Unifying Themes in Complex Systems

Volume VIII

Proceedings of the
Eighth International Conference on Complex Systems

New England Complex Systems Institute Book Series
NECSI Knowledge Press

Hiroki Sayama
Ali A. Minai
Dan Braha
Yaneer Bar-Yam
Editors

Compositional Autoconstructive Dynamics

Kyle Harrington
DEMO Lab, Brandeis University, Waltham, MA
kyleh@cs.brandeis.edu
Emma Tosch
DEMO Lab, Brandeis University, Waltham, MA
etosch@cs.brandeis.edu
Lee Spector
School of Cognitive Science, Hampshire College, Amherst, MA
lspector@hampshire.edu
Jordan Pollack
DEMO Lab, Brandeis University, Waltham, MA
pollack@brandeis.edu

Autoconstructive evolution is the idea of evolving programs through self-creation. This is an alternative to the hand-coded variation operators utilized in traditional genetic programming (GP) and the deliberately limited implementations of meta-GP. In the latter case strategies generally involve adapting the variation operators which are then used in accordance with traditional GP. On the other hand, autoconstruction offers the ability to adapt algorithmic reproductive mechanisms specific to individuals in the evolving population. We study multiple methods of compositional autoconstruction, a form of autoconstruction based on function composition. While much of the previous work on autoconstruction has investigated traditional GP problems, we investigate the effect of autoconstructive evolution on two problems: Order, which models order-sensitive program semantics, and Majority, which models the evolutionary acquisition of semantic components. In doing so we show that compositional autoconstruction exhibits surprising dynamics of evolutionary improvement, and that compositional autoconstruction can be comparable to GP. This advance is a step towards the search for the open-ended evolution of problem-solving techniques.

1 Introduction

Evolutionary computation is a collection of techniques inspired by evolution. Evolutionary algorithms (EA) operate on populations of individuals. At the heart of any EA there is a three step loop: evaluate, select, and vary. The details of these steps and the addition of intermediate steps constitutes the study of evolutionary computation. In this research we focus on genetic programming (GP) the evolution of computer programs [13]. GP is related to genetic algorithms (GA) a field that classically studies the crossover and mutation of fixed-length bitstring genomes. GP moves beyond the fixed-length bitstring representation and uses computer programs as genomes. A detailed review of GP is beyond the scope of this paper, for additional details see [13].

Meta-evolutionary techniques have been plagued by the "meta-meta-... problem" since the proposal of the first meta-evolution algorithm [16]. The "meta-meta-... problem" is: if a meta-population varies a problem solving population, what varies the meta-population? The simplest option is to add additional meta-layers. However, not only does the addition of meta-meta-populations not solve the problem, but it also leads to a significant cost in evaluating the performance of higher-order meta-populations. In practice the use of more than one layer of meta-evolution is avoided; instead, researchers investigate various levels of the evolutionary process.

For all intents and purposes, there are three levels at which adaptation may take place within an EA: populations, individuals, and components. Population-level adaptation generally involves update rules on evolutionary parameters such as mutation and crossover rates. Individual-level adaptation involves similar parametric adjustments; however, the parameters are unique to individuals. Component-level adaptation associates parameters with individual components that influence variation. The distinction between levels of adaptative evolution is fuzzy. Some instances of individual-level adaptation associate values with components within individuals that bias variation [2], while some forms of individual-level adaptation may have direct effects upon variation within the entire population [3]. The distinctions of these levels of adaptation have been introduced in detail [1]; however, parametric adaptation is not the only option.

The evolution of programs in GP involves the variation of symbolic expressions. Although some of the previously mentioned techniques involve meta-evolutionary processes working with GP, the first application of GP to itself was done in the context of a sea of GP programs [14]. In this work a program is capable of using template-based matching to search the sea of programs for a match that can be permanently incorporated the program's structure. Programs are solutions to simple boolean problems. In concluding, the researcher notes that although the probability of spontaneous emergence of productive self-replication is low, the approach of self-improving GP is in fact computationally tractable.

A notable piece of later work on meta-evolution in GP was meta-GP [5]. In this work, a population of variation operators (the meta-population) act on a population of problem solving programs. The variation operators are tree

manipulation programs that are capable of performing recombinatory variation by using two solution programs to create a new candidate solution. With this additional meta-population it was shown that meta-GP can perform comparably to GP. However, the meta-population is evolved with hand-coded algorithms. This use of human-designed variation operators acting upon the meta-population circumvents the complications of layered meta-populations, but puts an inherent limit upon the evolutionary process.

