Evolution Evolves with Autoconstruction

6th Workshop on Evolutionary Computation for the Automated Design of Algorithms
Genetic and Evolutionary Computation Conference (GECCO)
Denver, Colorado, USA, July, 2016

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

Outline

- What is autoconstructive evolution?
- Prior work
- Recent developments
- Results
- Prospects

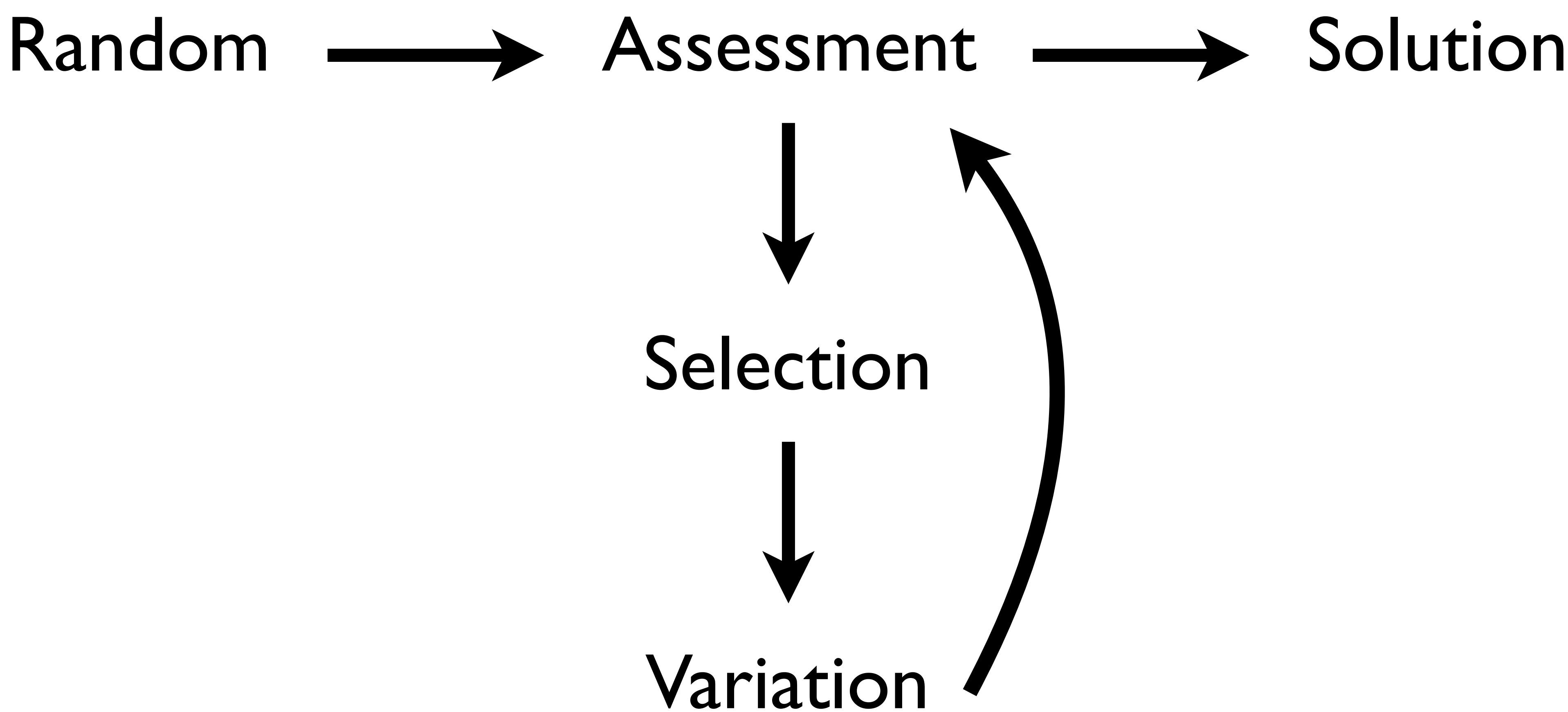
Autoconstructive Evolution (1)

- Evolve evolution while evolving solutions
- How? Individuals produce and vary their own children, with methods that are subject to variation
- Requires understanding the evolution of variation
- Hope: May produce EC systems more powerful than we can write by hand

