1068009.1068292