An advantage of the meta-GP approach is that it introduces less bias than most adaptive algorithms by evolving the operators of variation. However, the use of a standard EA on the meta-population is an additional constraint that is not required. Autoconstructive evolutionary algorithms (AEA) combine all levels of adaptive algorithms while avoiding the "meta-meta-... problem." Individuals in AEA contain their own variation code. When autoconstructing, a program may use itself, chosen, or selected individuals from the population as inputs when producing a child program. Self-directed selection has been explored for age- and geographic-based methods [17]. However, efficient autoconstruction continues to elude researchers [19].

We study a subset of autoconstruction called "compositional autoconstruction," where programs are autoconstructively composed by other programs for the production of offspring. Three forms of composition are explored: self-composition, asymmetric collision, and collision composition, which are discussed in detail later in the paper. In this work we present a study of the dynamics of autoconstructive evolution on two problems that model particular features of the evolution of program semantics. These problems are Order and Majority [8]. The Order problem is designed to model the organization of conditional branching, such as the true and false clauses of if-statements. On the other hand, the Majority problem models a general property of GP: the acquisition of more beneficial code than detrimental code. These problems serve to characterize the autoconstructive evolution by indicating its performance both on a problem solving task, and by demonstrating its evolutionary capabilities.

2 The Push Language

Push is a stack-based programming language specifically designed for evolutionary computation [22]. Instructions, such as arithmetic, program flow, and code manipulation instructions, take (pop) input from and output (push) to a set of global typed-stacks. Types include integers, floating-points, booleans, code, and tags (inexactly-matching identifiers that support a form of modularity [21]). A particularly innovative feature in Push is the *exec* stack [20]. The *exec* stack provides instructions for self-modification during execution through a stack. Aside from the content of the language itself Push has two primary properties that make it useful for evolution. First, instructions do not enforce a syntactic structure because of the use of global stacks for instruction inputs and outputs. Second, when the necessary inputs are not available on the stacks for an instruction, the instruction simply has no effect and execution continues

(the instruction acts as a "NOOP"). As opposed to reiterating a description of the language we present an example and direct the reader to the detailed descriptions of the language available in [22, 20, 21].

We begin by showing an example of simple arithmetic **(7 INTEGER.+ 3 INTEGER.* 0.5 INTEGER.DUP INTEGER.+)**. As Push programs are evaluated left to right we begin with **7**, which is pushed onto the integer stack. **INTEGER.+** is evaluated, but because there is only one value on the integer stack **INTEGER.+** has no effect. **3** is then pushed onto the integer stack. **INTEGER.*** pops **3** then **7** and pushes **21**. **0.5** is pushed onto the float stack. The instruction **INTEGER.DUP** takes the top item on the integer stack and pushes a copy of it onto the integer stack, so now there are two copies of **21** on the integer stack. Finally, **INTEGER.+** pops both copies of **21** and pushes **42**. The result of evaluating this expression is **0.5** and **42** on top of the float and integer stacks, respectively.

The PushGP evolutionary framework has been the primary mechanism for evolving problem solutions with Push. In general PushGP can act like any GP system, it is distinguished from standard GP primarily by its Push program-specific algorithms. Although many forms of variation have been implemented in PushGP, crossover, mutation, and replacement are the only forms of non-autoconstructive reproduction used in this study.

Autoconstruction was one of the primary considerations in the design of Push. The *code* type was introduced to allow for simple manipulation of programs and has been the primary datatype involved in autoconstruction studies thus far [22, 19]. The *code* type implements a large number of instructions, many of which are inspired by Common Lisp. This diverse autoconstructive vocabulary has led to a number of interesting solutions presented in the previously mentioned studies. However, the combinatorics of evolving programs are exponentially unfavorable with respect to the number of possible instructions, even with Push's evolution-friendly design.

The most recent version of Push has been developed in the programming language Clojure, a mostly-functional version of Lisp that runs on the Java Virtual Machine [9]. Clojure has revisited the idea of the zipper, a functional data structure that represents a location within a tree [11]. A zipper allows for simple tree traversal with directional commands. Zippers have been incorporated into Push as a reified data type with their own stack. In addition to the traditional zipper movements, instructions are added that allow for subtree manipulation and random movement. Although a larger number of zipper instructions have been added to Push, the subset used in this study are shown in table 1. The simplicity of these tree-based instructions facilitates autoconstruction.