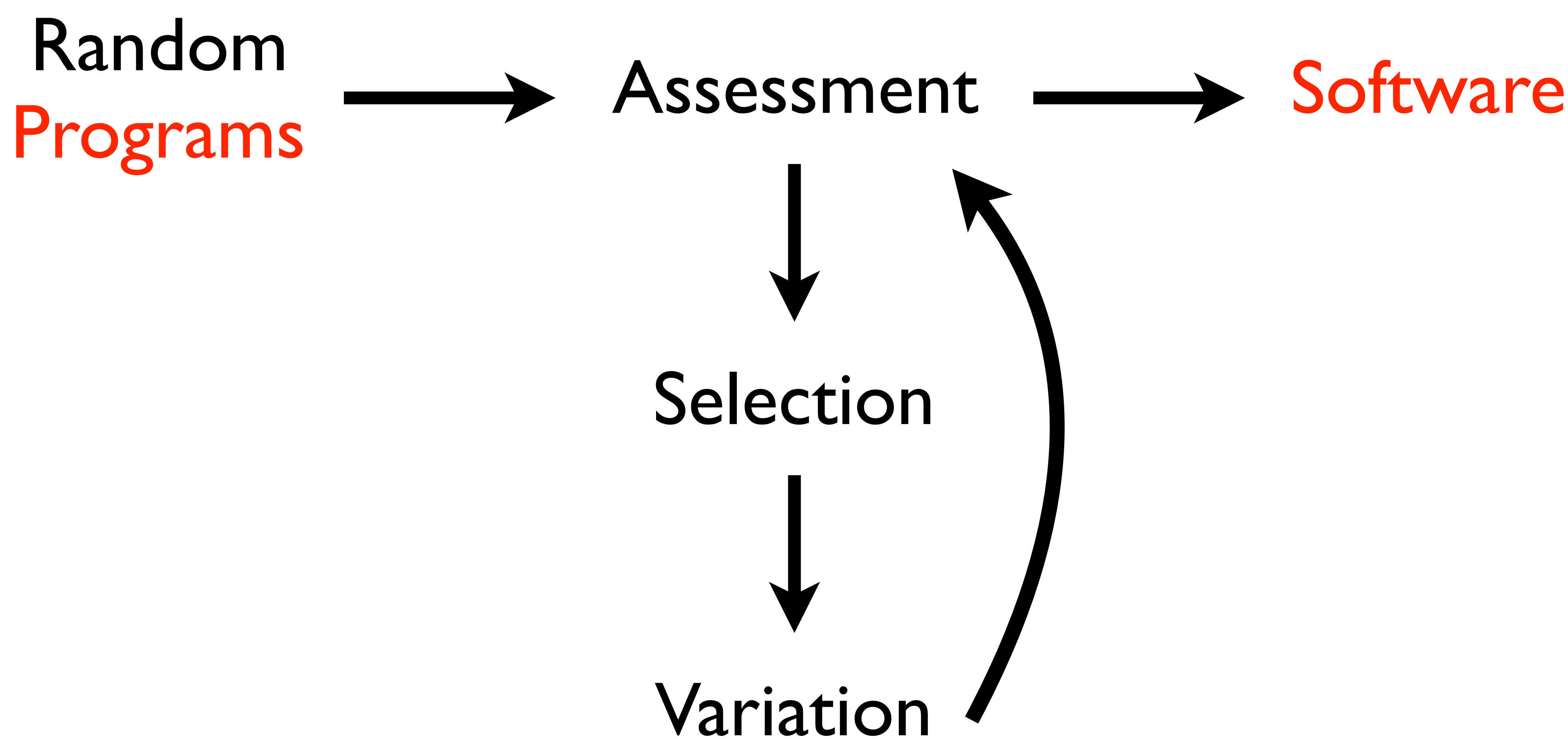
Autoconstructive Evolution (2)

- A 15 year old project (building on older and broader-based ideas)
- Like genetic programming, but harder and less successful!
- Recent: AutoDoG, sometimes solve significant problems, intriguing patterns of evolving evolution

