Evolution Evolves with Autoconstruction

Lee Spector
School of Cognitive Science
Hampshire College
Amherst, Massachusetts, USA
lspector@hampshire.edu

Nicholas Freitag McPhee
Div. of Science & Mathematics
U. Minnesota, Morris
Morris, Minnesota, USA
mcphee@morris.umn.edu

Thomas Helmuth
Dept. of Computer Science
Washington and Lee U.
Lexington, Virginia, USA
helmutht@wlu.edu

Maggie M. Casale
Div. of Science & Mathematics
U. Minnesota, Morris
Morris, Minnesota, USA
casal033@morris.umn.edu

Julian Oks
School of Cognitive Science
Hampshire College
Amherst, Massachusetts, USA
juao15@hampshire.edu

ABSTRACT

In autoconstructive evolutionary algorithms, individuals implement not only candidate solutions to specified computational problems, but also their own methods for variation of offspring. This makes it possible for the variation methods to themselves evolve, which could, in principle, produce a system with an enhanced capacity for adaptation and superior problem solving power. Prior work on autoconsruction has explored a range of system designs and their evolutionary dynamics, but it has not solved hard problems. Here we describe a new approach that can indeed solve at least some hard problems. We present the key components of this approach, including the use of linear genomes for hierarchically structured programs, a diversity-maintaining parent selection algorithm, and the enforcement of diversification constraints on offspring. We describe a software synthesis benchmark problem that our new approach can solve, and we present visualizations of data from single successful runs of autoconstructive vs. non-autoconstructive systems on this problem. While anecdotal, the data suggests that variation methods, and therefore significant aspects of the evolutionary process, evolve over the course of the autoconstructive runs.

1. INTRODUCTION

Designers of evolutionary algorithms have long faced decisions about which aspects of their algorithms they should try to specify explicitly themselves, and which aspects should be allowed to evolve by variation and selection.

In traditional genetic algorithms and genetic programming systems, solutions to problems evolve but almost everything else is specified by the system designer. However, several researchers have developed systems in which aspects of the systems themselves are subject to evolutionary adaptation [2, 34, 3, 9, 25, 10, 33, 4, 52, 31, 53]. In some cases, the primary aim of such "meta-evolutionary," or "self-adaptive" system research is to alleviate the burden on human system designers and applications engineers for specifying system configuration information or parameters. Other researchers, however, are motivated by the hope that evolution could do a better job than humans can do in making system design decisions, and that self-evolving evolutionary algorithms might therefore be able to solve problems that are beyond the reach of more "hard-coded" systems.

Various aspects of an evolutionary algorithm might be allowed to evolve, ranging from mutation rates to mate selection algorithms. Several researchers have noted that the design of variation methods, in particular, is both crucial and non-trivial, and have considered ways in which genetic operators might be produced and improved by evolutionary processes. Genetic programming systems are well suited to work in this area, which has sometimes been called "meta-genetic programming," because genetic programming systems are designed to evolve programs, and because genetic operators are themselves programs [30, 22, 8, 51, 7, 12].

"Autoconstructive evolution" is the name given to a particular approach to the evolution of variation in an evolutionary computation system, in which the evolving individuals implement not only candidate solutions to specified computational problems, but also their own methods for variation of offspring [35, 49, 38, 13, 14]. This makes it possible for the variation methods in the population to themselves be varied, and thereby to evolve.

Autoconstructive evolution systems can be contrasted to other meta-genetic programming approaches, in which the evolving variation methods are distinct from evolving problem solutions, and are sometimes evolved using distinct variation, selection, and assessment methods. In autoconstructive evolution the methods for variation are encoded in the same individuals that are being evolved as solutions to the target problem. The autoconstructive approach is arguably more biological than other approaches to meta-evolution, because although many aspects of biological variation mechanisms are shared across the biosphere and have been conserved for billions of years, others vary from organism to organism and change over the course of evolution.

Whether the autoconstructive approach can produce more powerful evolutionary computation systems than those produced by other approaches is currently an open question.

ECADA 2016 Denver, CO USA

ISBN 978-1-4503-4323-7/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2908961.2931727

Prior autoconstructive evolution systems could only solve relatively simple problems, but they demonstrated and allowed for exploration of the evolutionary dynamics that result when variation methods can themselves vary as components of individuals. The fact that these systems could be outperformed by standard genetic programming systems was not surprising; whereas standard systems employ human-designed variation mechanisms, autoconstructive evolution systems must discover effective variation mechanisms and maintain or enhance them over evolutionary time. The prior work allowed for the study of fundamental principles of autoconstructive evolution, but it did not produce high-functioning problem-solving systems; the methods did not reach the critical threshold for self-improvement that would permit higher levels of evolutionary innovation to occur.

