Autoconstructive Evolution for Structural Problems

Kyle I. Harrington
DEMO Lab
Brandeis University
Waltham, MA, USA
kyleh@cs.brandeis.edu

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

Jordan B. Pollack
DEMO Lab
Brandeis University
Waltham, MA, USA
pollack@brandeis.edu

Una-May O'Reilly
EvoDesignOpt
MIT
Cambridge, MA, USA
unamay@csail.mit.edu

ABSTRACT

While most hyper-heuristics search for a heuristic that is later used to solve classes of problems, autoconstructive evolution represents an alternative which simultaneously searches both heuristic and solution space. In this study we contrast autoconstructive evolution, in which intergenerational variation is accomplished by the evolving programs themselves, with a genetic programming system, PushGP, to understand the dynamics of this hybrid approach. A problem size scaling analysis of these genetic programming techniques is performed on structural problems. These problems involve fewer domain-specific features than most model problems while maintaining core features representative of program search. We use two such problems, Order and Majority, to study autoconstructive evolution in the Push programming language.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program synthesis*

General Terms

Algorithms

Keywords

Autoconstruction, Structural problems, Order, Majority, Push, PushGP

1. INTRODUCTION

Heuristics are "rules of thumb" that can improve the search for solutions within a class of problems [23, 6]. Evolutionary computation has been used to evolve heuristics for several kinds of problems including path functions for the traveling salesman problem [21] and bin evaluation functions for bin packing problems [5]. These heuristics are often employed within problem-solving algorithms that are otherwise quite minimal; for example, when solving the traveling salesman problem a simple loop may be used to iteratively call a heuristic function to choose the next node [21]. But evolutionary computation systems themselves embody heuristics of several kinds; for example, the variation operators used for mutation and crossover can be considered heuristics. Just as evolutionary computation can be used to evolve heuristics in other domains, it can also be used to produce or refine its own heuristics; methods for doing this are called hyper-heuristics. In this study we investigate the scaling properties of a particular hyper-heuristic, called autoconstructive evolution, on structural problems.

Autoconstructive evolution is an evolutionary computation technique that encodes mechanisms for reproduction and variation in the genomes of individual problem solvers. We examine the capabilities of autoconstructive evolution on two problems, Order and Majority, which are designed to model representative properties of genetic programming (GP) algorithms [11]. Order programs are scored based upon their ordering of complimentary pairs of terminal instructions in a depth first traversal of the programs' code. Majority programs are scored based upon the relative quantity of complimentary pairs of terminal instructions. The essential properties of these problems that make them interesting in the context of autoconstruction are: multiple optima in the fitness landscape (Majority/Order), accumulation of beneficial components (Majority), and eliminative transformations (Order).

Many interesting problems have multiple optima, making it important for general search methods to be able to handle such problems. In addition to the shape of the fitness landscape, the way the landscape is traversed is also important. When traversing a program space the accumulation of beneficial components is crucial for two reasons. Firstly, for some problems the evolution of programs involves progressive refinement of sub-expressions (i.e. accumulating numeric constants in symbolic regression and increasingly descriptive boolean expressions for parity and multiplexers). Secondly, programs that contain beneficial components may serve to act as enhanced donors of code.

GECCO'12, July 7-11, 2012, Philadelphia, Pennsylvania, USA.

While the Majority problem serves as a model for the accumulation of beneficial components, the Order problem is based upon the idea of eliminative transformation. An eliminative transformation is a change whereby the relative ordering of a component may enable or disable the functionality of other components. For example, in the expression: "*if* s_1*, then* s_2*, else* s_3" when s_1 evaluates to $false$, s_2 is disabled. By changing the relative ordering of s_1, s_2, and s_3 a different component may be disabled, such as in the case: "*if* s_2*, then* s_1*, else* s_3" when s_2 evaluates to false; here s_3 will change from being enabled to being disabled even though it was not moved by the reordering. Eliminative transformations commonly appear in the control flow of programs as *if*, *while*, and other branching or looping statements. As Goldberg and O'Reilly note, eliminative transformations are related to program bloat, a common problem in GP.

Order and Majority model different, but important features of general program search. The size of both problems can be easily rescaled to allow for analysis with respect to program size. While many model problems exist that introduce features of search (i.e. largest common subsequence-based fitness [12] and context-sensitivity [32]), we believe that these model structural problems facilitate the study of critical properties of variation in evolutionary algorithms with a minimal set of features of program search.

The purpose of model GP problems is to provide insights into evolutionary mechanics. In GP we use mutation and crossover as evolutionary mechanisms which traverse the search space by moving towards programs of increasing correctness, and/or that have a higher probability of leading to programs of increasing correctness. Order and Majority can provide insights into evolutionary mechanisms as well as program structure assembly. The complexity of solving these problems has been previously described [7], but when considering alternative approaches to variation the evolutionary dynamics are of particular interest. The performance of a variation algorithm in solving a given structural problem may be indicative of its capabilities for solving problems with properties similar to the respective model problem. By modeling accumulative expressions, Majority provides insight into how well a variation operator promotes beneficial genetic material. On the other hand, Order can be used to discern how well an algorithm varies programs to discover beneficial eliminative transformations.