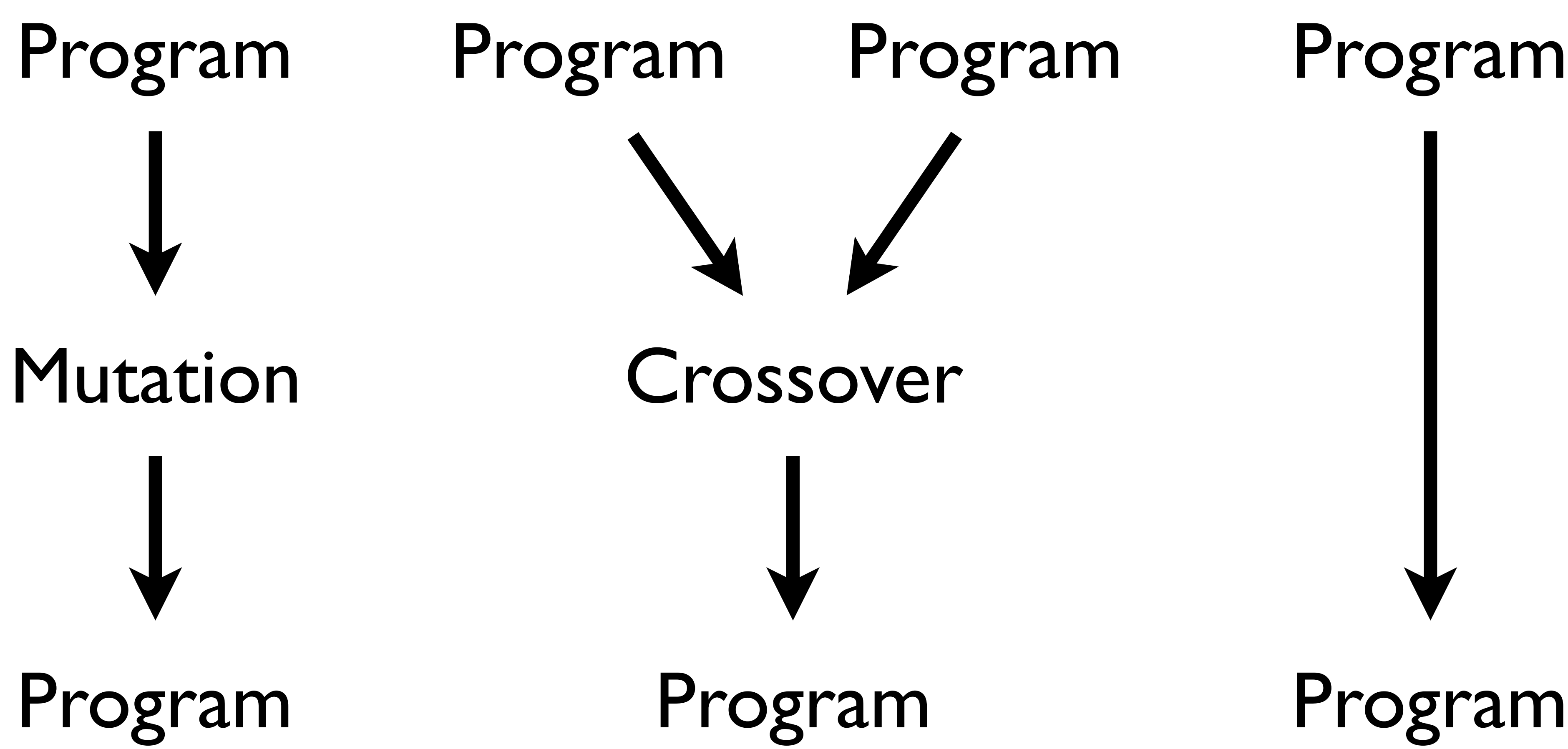
Evolutionary Computing



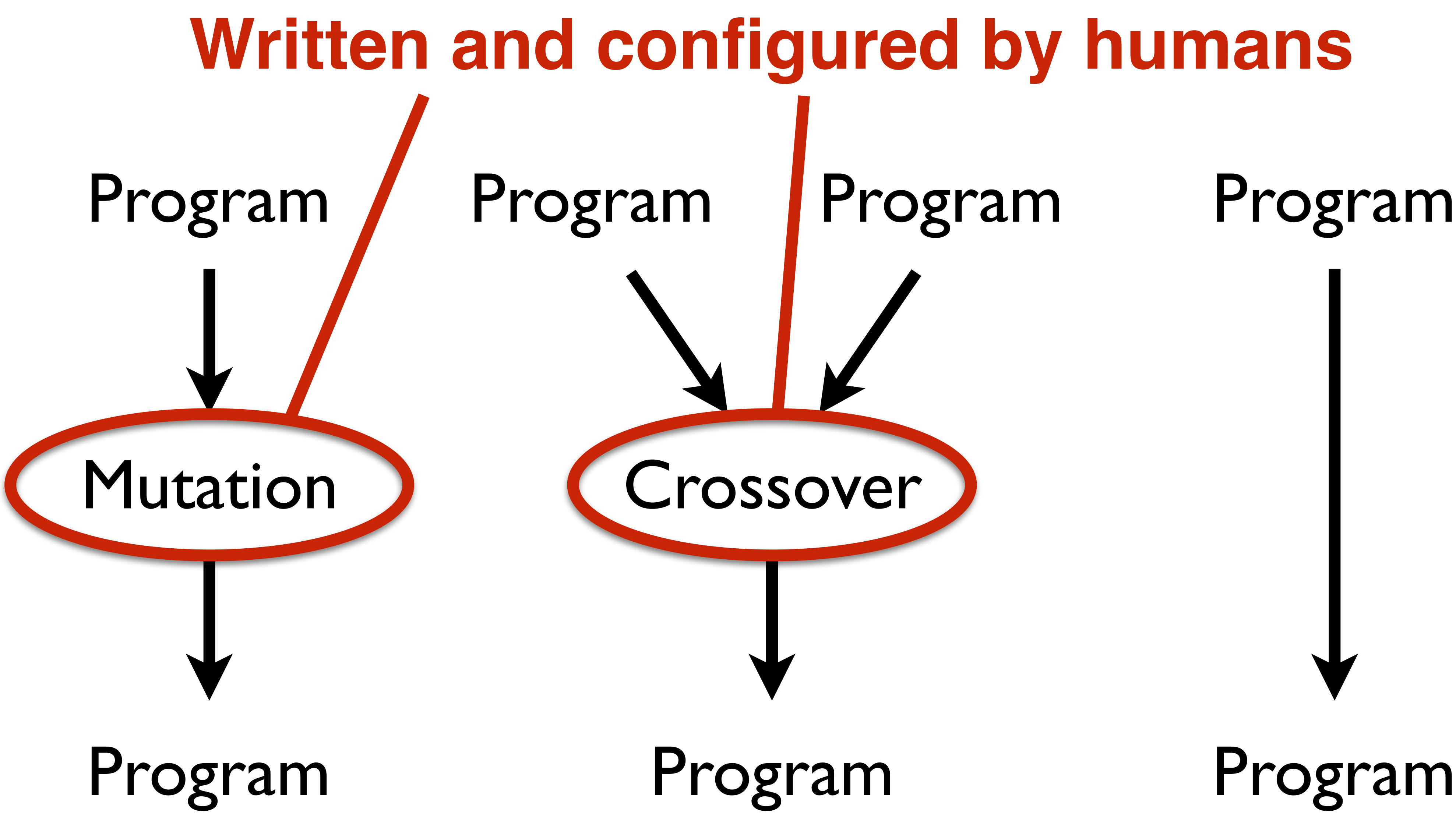
Genetic Programming



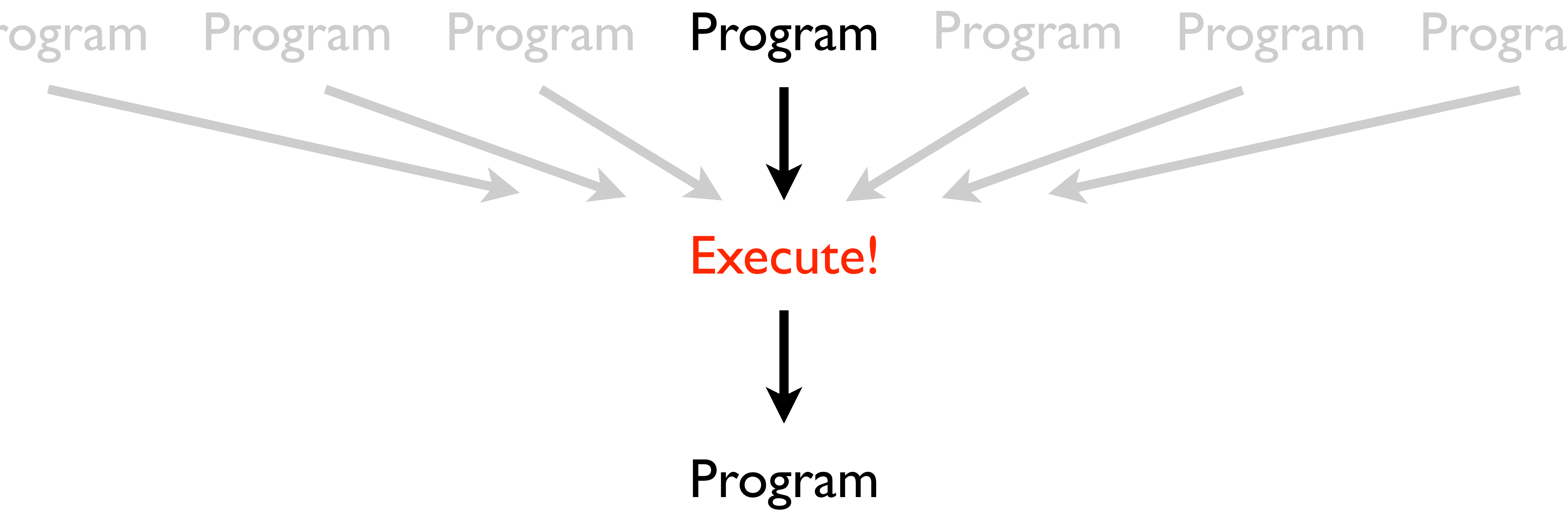
Variation in Genetic Programming



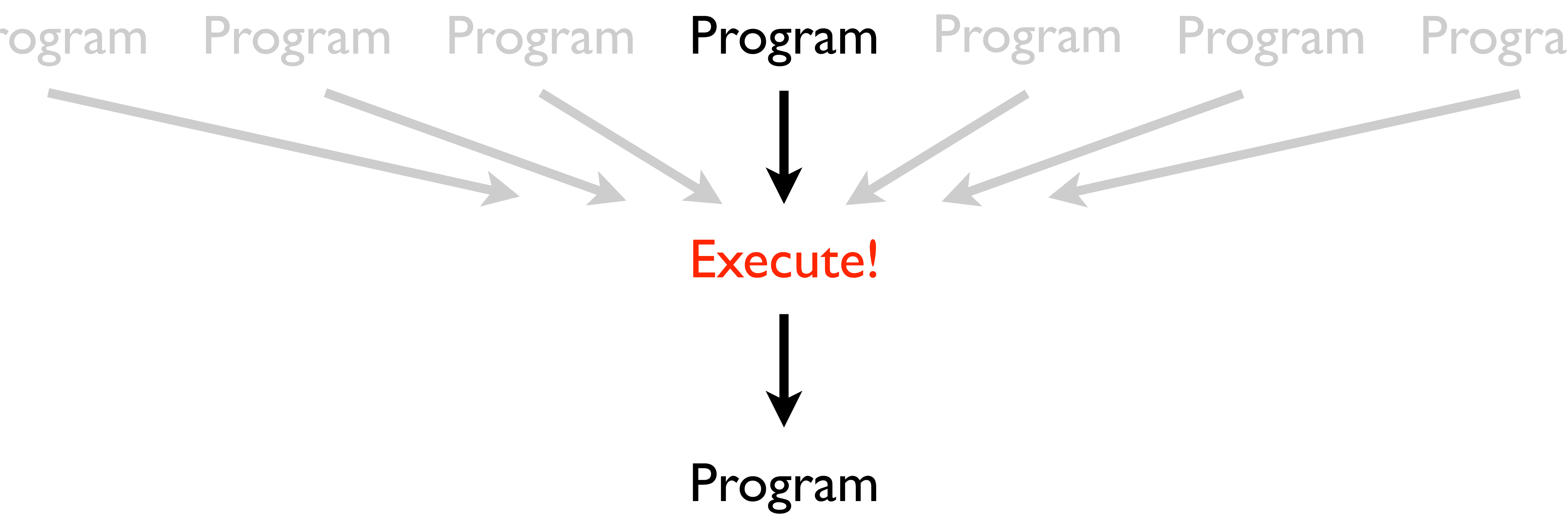
Variation in Genetic Programming



Autoconstruction



Autoconstruction



A bit more complicated when genomes distinguished from programs

Autoconstructive Evolution (3)

- Individual programs make their own children
- In doing so, they control their own mutation and recombination rates and methods, and in some cases mate selection, etc.
- The machinery of reproduction and diversification (i.e., the machinery of evolution) evolves
- In **Push**, experimentation with autoconstructive evolution is easy and natural

Hazards

- Clones
- Deadenders
- Error catastrophes

Push (1)

- A programming language
- Designed for programs that evolve
- Data flows via stacks, not syntax
- One stack per type:
  integer, float, boolean, string, code, exec, vector, ...

Push (2)

- `program → instruction | literal | ( program* )`
- Turing complete, rich data and control structures
- PushGP: GP system that evolves Push programs
- Makes it easy to combine manipulation of programs/genomes with other computations

Plush

Instruction	integer_eq	exec_dup	char_swap	integer_add	exec_if	
Close?	2	0	0	0	1	
Silence?	1	0	0	1	0	

- Linear genomes for Push programs