In this paper we describe a new approach to autoconstruction that can indeed solve at least some hard problems. In the next section we describe Push, a representation for evolving programs that was originally developed to support autoconstruction and has been used in several previous studies. We then describe the key components of our new approach, which is implemented in a system called AutoDoG (for "Autoconstructive Diversification of Genomes"), emphasizing the fact that we do not yet know which components may be responsible for AutoDoG's successes. Next, we describe a software synthesis benchmark problem, "Replace Space with Newline," that AutoDoG can solve, and we compare data from single successful runs of autoconstructive vs. non-autoconstructive systems on this problem. Finally, we argue that the data suggests that variation methods, and therefore significant aspects of the evolutionary process, evolve over the course of the autoconstructive runs.

2. THE PUSH LANGUAGE

Push is a programming language that was developed to serve as the language in which evolving programs are expressed [35, 49, 45, 15]. It has a minimal syntax while nonetheless providing unusual semantic power through the use of a multi-type stack-based execution architecture and the inclusion of code-manipulation instructions.

While the Push language cannot be described completely here, it can be characterized briefly as a stack based language with a separate stack for each data type. Among the supported data types is `code`, which can be manipulated and executed. Programs are executed by putting them on the `exec` stack, from which the interpreter continuously takes and processes items. When the interpreter sees literals it pushes them onto stacks as indicated by their types. When it sees instructions, it executes them, taking any needed arguments from stacks of the appropriate types and pushing any results onto stacks of the appropriate types. If the arguments needed by an instruction are not available on the stacks, then the instruction acts as a "NOOP" and does nothing. The `exec` stack can be manipulated in the same ways as other data stacks, and this simplifies the implementation and evolution of a wide range of novel control structures.

Push has several features that facilitate the evolution of complex programs. Its multi-stack architecture allows programs to manipulate multiple data types without syntax restrictions and also without the possibility of type errors occurring [49]. Its re-entrant interpreter loop simplifies the evolution of several forms of iteration, recursion, and modularity [45, 41]. Its minimal syntax allows for simple hill-climbing simplification algorithms to effectively minimize the size of evolved programs [43]. It has been been used for a range of applications and as the foundation for several studies of new genetic programming techniques [50, 29, 47, 46, 48, 37, 6, 42, 44, 23]. Push implementations have been written in C++, Java, JavaScript, Python, Common Lisp, Scheme, Erlang, Clojure, and R. Many of these are available for free download from the Push project web page, `http://pushlanguage.org`.

Most prior studies and applications of Push have used human-specified variation (crossover and mutation) methods, as are used in most other genetic programming systems. These have ranged from tree-swapping operators similar to Koza's [24], to operators that control program size [6] or facilitate uniform variation [42]. No autoconstructive evolution was involved in these studies, or in the uses of Push to achieve human-competitive results [21].

Nevertheless, Push is well-suited for autoconstructive evolution research, and indeed support for autoconstruction was one of the driving forces behind the language's original design [35]. In autoconstructive evolution, evolving programs must interact with their problem environments while also producing offspring. In Push, code for different tasks can be freely intermingled, producing as many outputs of as many types as needed, which can be left on data stacks at program termination. Outputs can include both candidate solutions to target problems and genomes of potential children. This makes it as simple to express autoconstructive programs in Push as it is to express other programs.

3. AutoDoG

Although autoconstructive evolution systems allow many aspects of the evolutionary algorithm to evolve, one must nonetheless make design decisions about the autoconstructive evolution system itself. What instructions will be available for varying offspring? What kind of access will individuals have to other individuals, for implementing recombination? When will the production and variation of offspring take place, relative to fitness assessment? How will the emergence of evolution-ending exact cloning strategies be prevented? A range of answers to these and related questions have been explored in prior work.

For example, the Pushpop system produced children during fitness assessment, using a Turing-complete collection of code-manipulation instructions to generate a potential child for each test case; subsequent tournaments selected individuals from whom potential children would be promoted to the following generation (assuming that they were not clones of their parents, which was prohibited) [49]. Individuals in Pushpop could conduct tournaments among the rest of the population, from which they could take mates, and they could could use code from their mates in the construction of children and even in the computation of solutions to the target problem. Pushpop could solve only relatively simple problems (for example, symbolic regression of $y = 5x^2 + x - 2$), and because of the ways in which individuals could depend on others in the population, it was often difficult to understand how evolved solutions actually worked. Nonetheless, the system allowed for the study of evolutionary dynamics under autoconstruction [36].

Subsequent systems explored a variety of alternatives, involving, for example, changes to the reproductive instruction set, changes to the ways in which mates (if any) are

selected, and changes to constraints on the birth (for example, extending the "no cloning" rule to prohibit other forms of pathological replication) and constraints on selection (for example, favoring parents with non-stagnant lineages) [38, 14]. They also explored applications to different classes of problems, including the evolution of agent behaviors in a 3D virtual world [48] and the evolution of programs with specified structural properties [11]. We cannot describe all of the previous studies here in detail, and will instead focus only on the design decisions behind our new system, AutoDoG (for "Autoconstructive Diversification of Genomes").