The representative properties of these two problems make them ideal candidates for studying alternative approaches to variation in program search. In this work our baseline GP algorithm is PushGP, a stack-based GP system. In its full extension it is also *autoconstructive*. That is, rather than relying upon a pre-specified mechanism for genetic inheritance and variation, each genome encodes its own reproduction process. In particular we compare autoconstructive evolution, in which intergenerational variation is accomplished by the evolving programs themselves, to PushGP, a standard genetic programming system, on both Order and Majority as the problems are scaled in size.

2. BACKGROUND

In GP, work on the adaptation of variation has largely focused on parametric adaptation and meta-populations. Self-adaptive EAs, which historically started as meta-EAs [24] with the rationale that parameters and search mechanisms could themselves adapt, are related [31, 9, 2, 20, 10]. However, most research on adaptive EAs investigate the tuning of population-level parameters, and work on self-adaptive EAs generally investigates the tuning of individual-level parameters. A unique approach to self-adaptive EAs operating at the component-level is presented in [1], where probabilities of selection/variation are associated with nodes in a program tree. When subtrees are swapped between programs the respective probabilities are also swapped, thus the node selection probabilities and candidate solution programs are subjected to the same evolutionary pressures. Yet parametric adaptation focuses on techniques for tweaking parameters that are used by human-coded variation operators.

Evolving variation programs, meta-GP, was introduced in [29] where a population of variation operators is coevolved to select subtrees for recombination. This work was expanded in the context of graph-GP to introduce both a second meta-level and the ability to perform mutations [16]. Both Teller and Kantschik et al., demonstrate that meta-GP can outperform a traditional GP variation algorithm. Edmonds extended meta-GP from the evolution of node selectors to entire tree-manipulation programs [8]. Unlike node selection-based meta-GP, Edmonds' implementation does not surpass traditional GP in performance. Recently we have shown that zipper-based tree-manipulation languages are capable of expressing variation operators that outperform traditional mutation and crossover operators in a meta-GP system [14]. These meta-GP implementations are more closely related to adaptive EAs than self-adaptive EAs, where evolutionary history informs future variations at the population-scale.

Autoconstruction unifies meta-GP and self-adaptation by integrating the reproductive mechanism into the genome [28]. The incorporation of reproduction into the genomes of problem solvers was first presented in [18], where a "sea" of computer programs incrementally self-improve to solve simple boolean problems. This integration of reproduction into the genome makes autoconstructive evolution both an individual- and component-based meta-evolutionary technique. This is different than adaptive EAs which vary population-level parameters, and most self-adaptive EAs which generally vary parameters at the individual level. While the autoconstructive approach is in some respects both elegant and analogous to biology it leads to a number of complexities. Problem-specific instructions intermingle with variation instructions within individual programs. In the majority of cases this requires programs to utilize multiple data types (e.g. solving symbolic regression requires numeric instructions for solving the regression problem and code manipulation instructions for producing offspring). Programs are also expected to evolve both a beneficial variation operator and a solution, which could arguably require more evolutionary time than simply evolving a solution. Up to this point autoconstruction has not been found to outperform a traditional GP algorithm [25, 13], but evidence from biology provides reasons to believe that in the long run, when autoconstructive processes are better understood, these systems will outperform methods based on hand-designed reproductive methods.

A practical way to study the capabilities of autoconstruction is to use structural problems which posess many of features of typical GP problems, yet are simple, quickly solvable, and scalable. In this paper we show how a zipper-based

language for autoconstruction performs relative to PushGP on large structural problems.

3. AUTOCONSTRUCTION AS A HYPER-HEURISTIC

Hyper-heuristics can be described as "the process of using (meta-)heuristics to choose (meta-)heuristics to solve the problem in hand" [4]. Although autoconstruction is not like most hyper-heuristics due to the dualism of function encoded in evolved individuals (variation and solution), it clearly falls in the realm of hyper-heuristics. In fact, autoconstructive evolution is a hyper-heuristic in two ways: reproductive mechanisms are evolved which are then used to vary problem solutions, and reproductive mechanisms vary the reproductive mechanisms. Yet one key difference between autoconstruction and most hyper-heuristics is the entanglement of the heuristic and the problem solution within individual programs. In most cases this leads to the solution and heuristic being inseparable. However, if reusability is a desirable characteristic for a particular class of problems, then it is often the case that the individual program may be used as an independent heuristic by treating the program as a variation operator.