- Facilitates useful placement of code blocks
- Permits uniform linear genetic operators
- Allows for epigenetic hill-climbing

Push (3)

- Implementations in C++, Clojure, Common Lisp, Java, Javascript, Python, Racket, Ruby, Scala, Scheme, Swift

- The work described here uses Clojush, in Clojure

- http://pushlanguage.org

Pushpop (2001)

- Construct eggs while solving problems
- Tournaments for hatching rights
- Clones are stillborn
- Programs can access and run code of the entire population, both while solving problems and while building babies (which they do at the same time)

SwarmEvolve 2.0 (2005)

- Behavior (including reproduction) in a 3D virtual world controlled by evolved programs
- Autoconstruction followed by imposed but program-controlled mutation
- Color sensing, energy conservation, communication, energy sharing

AutoPush (2010)

- Goals:
  - Superior problem-solving performance (maybe)
  - Tractable analysis (yes)
- Asexual
- Reproductive vs. problem-solving phases
- Constraints on selection and birth

Example Constraints

- Prefer parents with non-stagnant lineages (changed performance in the most recent half of the lineage, after some threshold lineage length)
- Prevent birth from lineages with constant differences from each generation to the next

AX/AM (2012)

- Kyle Harrington, Una-May O'Reilly, and Jordan Pollack
- Zipper data structures for constructing children
- No cloning + neutral-or-better error
- Synthetic problems involving program structure

Results of Prior Work

- Demonstrated that selection can promote diversity
- Exhibited dynamics of diversification and adaptation
- Weak problem-solving power
- Difficult to analyze results, compare to ordinary genetic programming, or generalize

AutoDoG (2016)

- **Auto**constructive **D**iversification **o**f **G**enomes
  - Construct genomes, not programs
  - Distinct mode/phase for construction of offspring
  - Select combinatorially, not on aggregate error
  - Enforce diversification constraints

What is Constructed?

- In prior work: Push programs, manipulated on code stacks using Lisp-inspired code-manipulation instructions
- In AutoDoG: Plush genomes, which are linear sequences of genes that specify instructions along with epigenetic markers that determine structure when Plush genomes are translated into Push programs, prior to running them

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

When/how is it Constructed?

- In prior work: Various; sometimes during error testing, sometimes with problem inputs, sometimes with imposed but controllable variation
- In AutoDoG: Only within the autoconstruction genetic operator, entirely by the program itself
  - Construction: inputs are no-ops
  - Error testing: rand instructions produce constants

Who Constructs?