Before we describe the features of AutoDoG, however, we want to be clear that *we do not know which of these, or which combinations of these, may be responsible for the fact that AutoDoG appears to be capable of solving more difficult problems than previous autoconstructive evolution systems.* We have results, including those described below, that seem to indicate that *something* about our new approach constitutes an important step in the right direction. More study will be required, however, to determine what role each of the new features plays, if any, in the improved performance of AutoDoG relative to previous systems.

3.1 Diversity-maintaining parent selection

Many problems involve multiple test cases that are used to determine the performance of programs during evolution. There are several ways in which the results of multiple test cases can be taken into consideration during parent selection. The most common approach is to sum the squares or the absolute values of the errors across all cases, and then to select parents by tournaments favoring programs with lower total error. This can work well in simple, homogenous problems, but it can work poorly when cases are heterogeneous, unbalanced, or interdependent. A technique called "implicit fitness sharing," which was first described in [32], and later adapted for genetic programming in [26], uses a weighted sum, where weights are inversely proportional to the number of individuals that solve a particular case; this gives a program a greater reward for solving a case when fewer individuals in the population solve that case.

A newer approach, called "lexicase selection," performs significantly better than tournament selection, either with or without implicit fitness sharing, on many kinds of problems [39, 20]. In lexicase selection, each parent selection event starts with a pool that contains the entire population. The pool is then filtered on the basis of performance on individual fitness cases, considered in random order one at a time. For each case, it retains only those individuals that are best on that case. This process filters the population using a kind of "lexicographic ordering" of cases. Over many selection events, each of which will filter the population using a sequence of cases in a different random order, this will select parents that are best on each individual case and, in the limit, on all pairs and larger subsets of the cases.

Lexicase selection was developed for the sake of its anticipated effects on program search, but it could also be argued that it is more analogous to selection in nature than are prior algorithms. Individual biological organisms are subjected to sequences of challenges over their lifetimes, reproducing if they successfully handle the challenges that they happen to face before the opportunity arises; they do not become parents on the basis of their average ability to handle all possible challenges that might arise.

Lexicase selection appears to help solve "modal" problems, in which qualitatively different modes of response are required for inputs from different regions of the domain. It also appears to help solve problems that are "uncompromising" in the sense that a solution must perform as well on each test case as it is possible to perform on that case; that is, for which it is not acceptable for a solution to perform suboptimally on any one case in exchange for good performance on others [20]. Lexicase selection performs significantly better than the standard approach and than implicit fitness sharing, allowing the solution of more problems, in fewer generations, on a benchmark suite of problems taken from introductory programming textbooks [27, 19]. Analysis has shown that lexicase selection produces and maintains population diversity significantly better than previous techniques, and that it appears to combine exploration and exploitation in novel ways [15, 16]. This may be particularly important for autoconstructive evolution systems, in which one cannot rely on hand-coded mutation and crossover operations to maintain diversity and support exploration.

3.2 Linear genomes

Several previous autoconstructive evolution systems have produced children in the form of Push programs, which can have a nested hierarchical structure (expressed with parentheses), using code-manipulation instructions to access and alter the programs of the parent(s) in order to produce a child. Recently, however, a linear genome format for Push programs has been developed, under the name of "Plush," where the "l" is for "linear" [17]. Plush was developed in part to facilitate the development of genetic operators with "uniformity" properties [42], and in part to increase the likelihood that instructions that are intended to operate on structured code blocks, for example conditionals like `exec_if` and iterators such as `exec_do*range`, will in fact receive such code blocks as arguments.

To support translation of linear Plush genomes into structured Push programs, each instruction that is intended to operate on code blocks is annotated with the number of code blocks that it should open. During translation, this information is used to open parenthesized sub-programs, while "epigenetic close markers" on the linear genome indicate where parenthesized sub-programs should end. Additionally, epigenetic silencing markers can be turned on to indicate that marked instructions should not be considered when translating a genome into a Push program.

In AutoDoG, autoconstructive reproduction operates on the Plush genomes of two parents to produce the Plush genome of a child. Genome manipulation instructions are included in the instruction set, and the (single) parent that is run may use these to produce a child from the genomes of the two parents. Table 1 shows AutoDoG's current set of genome instructions.

3.3 Diversification constraint

From the first work on autoconstructive evolution it was recognized that programs in a population could not be allowed to make exact clones of themselves. If they were allowed to do so, and if a cloning program were to arise with reasonably good performance on the target problem, then the descendants of the cloning program, all of which would be identical, would rapidly fill the population. After this happens no further evolution is possible, since these