3 Compositional Autoconstruction

We use the phrase "compositional autoconstruction" here to emphasize a view of autoconstruction as something akin to function composition, in which programs

Instruction	Description
ZIP.DOWN	Move the top zipper deeper in the tree, if possible.
ZIP.LEFT	Move the top zipper to the left sibling, if possible.
ZIP.RIGHT	Move the top zipper to the right sibling, if possible.
ZIP.RAND	Replace the subtree rooted at the top zipper's location with a random subtree.
ZIP.ROOT	Move the top zipper to the root of the tree.
ZIP.RLOC	Move the top zipper to a random location in the subtree rooted at the top zipper.
ZIP.RRLOC	Move the top zipper to a random location in the tree.
ZIP.SWAP	Pop two zippers and push them back onto the stack in reverse order.
ZIP.SWAPSUB	Swap the subtrees rooted at the top two zippers.
INTEGER.ERC	An ephemeral random integer in: $[-16, -1] \cup [1, 16]$.
EXEC.NOOP	A placeholder instruction used for "padding" that has no effect.

Table 1: Instruction set used in this study.

are applied to programs in order to produce new programs.[1] This usage follows that of Fontana [7], and it also highlights connections to the "compositional evolution" work of Watson [23], which emphasizes additional forms of biological composition such as symbiogenesis.

In the more specific context of autoconstructive evolution "compositional autoconstruction" emphasizes the nature of the inputs and outputs to programs that produce offspring in an autoconstructive population. In particular, we will consider programs that are applied *only* to their own code in order to produce offspring (without having access to other programs in the population), programs that are applied *only* to the code of single, randomly selected programs in the population to produce offspring (without having access to their own code), and programs that are applied both to their own code and to the code of other single programs in the population. In contrast, many previous autoconstructive evolution systems have allowed arbitrary access to any code in the population [17, 18], while others have restricted access but without the systematic delineation of access restrictions that we present here [19].

In the present work, because we are interested in studying the self-organized dynamics of autoconstructive populations, we also avoid most of the constraints on reproductive behavior that have been used in prior systems (e.g. the "improvement-based" constraints on reproduction in [19]). The only constraint

[1]Note that this is somewhat different from the standard mathematical usage in which composition refers to the application of a function to the output of another function. In compositional autoconstruction this kind of composition does occur across generations, with the outputs of programs (which are themselves programs) serving as the input to the programs of the next generation. But within the reproductive processes of a single generation programs are applied to the *code* of other programs, not their outputs.

on autoconstruction in this work, aside from the constraint that programs cannot exceed the global size limit, is a "no cloning" rule: a child may not have the exact same program as its parent.

Furthermore, for all experiments presented in this study the zipper instructions have no direct interaction with the problem-specific instructions, and vice versa. This is because both the Order and Majority problems are scored via inspection as opposed to evaluation. This may help to facilitate autoconstruction, and in any event it helps us to focus our study on the dynamics of autoconstructive populations by allowing autoconstructive and problem-specific instructions to be mostly independent. Although some of the more ambitious goals of autoconstruction involve code-aware instructions (see [17, 19]), we show that autoconstruction can have acceptable performance with just simple tree-based instructions.

3.1 Self-composition

Self-composition is one of the most basic forms of compositional autoconstruction. A similar approach is taken in [19] where Push's *code* stack is used with a large instruction set. This type of autoconstruction is a type of mutation. During self-composing autoconstruction a program can perform mutations at particular locations in a copy of its program in order to create a child. The reproduction rule for self-composition can be written as

$$f(f) = h$$

where f is the parent and h is the child. This form of autoconstruction is entirely self-contained within an individual, and for this reason there is no external dependency on the GP population. The next form of autoconstruction we consider is asymmetric collision, which introduces a population-level dependency.

3.2 Asymmetric Collision

Asymmetric collision compositional autoconstruction is inspired by the AlChemy system [7]. In the AlChemy system a collection of objects (programs) are contained within a reactor. The system evolves by iteratively selecting two objects at random which "interact." Interaction occurs via function composition, the result of which is subjected to a collision rule. In practice the collision rule is used to define boundary conditions, such as evaluation limit (to ensure termination) and replication of input functions. In our system asymmetric collisions follows essentially the same algorithm. Two programs are selected from the population and one is composed by the other. Asymmetric collisions introduce indirect interaction amongst individuals within a population. Asymmetric collision composition can be expressed as

$$f(g) = h$$

where f is the active parent, g is the passive parent, and h is the child. Like self-composition, asymmetric collision composition is a form of mutation. The only code available to the autoconstructive process is **g**s code and code obtained from the random code generator via ZIP.RAND. In the next section we introduce collision compositional autoconstruction, which gives programs the ability to exchange genetic information during the autoconstructive process.