- In prior work: Parents selected using standard, error aggregating methods (tournament selection)
- In AutoDoG: Lexicase selection

Lexicase Selection

To select single parent:

1. Shuffle test cases
2. First test case – keep best individuals
3. Repeat with next test case, etc.

Until one individual remains

The selected parent may be a specialist, and may or may not be particularly good on average, even though it may contribute to the evolution of generalists later

IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, VOL. XX, NO. XX, XXXX 2014 1

Solving Uncompromising Problems with Lexicase Selection

Thomas Helmuth, Lee Spector *Member, IEEE,* James Matheson

***Abstract*—We describe a broad class of problems, called "uncompromising problems," characterized by the requirement that solutions must perform optimally on each of many test cases. Many of the problems that have long motivated genetic programming research, including the automation of many traditional programming tasks, are uncompromising. We describe and analyze the recently proposed "lexicase" parent selection algorition and show that it can facilitate the solution of uncompromising problems by genetic programming. Unlike most traditional parent selection techniques, lexicase selection does not base selection on a fitness value that is aggregated over all test cases; rather, it considers test cases one at a time in random order. We present results comparing lexicase selection to more traditional parent selection methods, including standard tournament selection and implicit fitness sharing, on four uncompromising problems: finding terms in finite algebras, designing digital multipliers, counting words in files, and performing symbolic regression of the factorial function. We provide evidence that lexicase selection maintains higher levels of population diversity than other selection methods, which may partially explain its utility as a parent selection algorithm in the context of uncompromising problems.**

***Index Terms*—parent selection, lexicase selection, tournament selection, genetic programming, PushGP.**

I. Introduction

GENETIC programming problems generally involve test cases that are used to determine the performance of programs during evolution. While some classic genetic programming problems, such as the artificial ant problem and the

example, we can imagine
simulated wind turbine in
performance in low wind
performance in high wind c
optimize performance on al
and some sort of compro
Many common parent sel
ment selection, introduce c
aggregating the performan
cases into a single fitness
may be as simple as sum
squares, into a single error
as implicit fitness sharing
based on population statist

By contrast, we wish to
mising" problems: problem
must perform as well on
perform on that test case; t
is a problem for which i
to perform sub-optimally
for good performance on
problem defined by the se
programs in the search s
produced by program p_j
lower error being better. T
program $p \in P$ would be c
if and only if $p(t_i) \leq p_j(t$

Previous Results

Problem	Lexicase	Tourney
Count Odds	8	0
Double Letters	6	0
Mirror Image	78	46
Negative To Zero	45	10
Replace Space with Newline	51	8
String Lengths Backwards	66	7
Syllables	18	1
Vector Average	16	14
X-Word Lines	8	0

- 9 of 29 program synthesis benchmark problems
- Also higher levels of behavioral diversity

See Bill La Cava's presentation on Epsilon Lexicase Selection, and Tom Helmuth's presentation in the GP best paper track, both on Saturday afternoon

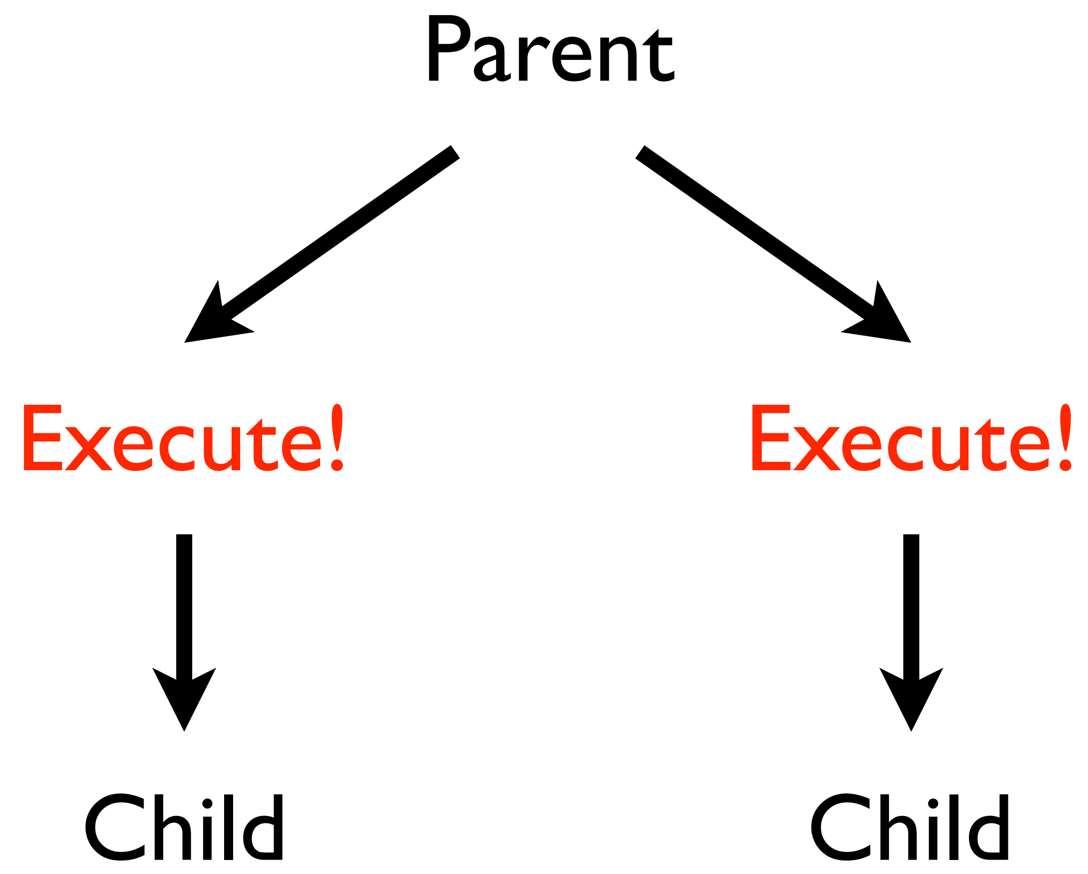
Who Survives?