Table 1: Genome instructions in AutoDoG

Instruction	Description
close_dec	Decrement close marker on a gene
close_inc	Increment close marker on a gene
dup	Duplicate top genome
empty	Boolean, is genome stack empty?
eq	Boolean, are top genomes equal?
flush	Empty genome stack
gene_copy	Copy gene from genome to genome
gene_copy_range	Copy genome segment
gene_delete	Remove gene
gene_dup	Duplicate gene
gene_randomize	Replace with random
new	Push empty genome
parent1	Push first parent's genome
parent2	Push second parent's genome
pop	Remove top genome
rot	Rotate top 3 genomes on stack
rotate	Rotate sequence of top genome
shove	Insert top genome deep in stack
silence	Add epigenetic silencing marker
stackdepth	Push integer depth of genome stack
swap	Exchange top two genomes
toggle_silent	Reverse silencing of a gene
unsilence	Remove epigenetic silencing marker
yank	Pull genome from deep in stack
yankdup	Copy genome from deep in stack

programs can only generate offspring that are identical to themselves. Although one could conceivably apply human-devised mutation operators to cloned offspring, this would eliminate the evolutionary pressure for generating variation with the code in the programs themselves, which is what will allow the variation mechanisms to evolve. Instead, therefore, autoconstructive evolution systems have simply prevented clones from entering the population; results of reproduction events that would produce clones have either been discarded or replaced with new random individuals.

AutoDoG broadens the "no cloning" rule to require that offspring are only allowed to enter the population if they pass a more stringent diversification test. Specifically, to test if an individual will be permitted to enter the population, its program is run to produce several temporary children, with itself as its mate. These children are used only for the diversification test, and are discarded after the test is complete. The program passes the diversification test only if the temporary children differ both from the program itself and from each other. Furthermore, some of of the temporary children must differ from the program itself by different amounts (which is calculated as the Levenshtein distance between linearized representations of the Push programs). If an individual that has been produced by autoconstruction fails the diversification test then a new, random individual is generated. If that new, random individual passes the diversification test then it is added to the population in place of the original individual. If it too fails the diversification test, then an individual with an empty genome is added to the population instead. This is intended to ensure that after the first, random generation, only individuals that pass the diversification test will be allowed in the population and compete for selection as parents in the next generation.

AutoDoG's current diversification constraint is only one of many possible such constraints, and perhaps not the most effective. Ideally, individuals that satisfy the constraint will produce descendants that vary *in the ways in which they vary their offspring,* thereby allowing variation methods to evolve. This is not necessarily true for the current constraint, and for this reason we are exploring alternatives. In the interim, we have used the constraint described here because it is simple and, it appears, reasonably effective.

3.4 AutoDoG architecture

AutoDoG is implemented within Clojush [40], which is an implementation of the Push programming language and the PushGP genetic programming system in the Clojure programming language. In this implementation, AutoDoG is simply PushGP, run using an autoconstruction genetic operator rather than human designed mutation and crossover operators. The overall control flow of AutoDoG is the same as that of PushGP, which is a reasonably standard generational genetic programming system, with a main loop that iteratively tests the error of all individuals and then builds the next generation by selecting parents and passing them to genetic operators. In AutoDoG, however, only the autoconstruction genetic operator is used.

The autoconstruction genetic operator takes two parents (which will have been selected using lexicase selection), and returns a child. It produces the child by running one of the parents in a context in which the genomes of both parents are pre-loaded on the `genome` stack and are also available via the `genome_parent1` and `genome_parent2` instructions. Prior to running the parent, any instructions that would access input are replaced with `code_noop` instructions (so that they will do nothing), and substitutions are made so that `autoconstructive_integer_rand` and `autoconstructive_boolean_rand` will act as calls to the random value generator, even though they are deterministic (pushing `0` or `false`, respectively) when run during error testing. This allows programs to use random values when generating offspring, even in problem domains for which it would be inappropriate to use a random number generator when testing a program for errors. The child produced by running the parent is then subjected to the diversification test described in Section 3.3, and either the child (if it passes the test), a random replacement (if that passes the test), or an individual with an empty genome is returned as the result of the autoconstruction genetic operator and added to the population for the next generation.

4. RESULTS

4.1 Solving a hard problem

Among the problems to which AutoDoG has been applied is "Replace Space with Newline," a software synthesis problem: given a string, print the string, replacing spaces with newlines, and also return the integer count of the non-whitespace characters [19]. This involves multiple data types and multiple outputs, and requires conditionals and iteration or recursion. Full details of the problem (instruction set, test cases, etc.) are documented in [19, 18, 15], as is the fact that this is a difficult software synthesis problem, for which the best prior work has produced success rates of only about 50%, and for which we have never observed solutions that were generated randomly, without evolution.

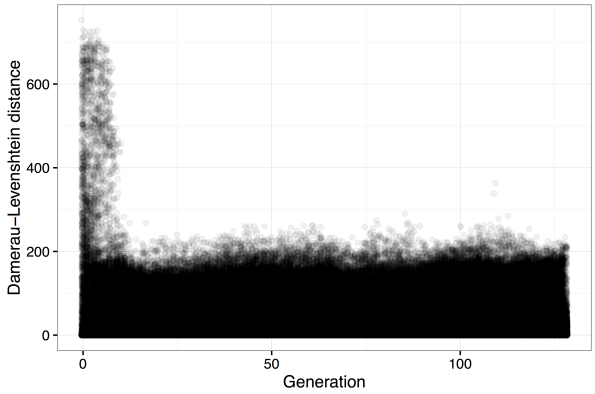

Figure 1: DL-distances between parent and child during a single non-autoconstructive run of GP on the Replace Space With Newline problem