4. GENETIC PROGRAMMING WITH THE PUSH LANGUAGE

Push is a stack-based programming language specifically designed for evolutionary computation [26]. Instructions, such as arithmetic, program flow, and code manipulation instructions, take (pop) input from and output (push) to a set of typed, global stacks. Types include integers, floating-point numbers, booleans, and code. Among the properties of Push that make it useful for evolutionary computation is the lack of any syntactic restrictions, aside from the balancing of parentheses, on program structure. This is possible because the values are passed among instructions by means of global data stacks, and it is therefore not necessary for the flow of function arguments and return values to be indicated in a program's syntactic structure. When necessary inputs for an instruction are not available on the stacks, 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 available in [28, 26, 27].

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.

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 [28, 25]. Evaluating autoconstructive programs for problem fitness is performed in the same manner as evaluating a Push program in any other context and irrelevant outputs are ignored. 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 instructions, even with Push's evolution-friendly design.

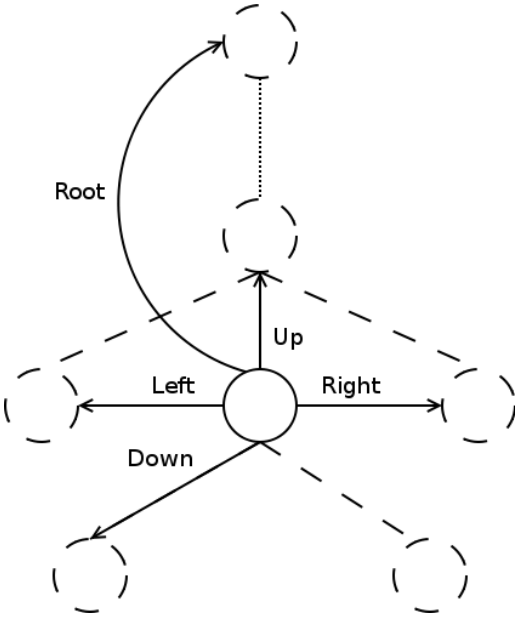


Figure 1: A diagram of a zipper.

5. ZIPPER-BASED AUTOCONSTRUCTION

The autoconstructive instruction set used in this research is based upon zippers. Zippers are functional data structures that represent locations within trees and allow for simple tree traversal with directional movement and editing commands [15]. A diagram of a zipper is shown in Figure 1. Zippers have been incorporated into Push as a first-class data type with their own stack. In addition to the traditional zipper movements, instructions are added that allow for multi-zipper manipulation, random movements, and random subtree generation. 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 and small number of these tree-based instructions facilitates autoconstruction.

We use a deliberately constrained approach to autoconstruction. In previous work we explored the effect of access restrictions in autoconstruction by considering applying programs to themselves, randomly selected individuals, and themselves in conjunction with other individuals [13]. This led to the identification of two forms of autoconstruction of particular interest. The first is autoconstructive mutation (AM) where individuals are composed with other randomly selected individuals. The second is autoconstructive crossover where individuals are composed with themselves

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 within the subtree pointed to by 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.SSUB	Swap about the subtrees pointed to by the top two zippers.
INTEGER.ERC	An ephemeral random integer in: $[-size, -1] \cup [1, size]$ where $size$ is the respective problem size.
EXEC.NOOP	A placeholder instruction used for "padding" that has no effect.

Table 1: Instruction set used in this study.

and other randomly selected individuals (AX). In AM, a child f' is created by parents f and g by the expression $f' = f(g)$, and in AX, $f' = f(f, g)$. In practice a program f performing AM on a program g is evaluated with program g as the top item on the zipper stack. After evaluation the top item on the zipper stack is popped off and returned as the child program, f'. A program f performing AX on program g is evaluated with f and then g pushed onto the zipper stack, and again the top item on the zipper stack is popped off and returned as the child program, f'. If f does not contain instructions that modify the top item on the zipper stack, then the resulting child will be a clone of the input. This problem of cloning is prevalent in autoconstruction, especially in early generations. The most common resolution to this problem is the enforcement of "no cloning" rules.

Here we present an example of zipper-based autoconstructive crossover where $f = g =$ **((ZIP.DOWN ZIP.RIGHT ZIP.DOWN ZIP.RIGHT) (12 18 INTEGER.+) ZIP.SWAP ZIP.DOWN ZIP.RIGHT ZIP.SSWAP))**. The Push interpreter is initialized with f and then g pushed onto the zipper stack. The first part of the program to be evaluated, **(ZIP.DOWN ZIP.RIGHT ZIP.DOWN)**, moves the top zipper to **18**. **ZIP.SWAP** swaps the top two items on the zipper stack, and now the top zipper points to the root of g. The instructions **ZIP.DOWN ZIP.RIGHT** move the top zipper to the subtree **(12 18 INTEGER.+)**. **ZIP.SSWAP** then swaps about the subtrees pointed to by the top two zippers. The root of the resulting item on top of the zipper stack is **((ZIP.DOWN ZIP.RIGHT ZIP.DOWN ZIP.RIGHT) (12 (12 18 INTEGER.+) INTEGER.+) ZIP.SWAP ZIP.DOWN ZIP.RIGHT ZIP.SSWAP))**, which evaluates to 42. This program would then be treated as the candidate child program and tested for acceptance criteria, such as appropriateness of size and difference in fitness relative to its parent programs (see Section 8).