- In prior work: Sometimes everyone except clones, sometimes only those satisfying constraints on progress within lineages
- In AutoDoG: Only those satisfying diversification constraints on reproductive behavior, determined from a cascade of temporary descendants

Diversification Constraint

- Applied to a cascade of temporary descendants
- Used here:
  - Children must differ from parent, differently
  - Applied to programs expressed by genomes
  - Enforced on a cascade with two children

Diversification Constraint



Parent/child **program** differences positive; not the same

Diversification Constraint

- Still under development
- How can you tell if an individual has the potential to produce diverse, adaptive descendants?
- Considering larger cascades, variation of:
    - genomes
    - reproductive behavior
    - problem solving behavior

Needed for Evolution to Evolve

- Diversity: Individuals vary
- Diversification: Individuals produce descendants that vary, in various ways (used here)
- Recursive Variance: Individuals produce descendants that vary in the ways that they vary their offspring (under development)

29 Software Synthesis Benchmarks

- Number IO, Small or Large, For Loop Index, Compare String Lengths, Double Letters, Collatz Numbers, Replace Space with Newline, String Differences, Even Squares, Wallis Pi, String Lengths Backwards, Last Index of Zero, Vector Average, Count Odds, Mirror Image, Super Anagrams, Sum of Squares, Vectors Summed, X-Word Lines, Pig Latin, Negative to Zero, Scrabble Score, Word Stats, Checksum, Digits, Grade, Median, Smallest, Syllables

- PushGP has solved all of these except for the ones in blue

- Presented in a GECCO-2015 GP track paper

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

Replace Space With Newline

- Multiple types, looping, multiple tasks
- Simplified solution:
```
(\space char_dup exec_dup in1 \newline string_replacechar
print_string string_removechar string_length)
```
- PushGP can achieve success rates up to ~95%
- AutoDoG as described here succeeds 5-10%