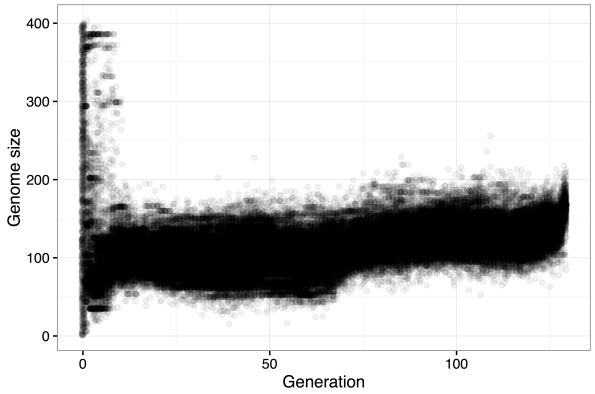

Figure 2: Genome sizes during a single non-autoconstructive run of GP on the Replace Space With Newline problem

AutoDoG does not succeed as reliably on this problem as has PushGP in some other configurations, but it does solve the problem approximately $5 - 10\%$ of the time, producing general solutions. Because this is a harder problem than has been solved by previous autoconstructive evolution systems, we take this as an indication that something in AutoDoG is on the right track, and we sought to understand what is happening in AutoDoG populations when it does succeed.

4.2 Evolution evolving

One way to better understand the behavior of reproduction mechanisms is to look at the the ways in which they convert parent genomes into child genomes. Here we use the Damerau-Levenshtein (edit) distances (DL-distances), applying them to sequences obtained by extracting the three components of each gene (the instruction and the close and silencing epigenetic markers discussed in Section 3.2).[1] The stability of the reproduction mechanisms in traditional, non-autoconstructive runs is apparent in Figure 1. Here we see that, after the variation in the initial random population settles out, the distances between parents and children remain fairly consistent across the duration of the run. This is also reflected by the changes in genome size over time (Figure 2), where there is a slight upward creep, but the sizes are again bounded in a fairly narrow range.

Looking at the same data for a successful autoconstruction run, we see dramatically different behaviors. Figure 3 shows the DL-distances between parents and children in this run. Distances were consistently under about 200 throughout the non-autoconstructive run, while here they are scattered in clear clusters across a much broader range, extending up to nearly $2,500$ around generation 100. There is similar clustering in the plot of genome sizes over time (Figure 4), with many of those clusters having clear analogues in Figure 3.

Presumably these different groupings represent different approaches to replication that are being explored by autoconstruction. Most of the DL-distances in the autoconstruction run are small (a quarter below 5, half below 20), suggesting that most of the autoconstruction mechanisms in this run create offspring by making small changes to the par-

[1]This is why the maximum DL-distances in Figure 3 are about three times the largest genome sizes in Figure 4.

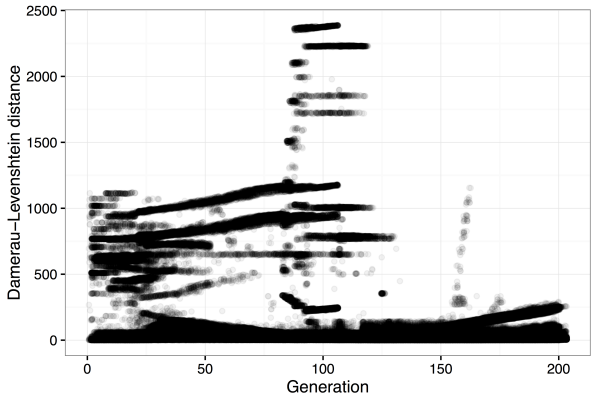

Figure 3: DL-distances between parent and child during a single autoconstructive run of GP on the Replace Space With Newline problem

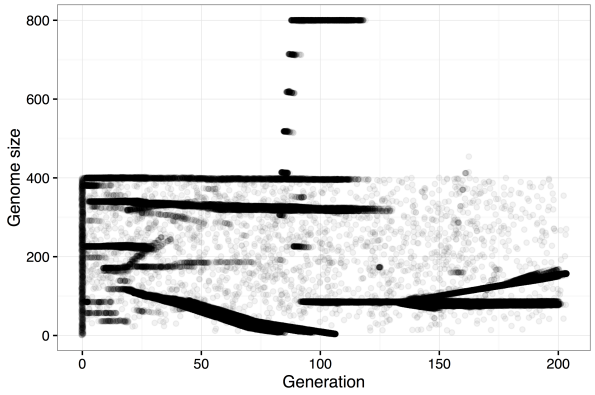

Figure 4: Genome sizes during a single autoconstructive run of GP on the Replace Space With Newline problem

ent genomes. There are definite exceptions, however, such as the steep "stair step" climb in both DL-distances and genome lengths starting around generation 80, bringing the largest genome lengths up to the maximum allowed (800) for about 50 generations. There are also clusters with definite directional trends in genome length, suggesting reproduction methods that consistently either lengthen or shorten child genomes relative to parent genomes. Some of these appear to be dead ends, such as the strong downward trend in genome sizes from about 100 around generation 25 down to nearly 0 by about generation 75; presumably at some point the programs got short enough that they either performed too poorly on the test problem to be selected as parents or were unable to satisfy the diversification constraint.