6. ORDER

The fitness of a program for Order, $r(f)$, is computed as

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

where s is the problem size and $M(z, f)$ is the number of elements preceding the first occurrence of z within program f in depth-first order. An example 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) (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).

7. MAJORITY

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

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

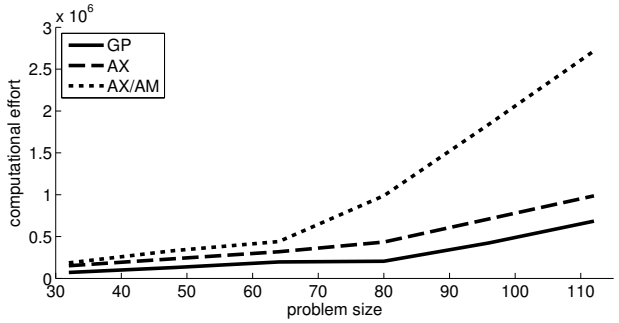
where s is the problem size and $N(z, f)$ is the number of instances of integer z in program f. The example program presented in the previous section on Order has a Majority fitness of 8.

8. EXPERIMENTS

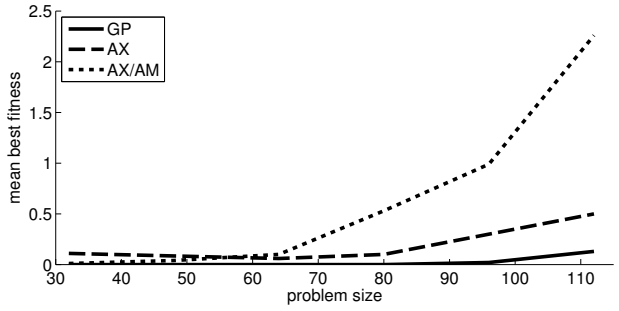
The system used for the following experiments is based upon the PushGP evolutionary algorithm (EA) with a number of modifications to the variation algorithm. The PushGP EA is similar to most EAs that are used in GP; for a detailed explanation of EAs in GP see [22, 3, 19]. Most of the differences between PushGP and GP relate to representation-specific operations, such as node selection and code generation. A detailed explanation of the PushGP EA is presented in [26]. We briefly explain our modified variation algorithm.

First, the parent population is copied into the child population. For all individuals in this child population a candidate replacement program is produced with the following algorithm: a uniform random number between $[0, 1]$ is generated and used to choose the variation mechanism according to the parameters of the respective experiment. Parents are selected with tournament selection from the parent population. The tournament size is 7 in all cases. The variation mechanism is applied to the parents to create a candidate program. The candidate program is accepted if: its size is less than the size limit, all elements of its error vector are less than or equal to those of its parents, and its code is not exactly the same as its parents' code. If the candidate program is accepted, then it replaces an individual in the child population. If the candidate program is rejected, then the individual in the child population, which was copied from the parent population, remains.

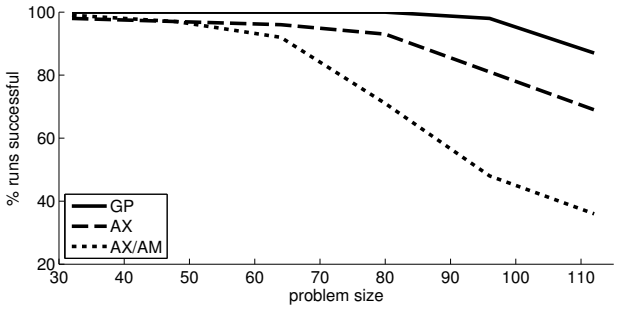
In these scaling experiments we evaluate problem sizes: 32, 48, 64, 80, 96, and 112. Each problem size is evaluated



(a) Problem size v. computational effort. Lower is better.

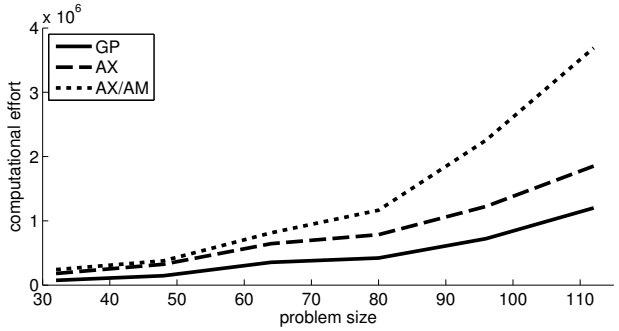


(b) Problem size v. mean best fitness. Lower is better.