3.3 Collision

Collision compositional autoconstruction is a recombinatory form of autoconstruction. We present a simple binary form of collision compositional autoconstruction in the form

$$f(g, f) = h$$

where f is the active parent, g is the passive parent, and h is the child. The composition of two programs allows exchange of genetic information between the two programs. The importance of recombination in evolutionary computation has been emphasized in the fields of genetic algorithms and genetic programming [10, 13]. We now go on to explore the properties of these forms of compositional autoconstruction.

4 Autoconstruction of Program Semantics

The Order and Majority problems are model problems for studying the evolution of program semantics in genetic programming [8]. The Order problem is designed to model conditional structures in programs, such as the ordering of true and false clauses of if-statements. The Majority problem models evolutionary accumulation of semantically important components. An analysis of the hardness of these two problems is presented in [4]. When originally presented these problems were used to model the effects of different types of variation operations. Within the context of Push and autoconstructive evolution both problems are of interest; however, the implications of these problems in Push are slightly different. These implications are presented in the relevent sections.

For both problems the problem size was 16. The lowest fitness possible is 0, while the highest fitness is 16. The problem-specific "instructions" are just integers in the ranges $[-16, -1]$ and $[1, 16]$; no other Push instructions are used aside from the ZIP instructions listed in table 1 and the "padding" instructions described below. The positive integers represent components that should occur earlier during evaluation in the Order problem, and beneficial components in the Majority problem. The negative integers represent detrimental components in both problems. In these experiments the autoconstructive configurations are compared to one another and to standard PushGP, using instruction padding (the inclusion of NOOP instructions) in any configuration with less than the maximum number of instructions of any configuration. This is done to make

Parameter	Value
Selection method	tournaments of 5
Population size	1000
Maximum program size	500
Maximum number of generations	1001
Number of runs per condition	250

Table 2: Parameters used.

the combinatorics of the problem spaces equivalent[2]. The parameters used in the following experiments are presented in table 2. Performance of parameter settings are measured with the median success generation and computational effort. Median success generation is the generation in which an individual in the population achieves a fitness of 16. Computational effort is a metric developed for GP [13] to quantify the amount of computation required to solve a problem. It is computed by estimating the probability of a population containing a success at a given generation. This probability distribution is then used to compute how many individuals are required to solve a problem with a certain probability, z. In this study $z = 99\%$. The other metric used is diversity, which is ratio of the number of distinct programs to the total number of programs in the population. In this study we show that compositional autoconstruction can be comparable to GP, which means that the computational effort and median success generation of autoconstruction are on par with GP.

4.1 Order

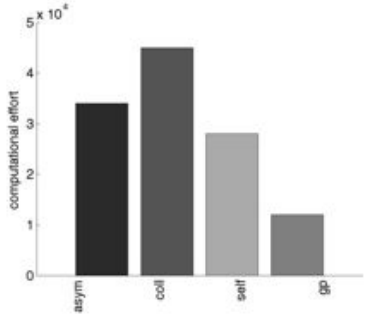
The stack-based nature of Push means that the Order problem represents any order-specific binary stack-based instruction, such as division, subtraction, conditionals, loops, code manipulation, etc.. Although it is possible that intermediate instructions could interfere with stack contents (i.e. the addition/removal of stack items), we do not introduce distance between complimentary instructions.

The fitness of an Order solution, $r(f)$, is computed as

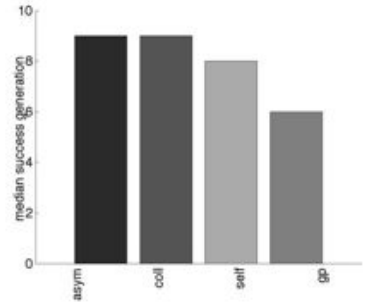
$$\begin{aligned} \forall z_i \in [1, s], \quad & if M(z_i, f) < M(-z_i, f) \quad v_i = 1 \\ & otherwise \qquad\qquad\quad v_i = 0 \\ r(f) = \sum_i v_i & \end{aligned} \tag{1}$$

where s is the problem size, $M(z, f)$ is the number of elements preceeding the first occurrence of z within program f in depth-first order. An example of program is **(ZIP.RRLOC (3 (5) (16 (10) ZIP.RAND 10) -8) -6 (-8 (ZIP.RIGHT 13)) (-1 ((-11 (-6 13 (9)) -15) 8 (12) 8 -15) 10)**