Of particular interest is the "forked" structure in Figure 4 that starts at generation 91 and continues to the end of the run. The genome lengths start just under 100 and remain fairly constant until around generation 140 when there is a split. After the split, one section of the population continues to have genome sizes around 100, while the other shows a slow but steady increase in the genome sizes, coming close to 200 by the end of the run. The beginning of this fork is "seeded" by a new randomly generated individual in generation 91, introduced because an individual failed to pass the diversification test as described in Section 3.3. That individual becomes the progentitor of this entire structure, whose success (both at reproduction and at the target problem) appears to eventually dominate the run, driving out the other clusters. The split, however, shows that it, too, continues to evolve its reproduction methods, perhaps influenced by recombination with individuals from the other clusters. The ultimately successful individual is in fact located at the end of the upper branch of this fork, and most of its immediate ancestors are also located along that branch, suggesting that the small increases in size were sometimes accompanied by at least small improvements in performance on the problem.

5. CONCLUSIONS AND FUTURE WORK

In this paper we presented recent research on autoconstructive evolution, in which evolving individuals implement their own methods for variation of offspring along with solutions to target computational problems. Our new approach, which is implemented in a system called AutoDoG, uses linear genomes for hierarchically structured programs, a diversity-maintaining parent selection algorithm (lexicase selection), and the enforcement of diversification constraints on individuals in the population.

We observed that AutoDoG is capable of finding general solutions to the "Replace Space with Newline" software synthesis problem, which sets a new standard for the problem solving power of autoconstructive evolution. While autoconstructive evolution has not yet been demonstrated to have capabilities beyond those of genetic programming with hand-designed genetic operators, these new results give us hope that such results may not be too distant.

Toward that end, we have studied the details of the evolutionary dynamics of AutoDoG in comparison to other (non-autoconstructive) runs of PushGP, and we have presented visualizations showing that the processes of variation, and therefore the processes of evolution, do indeed vary and evolve over the course of AutoDoG runs.

We do not yet know which features of AutoDoG are primarily responsible for its problem-solving power, and investigation of this question is clearly the next order of business. Through study of this question, further study of the evolutionary dynamics of successful and unsuccessful runs of AutoDoG, and refinement of the methods that it uses for autoconstruction (for example, refinement of the diversification constraint discussed in Section 3.3), we hope eventually to produce autoconstructive evolution systems with problem-solving power that rivals or exceeds that of genetic programming systems with hand-designed genetic operators.

Another promising area for future work is the use of methods presented here in other kinds of evolutionary computation systems. For example, in the Avida artifical life platform [1], it is possible for digital organisms to encode procedures for varying their offspring, although in practice they generally make exact copies that are subjected to an externally imposed mutation process to produce variation. In Avida, variation methods encoded in individuals are rarely observed, they have never been observed to contribute significantly to evolution, and have sometimes been explicitly prohibited [5, 28]. It would be interesting to see if the use of techniques presented here, particularly the enforcement of diversification constraints, might allow for autoconstructive evolution, and therefore evolving variation methods, in Avida and in other evolutionary computation systems.

6. ACKNOWLEDGMENTS

We thank the members of the Hampshire College Computational Intelligence Lab for helpful discussions, J. Erikson for systems support, and Hampshire College for support for the Hampshire College Institute for Computational Intelligence. We also thank the anonymous reviewers who provided both corrections and insightful comments. This material is based upon work supported by the National Science Foundation under Grants No. 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.

7. REFERENCES

[1] C. Adami, C. T. Brown, and W. K. Kellogg. Evolutionary Learning in the 2D Artificial Life System *Avida*. In *Artificial Life IV*, pages 377–381, 1994.

[2] P. J. Angeline. Adaptive and self-adaptive evolutionary computations. In M. Palaniswami and Y. Attikiouzel, editors, *Computational Intelligence: A Dynamic Systems Perspective*, pages 152–163. IEEE Press, 1995.

[3] P. J. Angeline. Two self-adaptive crossover operators for genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 5, pages 89–110. MIT Press, Cambridge, MA, USA, 1996.

[4] H.-G. Beyer and S. Meyer-Nieberg. Self-adaptation of evolution strategies under noisy fitness evaluations. *Genetic Programming and Evolvable Machines*, 7(4):295–328, Dec. 2006.

[5] A. W. Covert, R. E. Lenski, C. O. Wilke, and C. Ofria. Experiments on the role of deleterious mutations as stepping stones in adaptive evolution. *Proceedings of the National Academy of Sciences*, 110(34):E3171–E3178, 2013.