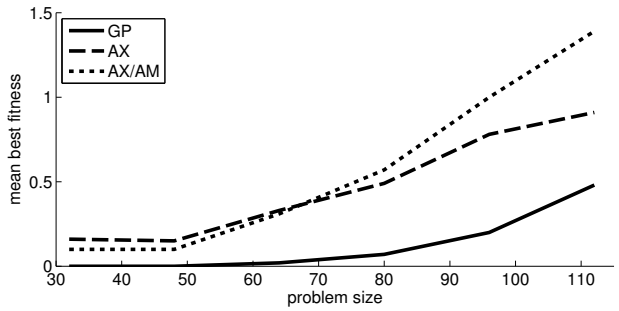


(c) Problem size v. % runs successful. Higher is better.

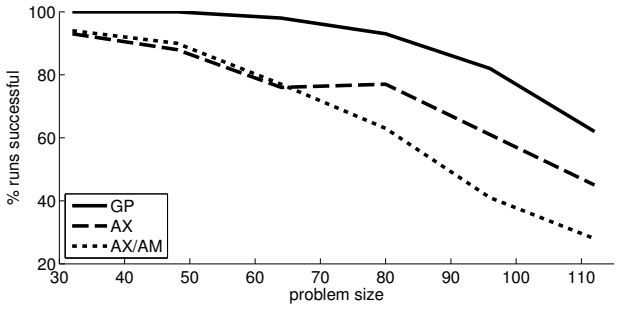
Figure 2: Results on the Order problem.



(a) Problem size v. computational effort. Lower is better.



(b) Problem size v. mean best fitness. Lower is better.



(c) Problem size v. % runs successful. Higher is better.

Figure 3: Results on the Majority problem.

for 100 runs with a population size of 500 and a generation limit of 501. The *limit* of maximum number of points in a program is 10 times the problem size. During the creation of a new program, a desired program size is randomly selected from the range $[0, limit]$ and a program of this size is generated according to the algorithm specified in [28]. Computational effort (CE) is computed as the number of individuals expected to be evaluated for a 99% chance of success [17]. Mean best fitness (MBF) is computed as the average fitness of the best individual at the end of all runs for a given set of parameters. For all parameters 10% replication is used. In these runs **GP** uses 45% crossover and 45% mutation; **AX** uses 90% autoconstructive crossover; and **AX/AM** uses 45% autoconstructive crossover and 45% autoconstructive mutation. Note that the criteria for acceptance/rejection of candidate programs are applied to runs for all parameter sets.

9. RESULTS

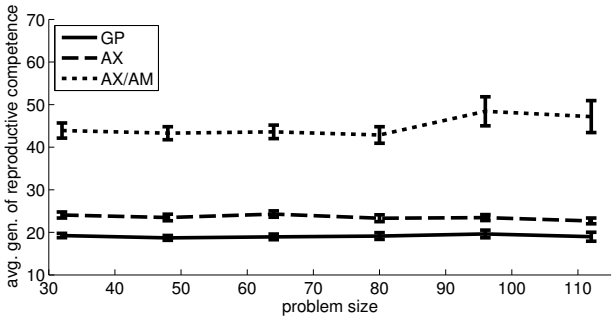
The results for the Order problem are presented in Figure 2. For all scales of Order the ranking of performance is: **GP**, **AX**, **AX/AM**. In terms of the CE, Figure 2a, **AX/AM** degrades in performance significantly as problem size increases, yet **AX** stays nearly parallel to **GP**. For MBF, Figure 2b, **AX/AM** is again shown to perform worse as problem scale increases while both **AX** and **GP** have much less degredation in performance with respect to problem scale. This is also true in terms of the % of successful runs, Figure 2c. As a whole we can see that **AM** seems to decrease performance on larger instances of the Order problem, and **AX** on its own has comparable scaling properties to **GP**. The upward displacement of **AX** relative to **GP** is expected given that autoconstructive individuals are required to evolve both a reproductive mechanism and a problem solution.

The results for the Majority problem are presented in Figure 3. While the ranking of performance is the same as the Order problem for larger scales, we can see that for problem sizes smaller than 64, **AX/AM** outperforms **AX** in terms of MBF and % success. In terms of CE we also see a similar reduction of difference between **AX** and **GP** to the Order results, but to a lesser extent. This suggests that the unary zipper instructions (zipper movements and **ZIP.RAND**) are superior to recombinatory zipper operators for small Majority problems. This is expected due to

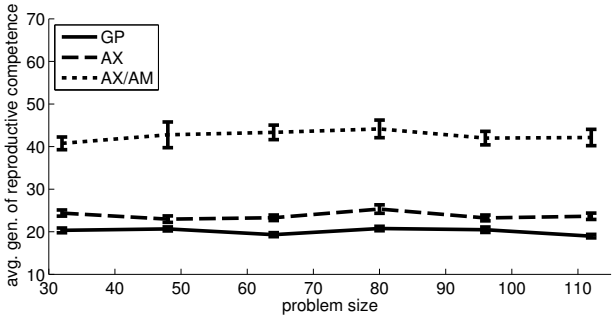
the formulation of the Majority problem. For small problem sizes randomly generated code has a high probability of containing each possible instruction. Given that Majority is a model of the accumulation of beneficial components, regardless of ordering, we can expect this increased probability of inserting beneficial components favors reproductive strategies that involve the insertion of random code.