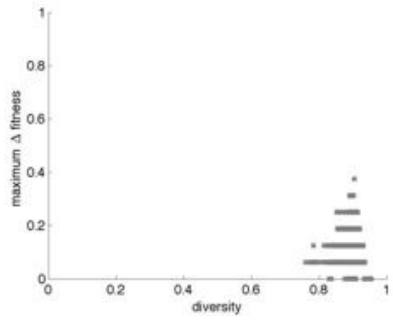
[2]Collision composition uses 2 additional instructions (**ZIP.SWAP** and **ZIP.SWAPSUB**) not accounted for in the instruction set padding.



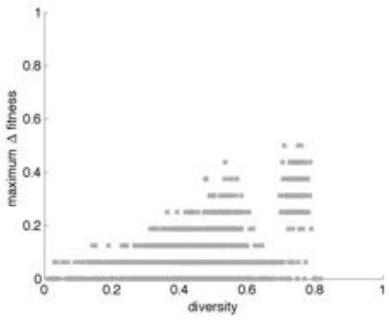
(a) Computational effort



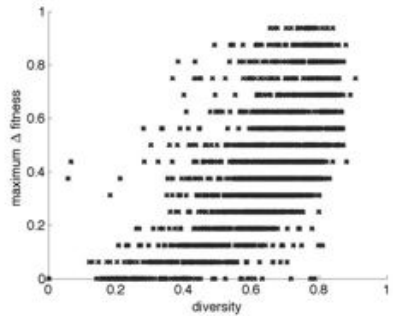
(b) Median success generation



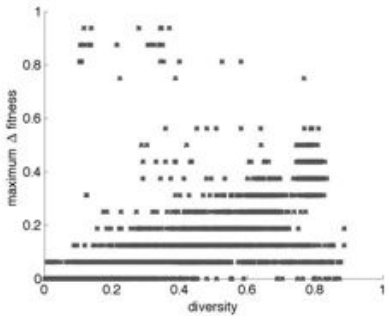
(c) GP



(d) Self-composition



(e) Asymmetric Collision



(f) Collision

Figure 1: Comparison of compositional autoconstruction to GP on Order. Fitness changes are normalized to range from 0 to a maximum of 1.

(ZIP.RLOC) (ZIP.ROOT) 12). This autoconstructive mutation program has fitness of 7 and produces a child using standard subtree replacement mutation (which is implemented by the first two zipper instructions, with the subsequent zipper instructions having no effect).