[6] R. Crawford-Marks and L. Spector. Size control via size fair genetic operators in the PushGP genetic programming system. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 733–739. Morgan Kaufmann Publishers, 2002.

[7] L. Diosan and M. Oltean. Evolutionary design of evolutionary algorithms. *Genetic Programming and Evolvable Machines*, 10(3):263–306, Sept. 2009.

[8] B. Edmonds. Meta-genetic programming: Co-evolving the operators of variation. *Elektrik*, 9(1):13–29, May 2001. Turkish Journal Electrical Engineering and Computer Sciences.

[9] A. E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transations on Evolutionary Computation*, 3(2):124–141, July 1999.

[10] R. Fry, S. L. Smith, and A. M. Tyrrell. A self-adaptive mate selection model for genetic programming. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 3, pages 2707–2714. IEEE Press, 2005.

[11] D. E. Goldberg and U.-M. O'Reilly. Where does the good stuff go, and why? how contextual semantics influence program structure in simple genetic programming. In *Proceedings of the First European Workshop on Genetic Programming*, pages 16–36. Springer-Verlag, 1998.

[12] B. W. Goldman and D. R. Tauritz. Self-configuring crossover. In *GECCO 2011 1st workshop on evolutionary computation for designing generic algorithms*, pages 575–582. ACM, 2011.

[13] K. Harrington, E. Tosch, L. Spector, and J. Pollack. Compositional autoconstructive dynamics. In *Unifying Themes in Complex Systems Volume VIII: Proceedings of the Eighth International Conference on Complex Systems*, New England Complex Systems Institute Series on Complexity, pages 856–870. NECSI Knowledge Press, 2011.

[14] K. I. Harrington, L. Spector, J. B. Pollack, and U.-M. O'Reilly. Autoconstructive evolution for structural problems. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, GECCO Companion '12, pages 75–82. ACM, 2012.

[15] T. Helmuth. *General Program Synthesis from Examples Using Genetic Programming with Parent Selection Based on Random Lexicographic Orderings of Test Cases*. PhD thesis, College of Information and Computer Sciences, University of Massachusetts Amherst, USA, Sept. 2015.

[16] T. Helmuth, N. F. McPhee, and L. Spector. Lexicase selection for program synthesis: A diversity analysis. In R. Riolo, W. P. Worzel, and K. Groscurth, editors, *Genetic Programming Theory and Practice XIII*. Springer, 2015. in press.

[17] T. Helmuth, N. F. McPhee, and L. Spector. Plush: Linear genomes for structured push programs. In *Genetic Programming Theory and Practice XIV*, Genetic and Evolutionary Computation. Springer, 2016.

[18] T. Helmuth and L. Spector. Detailed problem descriptions for general program synthesis benchmark suite. Technical Report UM-CS-2015-006, School of Computer Science, University of Massachusetts, Amherst, 2015.

[19] T. Helmuth and L. Spector. General program synthesis benchmark suite. In *GECCO '15: Proceedings of the 2015 Conference on Genetic and Evolutionary Computation*, July 2015.

[20] T. Helmuth, L. Spector, and J. Matheson. Solving uncompromising problems with lexicase selection. *Evolutionary Computation, IEEE Transactions on*, 19(5):630–643, Oct 2015.

[21] K. Kannappan, L. Spector, M. Sipper, T. Helmuth, W. L. Cava, J. Wisdom, and O. Bernstein. Analyzing a decade of human-competitive ("HUMIE") winners: What can we learn? In R. Riolo, W. P. Worzel, and M. Kotanchek, editors, *Genetic Programming Theory and Practice XII*, Genetic and Evolutionary Computation, pages 149–166, Ann Arbor, USA, 8-10 May 2014. Springer.

[22] W. Kantschik, P. Dittrich, M. Brameier, and W. Banzhaf. Meta-evolution in graph GP. In *Genetic Programming, Proceedings of EuroGP'99*, pages 15–28. Springer-Verlag, 1999.

[23] J. Klein and L. Spector. Genetic programming with historically assessed hardness. In R. L. Riolo, T. Soule, and B. Worzel, editors, *Genetic Programming Theory and Practice VI*, Genetic and Evolutionary Computation, chapter 5, pages 61–75. Springer, 2008.

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

[25] R. M. MacCallum. Introducing a perl genetic programming system: and can meta-evolution solve the bloat problem? In *Genetic Programming, Proceedings of EuroGP'2003*, pages 364–373. Springer-Verlag, 2003.

[26] R. I. McKay. Fitness sharing in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 435–442. Morgan Kaufmann, 2000.

[27] R. Moll. iJava—an online interactive textbook for elementary java instruction: Demonstration. *J. Comput. Sci. Coll.*, 26(6):55–57, June 2011.

[28] C. Ofria. private communication, 2016.

[29] A. Robinson and L. Spector. Using genetic programming with multiple data types and automatic modularization to evolve decentralized and coordinated navigation in multi-agent systems. In *Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO-2002)*, pages 391–396. AAAI, 2002.