Figures 4a and 4b show the time until reproductive competence is reached. Reproductive competence is defined as the first generation in which at least 10% of the candidate programs produced during the variation phase of the EA are accepted. It is clear that for both problems **GP**, on average, reaches reproductive competence before either **AX** or **AX/AM**. The number of generations until reproductive competence for **GP** is achieved may seem long to some. This delay is a result of the acceptance criteria that are employed by the EA. The difference between the time to achieve reproductive competence for **GP** and **AX** is relatively small, which indicates that zipper-based autoconstruction is capable of achieving reproductive competence rapidly. On the other hand, the time until **AX/AM** reaches reproductive competence is essentially double that of **AX**. Since the probabilities of **AX** v. **AM** in the **AX/AM** configuration are equal, we suggest that **AM** may not be an appropriate variation operator for either the Order or Majority problems. On the other hand, **AX** becomes reproductively competent within approximately 5 generations of **GP** for both problems. Examinations of the change in the acceptance rate of candidate programs over evolutionary time (not presented) show similar rates for **AX** and **GP** while the poor performance of **AX/AM** obscures the comparison.

In Figures 5a and 5b, we show a histogram of the % of successful runs for the Order and Majority problems as the problem size increases. The trends are similar in both figures; **GP** consistently outperforms both autoconstructive operators for the 3 smallest problem sizes for both Order and Majority. For larger problem sizes **GP** also does better, yet a modest number of runs of **AX** converge at early generations. Although there are insufficient early convergences of **AX** to claim significance, they do hint at the promise of autoconstructive evolution as a hyper-heuristic for the co-evolution of reproductive heuristics and problem solutions.



(a) Scaling of reproductive competence on Order.



(b) Scaling of reproductive competence on Majority.

Figure 4: Average number of generations until 10% of candidate programs are accepted for both the Order and Majority problems. Error bars indicate standard error. Fewer generations is better.

10. DISCUSSION

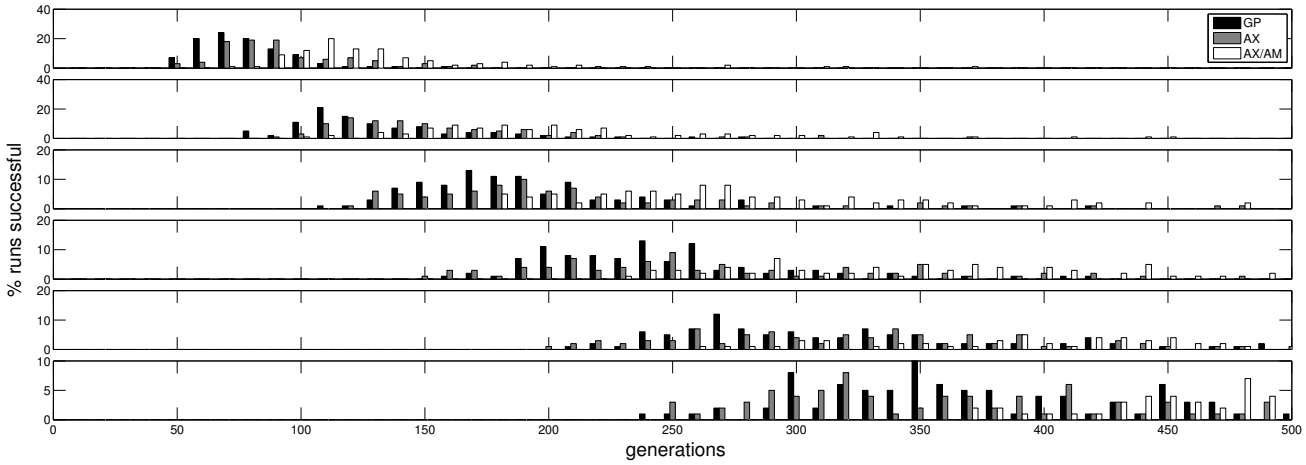
Our results show a number of interesting properties for autoconstructive evolution which have not been documented previously, such as the benefit of recombinatory variation, favorable scaling in computational effort with respect to problem size, and rapid evolution of reproductive competence. However, there is no indication that autoconstructive evolution would outperform GP for larger problem sizes. There are a number of constraints that have been imposed in this minimalist implementation of autoconstruction which may be limiting the potential of autoconstructive evolution.