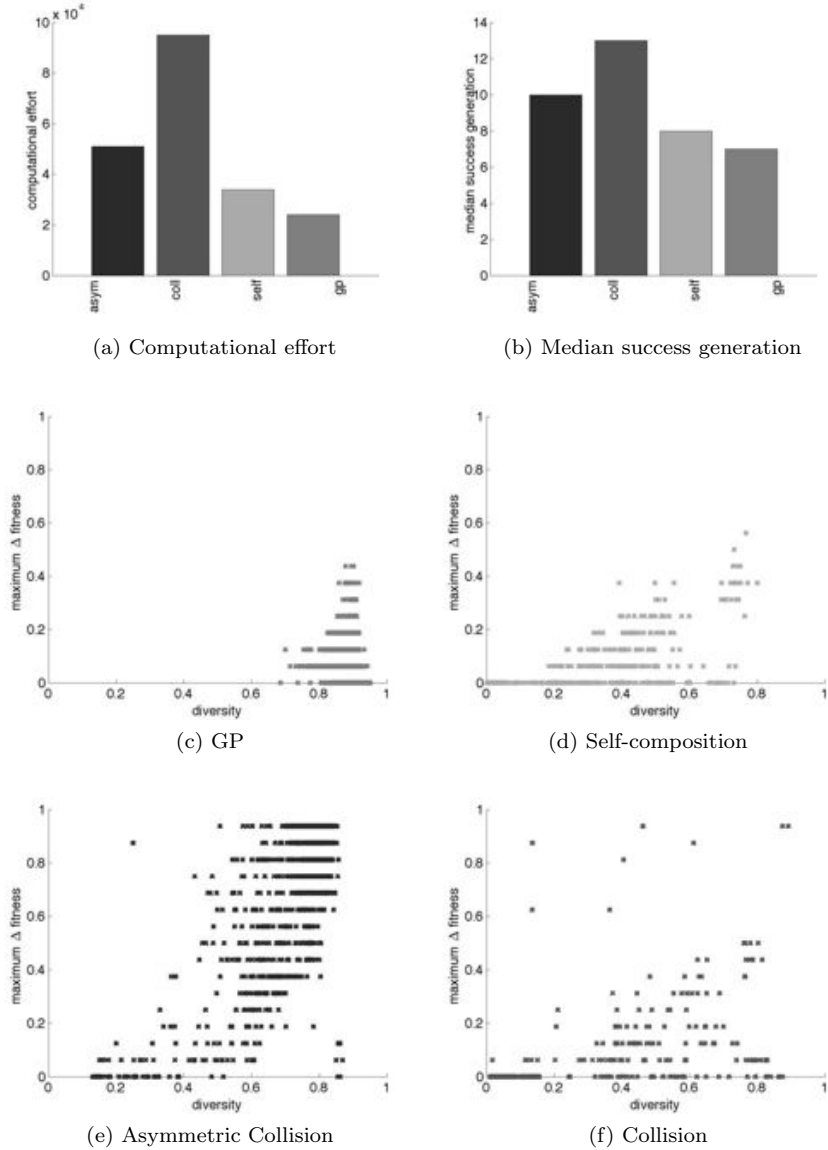
Overall performance on the Order problem is shown in figure 1. It is clear from the computational effort scores that Order is a fairly easy problem for all of the evolutionary techniques to solve. While GP outperforms autoconstruction in computational effort, autoconstruction performs similarly when considering the median success generation. Autoconstruction's higher metrics are not surprising for two reasons. First, good variation operators are used consistently, beginning with the first generation in GP, while the autoconstructive components of programs are initialized randomly. Second, Order is a simple problem that standard GP operators are known to work well on. However, this should not discount from the informativeness of this model problem.

Fitness dynamics for the Order problem are shown in figures 1c-1f. Each point represents a generation in one of the experimental conditions. In the standard GP condition (figure 1c) there are moderate increases in fitness with a high and consistent level of diversity. The consistently high level of diversity is due to the variation operators used in GP. Generally solutions are not repeatedly evaluated due to the small changes that frequently do not significantly decrease the fitness of individuals, allowing them to survive in subsequent generations. Figure 1d shows the performance of self-composition, which sometimes produces larger improvements in fitness, but suffers from lower levels of diversity. Of particular note is the cluster of generations with high diversity that have large improvements in fitness. This may either be due to the ease of improving the initially random and diverse population or speciation. In figure 1e asymmetric collision compositional autoconstruction is shown. This form of autoconstruction exhibits particularly interesting dynamics. There are many changes in fitness that lead to near optimal and optimal individuals. This method of autoconstruction also appears to maintain consistent levels of diversity, albeit not as consistently high as standard GP. Collision compositional autoconstruction (figure 1f) behaves similarly to self-composition while producing outliers with large improvements in fitness that are similar to the results of produced by asymmetric collisions. However, this form of binary collision appears to suffer from complications in diversity. This is most likely due to its relation to self-composition. Potential explanations are suggested in the discussion.

The Order problem is designed to model order-based semantic dependencies within programs. As such, Order primarily serves to model the ability of an EA to produce problem solutions. The results support the hypothesis that autoconstruction is in fact capable of facilitating the search for problem solutions in ways that are notably different than traditional GP. Now we consider the Majority problem, which models properties of the evolutionary process itself.

4.2 Majority

The Majority problem awards fitness for maintaining at least as many beneficial components as detrimental components. Majority has been described as a weak property of GP solutions. The primary property modeled by Majority in Push is the ability of a program to have the capability to push a sufficient amount of



(a) Computational effort

(b) Median success generation

(c) GP

(d) Self-composition

(e) Asymmetric Collision

(f) Collision

Figure 2: Comparison of compositional autoconstruction to GP on Majority. Fitness changes are normalized to range from 0 to a maximum of 1.

correct information onto the stacks. This is of interest for considering the probability that an individual can be used to produce a new and better individual. In the context of autoconstruction this is a particularly important question: how well does autoconstruction improve individuals?