[30] J. Schmidhuber. Evolutionary principles in self-referential learning. on learning now to learn: The meta-meta-meta...-hook. Diploma thesis, Technische Universitat Munchen, Germany, 14 May 1987.

[31] S. Silva and S. Dignum. Extending operator equalisation: Fitness based self adaptive length distribution for bloat free GP. In *Proceedings of the 12th European Conference on Genetic Programming, EuroGP 2009*, pages 159–170. Springer, 2009.

[32] R. Smith, S. Forrest, and A. S. Perelson. Population

diversity in an immune system model: Implications for genetic search. In *Foundations of Genetic Algorithms 2*, pages 153–166. Morgan Kaufmann, 1992.

[33] E. Smorodkina and D. Tauritz. Toward automating ea configuration: the parent selection stage. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 63–70, Sept 2007.

[34] W. M. Spears. Adapting crossover in evolutionary algorithms. In *Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pages 367–384. MIT Press, 1995.

[35] L. Spector. Autoconstructive evolution: Push, pushGP, and pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 137–146. Morgan Kaufmann, 2001.

[36] L. Spector. Adaptive populations of endogenously diversifying Pushpop organisms are reliably diverse. In *Proceedings of Artificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems*, pages 142–145. The MIT Press, 2002.

[37] L. Spector. *Automatic Quantum Computer Programming: A Genetic Programming Approach*, volume 7 of *Genetic Programming*. Kluwer Academic Publishers, Boston/Dordrecht/New York/London, June 2004.

[38] L. Spector. Towards practical autoconstructive evolution: Self-evolution of problem-solving genetic programming systems. In R. Riolo, T. McConaghy, and E. Vladislavleva, editors, *Genetic Programming Theory and Practice VIII*, volume 8 of *Genetic and Evolutionary Computation*, pages 17–33. Springer New York, 2011.

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

[40] L. Spector. Clojush: The Push programming language and the PushGP genetic programming system implemented in Clojure. https://github.com/lspector/Clojush, 2016.

[41] L. Spector, K. Harrington, B. Martin, and T. Helmuth. What's in an evolved name? the evolution of modularity via tag-based reference. In R. Riolo, E. Vladislavleva, and J. H. Moore, editors, *Genetic Programming Theory and Practice IX*, Genetic and Evolutionary Computation, chapter 1, pages 1–16. Springer, 2011.

[42] L. Spector and T. Helmuth. Uniform linear transformation with repair and alternation in genetic programming. In R. Riolo, J. H. Moore, and M. Kotanchek, editors, *Genetic Programming Theory and Practice XI*, Genetic and Evolutionary Computation, chapter 8, pages 137–153. Springer, 9-11 May 2013.

[43] L. Spector and T. Helmuth. Effective simplification of evolved push programs using a simple, stochastic hill-climber. In *GECCO Comp '14: Proceedings of the 2014 conference companion on Genetic and evolutionary computation companion*, pages 147–148. ACM, 2014.

[44] L. Spector and J. Klein. Trivial geography in genetic programming. In T. Yu, R. L. Riolo, and B. Worzel, editors, *Genetic Programming Theory and Practice III*, volume 9 of *Genetic Programming*, chapter 8, pages 109–123. Springer, 12-14 May 2005.

[45] L. Spector, J. Klein, and M. Keijzer. The push3 execution stack and the evolution of control. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1689–1696. ACM Press, 2005.

[46] L. Spector, J. Klein, and C. Perry. Tags and the evolution of cooperation in complex environments. In *Proceedings of the AAAI 2004 Symposium on Artificial Multiagent Learning*. AAAI Press, 2004.

[47] L. Spector, J. Klein, C. Perry, and M. Feinstein. Emergence of collective behavior in evolving populations of flying agents. In *Genetic and Evolutionary Computation – GECCO-2003*, volume 2723 of *LNCS*, pages 61–73, Chicago, 12-16 July 2003. Springer-Verlag.

[48] L. Spector, J. Klein, C. Perry, and M. Feinstein. Emergence of collective behavior in evolving populations of flying agents. *Genetic Programming and Evolvable Machines*, 6(1):111–125, Mar. 2005.

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

[50] L. Spector and A. Robinson. Multi-type, self-adaptive genetic programming as an agent creation tool. In *GECCO 2002: Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference*, pages 73–80. AAAI, 2002.

[51] J. Tavares, P. Machado, A. Cardoso, F. B. Pereira, and E. Costa. On the evolution of evolutionary algorithms. In *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, pages 389–398. Springer-Verlag, 2004.

[52] F. Vafaee, W. Xiao, P. C. Nelson, and C. Zhou. Adaptively evolving probabilities of genetic operators. In *Seventh International Conference on Machine Learning and Applications, ICMLA '08*, pages 292–299. IEEE, 2008.

[53] J. R. Woodward and J. Swan. The automatic generation of mutation operators for genetic algorithms. In *GECCO 2012 2nd Workshop on Evolutionary Computation for the Automated Design of Algorithms*, pages 67–74. ACM, 2012.