A minimal autoconstructive instruction set was used. While the zipper-based instruction set is quite powerful, it may be enhanced with additional instructions. For example, conditional zipper instructions could allow for the evolution of size-based variation operators or pattern-matching in subtrees. In previous studies of autoconstructive evolution very large instruction sets were used (often almost all instructions in the Push language). These studies led to the evolution of interesting reproductive mechanisms; however, the performance of autoconstruction in these cases was significantly worse than what one would expect from traditional GP. A study that begins with a minimal instruction set, such as our zipper-based instruction set, and incrementally adds additional instructions may prove to be fruitful.

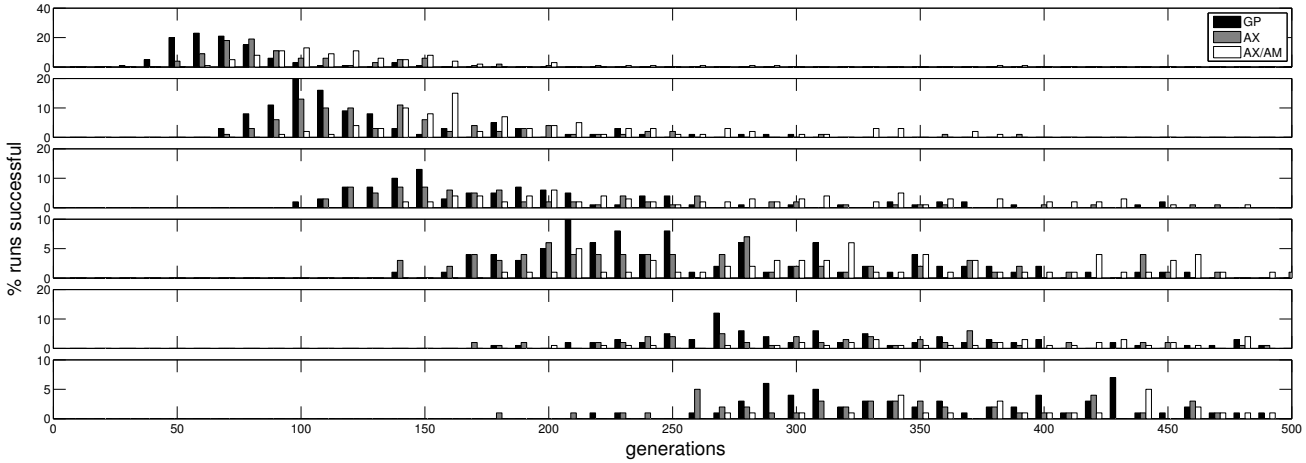
Stringent acceptance criteria were imposed. The acceptance criteria for candidate programs may not be optimal. While experience and guesswork brought us to impose both the "no cloning" rule and neutral-or-better fitness improvement criteria, it is not clear whether these criteria are ideal, sufficient, or limiting. The use of hereditary analysis, such as measures of conservation in reproductive mechanism, may allow for greater flexibility in the reproductive mechanisms that can evolve.

A clear issue that arose in this study is **autoconstruction takes longer to get started.** While the time until reproductive competence emerges under autoconstructive evolution is fast when compared to previous work on autoconstruction, it is still slower than GP. This suggests that autoconstructive evolution may simply require more evolutionary time. The reproductive competence was measured as the first generation where 10% of the candidate programs are accepted for the next generation. Although we also inspected the change in acceptance rates over evolutionary time, a rigorous study of the causes of candidate program rejection and how it might be prevented could improve the efficiency of autoconstructive variation.

A more subtle issue that can be seen in Figures 2c, 3c, and 5 is the apparent bimodal behavior of autoconstruction where one mode represents runs that succeed and the other mode represents runs that are likely to fail. An example of this can be clearly seen by comparing Figure 2c and



(a) Generations of convergence for Order.



(b) Generations of convergence for Majority.

Figure 5: Histogram of % runs converged at a given generation for both the Order and Majority problems. A higher count on the left means faster convergence.

the histogram of generation of convergence in Figure 5a for problem size 32, where the majority of **AX** runs converge before generation 200 yet more than 5% of runs do not converge before the generation limit is reached. This behavior suggests that autoconstructive evolution may benefit from terminating some runs prematurely; if runs that are likely to fail can be detected. One approach to detecting such runs is through the use of order statistics. A preliminary study on the use of order statistics in GP suggests that they may be a useful tool for measuring how effectively a given system explores a program space [30].

11. CONCLUSION

In this study we present two methods for performing autoconstructive evolution, and have shown that autoconstructive evolution can exhibit favorable scaling properties with respect to problem size on structural problems. Furthermore, as a hyper-heuristic autoconstruction is a particularly novel methodology because it functions as a hyper-heuristic at two levels. Evolved reproduction heuristics are both applied to problem solutions and to themselves. We have shown that autoconstructive crossover has a similar scaling of performance to genetic programming in the Push language. This is likely due to similar costs of evolving a reproductive mechanism for any problem size. The similar scaling of performance for autoconstructive crossover and GP lends credence to the hypothesis that, in the long-term, it may be the coevolution of reproductive mechanism and problem solution that leads to accelerated gains which outperform hand-designed algorithms.