The fitness of a Majority solution, $r(f)$, is computed as

$$\begin{aligned} \forall z_i \in [1,s], \quad if N(z_i,f) \geq N(-z_i,f) \wedge N(z_i,f) > 0 \quad & v_i = 1 \\ otherwise \quad & v_i = 0 \\ r(f) = \sum_i v_i \end{aligned} \tag{2}$$

where s is the problem size, $N(z,f)$ is the number of instances of integer z in program f. The example program presented in the previous section on Order has a Majority fitness of 1.

Overall performance of Majority is shown in figures 2a and 2b. Self-composing autoconstruction performs comparably to GP, both in terms of computational effort and median success generation. This is most likely because the Majority problem is focused on programs that have more of a certain set of components. This implementation of the problem simply has 32 problem-specific components, all in the form of terminals. Many programs in the population will contain all these components. In such situations, a self-composing autoconstructive program can easily evolve to preserve a beneficial set of components.

Fitness dynamics for the Majority problem are shown in figures 2c-2f. The performance of standard GP on Majority (figure 2c) is similar to that seen on the Order problem. A high level of diversity is maintained with moderate improvements in fitness. In figure 2d, the performance of self-composition on the Majority problem appears to be markedly poorer than on the Order problem when noting the low levels of diversity and moderate improvements in fitness. However, as can be seen in both the computational effort and median success generation, self-composition is still the best performing method of compositional autoconstruction. Also, there is the recurrence of the distinct cluster of generations where at a relatively high level of diversity large improvements in fitness are made. Asymmetric collisions (figure 2e) follow a similar trend to the one seen on the order problem, with a higher density of generations being at high levels of diversity with large improvements in fitness. Collision compositional autoconstruction (figure 2f) performs particularly poorly on the Majority problem. This is surprising, due to the ability of this form of composition to mimic both self-composition and asymmetric collision. Nevertheless, aside from a number of outliers collision composition leads to lower levels of diversity and produces smaller increases in fitness, even than standard GP. It is clear from these results that self-composition possesses some unique capabilities to perform well under certain circumstances, yet suffers from complications of diversity. On the other hand, asymmetric collisions tend to support higher levels of diversity while producing large improvements in fitness.

5 Discussion

We show that autoconstruction can perform comparably to GP on problems that model program semantics. The initial randomization of autoconstructive

programs should slow evolution, but surprisingly the extent to which it does so appears to be reasonably bounded. One of the primary expectations of autoconstruction is that it will be of greatest use for difficult problems for which standard GP fails to find solutions. While this may still be true because GP does outperform autoconstruction, the comparable performance of autoconstruction on problems of program semantics elucidates the benefits of autoconstruction. Our data shows that the AEAs considered here often produce larger jumps in program space than traditional GP search, which can be seen in the large increases in fitness produced by compositional autoconstruction.

On the Order problem we observe that compositional autoconstruction can generally find successful programs in a similar number of generations to standard GP. This observation is substantated by median success generations. Computational effort metrics are important to present because of their use as a gold-standard of GP performance. However, when autoconstruction is not subjected to techniques for reinforcing diversity it is not uncommon for diversity to crash within a population. For this reason it is important to primarily draw conclusions based upon the median success generation results.

The consistent diversity and significant improvements seen for asymmetric compositional autoconstruction in both the Order and Majority problems is of particular interest to practitioners of autoconstruction. The only difference between asymmetric collisions and self-composition is the subject of the composition. However, because of the implementation of these problems there is little linkage between autoconstructive functionality and the inspection-based fitness score. Thus, the primary difference between self-composition and asymmetric collisions is the source of the input during autoconstruction. This leads us to suggest that not only can the diversity in a population be increased by the application of a wide range of variation operators, but significant improvements in fitness can also be achieved with this strategy. It may be possible to further improve the performance of co-dependent autoconstruction mechanisms by increasing the amount of children produced within any given generation.

Aside from a no-cloning rule we do not introduce any reproduction-based selection pressures on the population. This decision is justified by the previous finding of the diversification capabilities of autoconstructive evolution [18] and the goal of studying autoconstruction in its most simple form. We suggest that crowding-based techniques can be used as a method for increasing diversification. Alternatively, autoconstructive improvement metrics [19] can be treated as additional objectives for use with coevolutionary techniques [12, 6, 15].

One reason that the autoconstructive instruction set used here may outperform instruction sets used in previous studies is that the zipper-based instructions provide the ability to express powerful operations with a small set of operators. This allows for the representation of useful variation operators with relatively few instructions. That being said, the combination of *code*-stack based autoconstruction and zipper instructions may support even more powerful forms of autoconstructive evolution

We have shown that compositional autoconstruction can perform compara-

bly to GP and that some forms of autoconstruction produce significantly higher improvements in fitness. By exploring problems that model program semantics we can see that autoconstruction supports the evolutionary acquisition of beneficial components, as well as the ordering of problem-solving components. By allowing algorithmic mechanisms of variation to evolve within individuals the dynamics of GP-comparable compositional autoconstruction is a step closer to the open-ended evolution of problem-solving techniques.

6 Acknowledgements

We thank the DEMO lab and the Hampshire College CI lab. This material is based upon work supported by the National Science Foundation under Grant No. 1017817 and 0757452. 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. Thanks also to Hampshire College for support of the Hampshire College Institute for Computational Intelligence.

Bibliography

[1] Angeline, P.J., "Adaptive and self-adaptive evolutionary computations", *Computational Intelligence: A Dynamic Systems Perspective*, (1995).

[2] Angeline, P.J., "Two self-adaptive crossover operations for genetic programming", *Advances in genetic programming*, (1995).

[3] Angeline, P.J., and J.B. Pollack, "The evolutionary induction of subroutines", *Proc. of the 14th Annual Conf. of the Cognitive Science Society*, Lawrence Erlbaum (1992), 236–241.

[4] Durrett, G., F. Neumann, and U.M. O'Reilly, "Computational complexity analysis of simple genetic programming on two problems modeling isolated program semantics", *Proc. of the 11th Workshop Proc. on Foundations of Genetic Algorithms*, (2011).

[5] Edmonds, B, "Meta-genetic programming: Co-evolving the operators of variation", *Turk J. Elec. Engin* (2001).

[6] Ficici, S., and J. Pollack, "Pareto optimality in coevolutionary learning", *Advances in Artificial Life* (2001), 316–325.

[7] Fontana, W, "Algorithmic chemistry", *Artificial life II* (1991).

[8] Goldberg, D., and U.M. OReilly, "Where does the good stuff go, and why? How contextual semantics influences program structure in simple genetic programming", *Genetic Programming* (1998), 16–36.

[9] Hickey, R., "The Clojure programming language", *Proc. of the 2008 Symp. on Dynamic Languages*, (2008), 1.

[10] Holland, J.H., *Adaptation in Natural and Artificial Systems*, (1975).

[11] Huet, G., and I.R. France, "Functional pearl: The zipper", *J. Functional Programming*, (1997).

[12] Juille, H., and J.B. Pollack, "Co-evolving intertwined spirals", *Proc. of the 5th Annual Conf. on Evolutionary Programming*, (1996).

[13] Koza, JR, *Genetic programming: on the programming of computers by means of natural selection*, (1992).

[14] Koza, J.R., "Spontaneous emergence of self-replicating and evolutionarily self-improving computer programs", *Artificial life III*, vol. 17, Citeseer (1994), 225–262.

[15] Popovici, E., A. Bucci, R.P. Wiegand, and E.D. de Jong, "Coevolutionary Principles", *Proc. of the 11th Workshop Proc. on Foundations of Genetic Algorithms*, (2011).

[16] Schmidhuber, J., *Evolutionary principles in self-referential learning. (On learning how to learn: The meta-meta-... hook.)*, PhD thesis Institut für Informatik, Technische Universität München (1987).

[17] Spector, L., "Autoconstructive evolution: Push, pushGP, and pushpop", *Proc. of the Genetic and Evolutionary Computation Conference*, (2001), 137–146.

[18] Spector, L., "Adaptive populations of endogenously diversifying pushpop organisms are reliably diverse", *Proc. of Artificial Life VIII*, (2002), 142.

[19] Spector, L., "Towards Practical Autoconstructive Evolution: Self-Evolution of Problem-Solving Genetic Programming Systems", *Genetic Programming Theory and Practice VIII* (2010), 17–33.

[20] Spector, L., J. Klein, and M. Keijzer, "The Push3 execution stack and the evolution of control", *Proc. of the 2005 Conf. on Genetic and Evolutionary Computation*, ACM (2005), 1689–1696.

[21] Spector, L., B. Martin, K. Harrington, and T. Helmuth, "Tag-based modules in genetic programming", *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2011)*, (2011).

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

[23] Watson, R.A., *Compositional Evolution: Interdisciplinary Investigations in Evolvability, Modularity, and Symbiosis*, PhD thesis (2002).