Acknowledgements

We thank Emma Tosch, James McDermott, the DEMO lab, the Hampshire College CI lab, and our anonymous reviewers. 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. Computing support was partially provided by the Brandeis HPC. Una-May O'Reilly acknowledges the support of General Electric and the Li Ka-Shing Foundation.

12. REFERENCES

[1] P. Angeline. Two self-adaptive crossover operations for genetic programming. In *Advances in genetic programming 2*, pages 89–110, 1995.

[2] T. Back. Self-adaptation in genetic algorithms. In *Towards a Practice of Autonomous Systems: Proc. of the First European Conf. on Artificial Life*, pages 263–271, 1992.

[3] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA, Jan. 1998.

[4] E. Burke, G. Kendall, and J. Newall. Hyper-heuristics: An emerging direction in modern search technology. In *Handbook of Metaheuristics*, pages 457–474. 2003.

[5] E. K. Burke, M. R. Hyde, and G. Kendall. Evolving Bin Packing Heuristics with Genetic Programming. In *Parallel Problem Solving from Nature - PPSN IX*, volume 4193 of *LNCS*, pages 860–869, 2006.

[6] E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and J. R. Woodward. Exploring Hyper-heuristic Methodologies with Genetic Programming. In C. L. Mumford and L. C. Jain, editors, *Computational Intelligence*, volume 1 of *Intelligent Systems Reference Library*, chapter 6, pages 177–201. 2009.

[7] G. Durrett, F. Neumann, and U.-M. O'Reilly. Computational Complexity Analysis of Simple Genetic Programming On Two Problems Modeling Isolated Program Semantics. In *Foundations of Genetic Algorithms*, pages 69–80, 2011.

[8] B. Edmonds. Meta-genetic programming: Co-evolving the operators of variation. *Turk J. Elec. Engin.*

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

[10] A. Fialho, L. D. Costa, M. Schoenauer, and M. Sebag. Extreme value based adaptive operator selection. In *Parallel Problem Solving from Nature–PPSN X*, pages 175–184, 2008.

[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 *Proc. of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 16–36, 1998.

[12] S. Gustafson, E. Burke, and N. Krasnogor. The tree-string problem: an artificial domain for structure and content search. *Genetic Programming*, pages 215–226, 2005.

[13] K. Harrington, E. Tosch, L. Spector, and J. Pollack. Compositional Autoconstructive Dynamics. In *Proc. of the 8th Intl. Conf. on Complex Systems*, 2011.

[14] K. l. Harrington and J. B. Pollack. Zipper-based Meta-Genetic Programming. Technical report, Department of Computer Science, Brandeis University, Waltham, MA, 2011.

[15] G. Huet and I. France. Functional pearl: The zipper. In *J. Functional Programming*, pages 549–554, 1997.

[16] W. Kantschik, P. Dittrich, M. Brameier, and W. Banzhaf. Meta-evolution in graph GP. *Genetic Programming*, pages 652–652, 1999.

[17] J. Koza. *Genetic programming: on the programming of computers by means of natural selection*. 1992.

[18] J. Koza. Spontaneous emergence of self-replicating and evolutionarily self-improving computer programs. In *Artificial life III*, volume 17, pages 225–262, 1994.

[19] W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[20] J. Niehaus and W. Banzhaf. Adaption of Operator Probabilities in Genetic Programming. In *Genetic Programming, Proc. of EuroGP'2001*, volume 2038 of *LNCS*, pages 325–336, 2001.

[21] M. Oltean and D. Dumitrescu. Evolving {TSP} Heuristics Using Multi Expression Programming. In *Computational Science - ICCS 2004: 4th Intl. Conf., Part II*, volume 3037 of *Lecture Notes in Computer Science*, pages 670–673, 2004.

[22] R. Poli, W. Langdon, and N. McPhee. A field guide to genetic programming. 2008.

[23] G. Polya. *How To Solve It: A New Aspect of Mathematical Method*. Princeton University Press, Princeton, NJ, 1945.

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

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

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

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

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

[29] A. Teller. Evolving programmers: The co-evolution of intelligent recombination operators. In *Advances in genetic programming 2*, pages 45–68, 1996.

[30] E. Tosch and L. Spector. Achieving COSMOS: A metric for determining when to give up and when to reach for the stars. In *GECCO-UP workshop, Workshop Proc. of GECCO-2012*, 2012.

[31] A. Tuson and P. Ross. Adapting operator settings in genetic algorithms. *Evolutionary computation*, 6(2):161–184, 1998.

[32] L. Vanneschi, M. Castelli, and L. Manzoni. The {K} landscapes: a tunably difficult benchmark for genetic programming. In *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1467–1474, 2011.