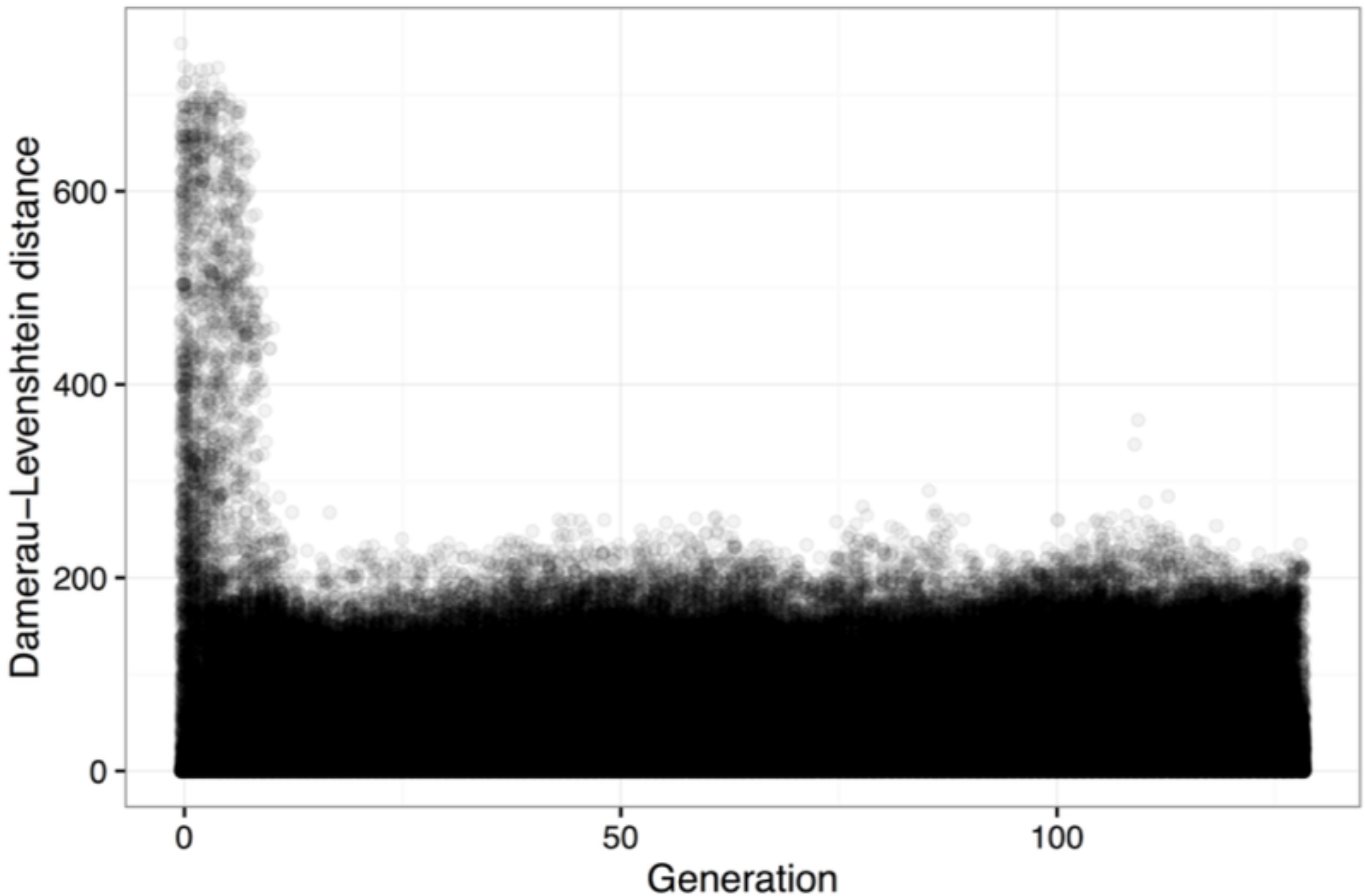


Figure 1: DL-distances between parent and child during a single non-autoconstructive run of GP on the Replace Space With Newline problem

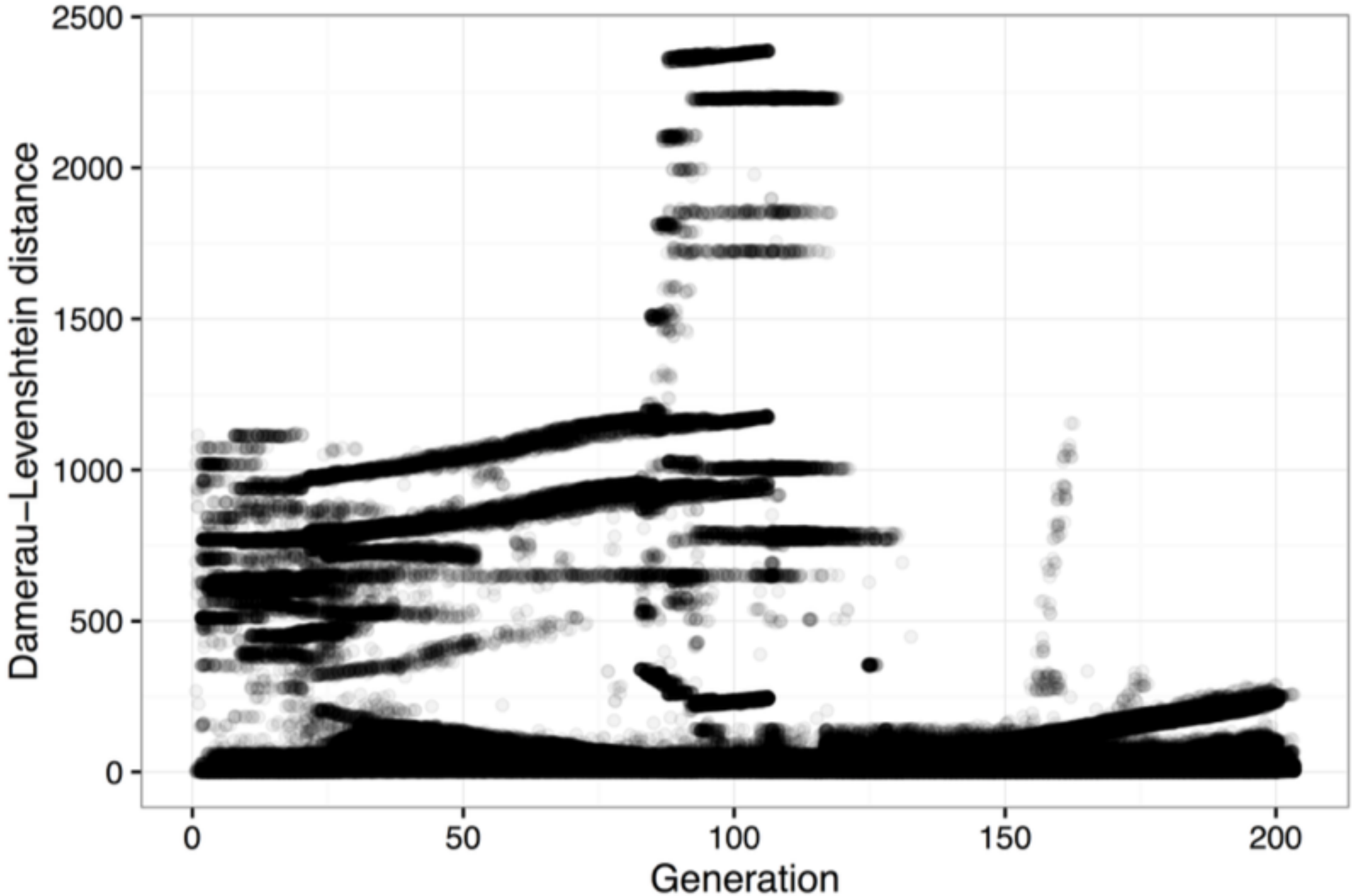


Figure 3: DL-distances between parent and child during a single autoconstructive run of GP on the Replace Space With Newline problem

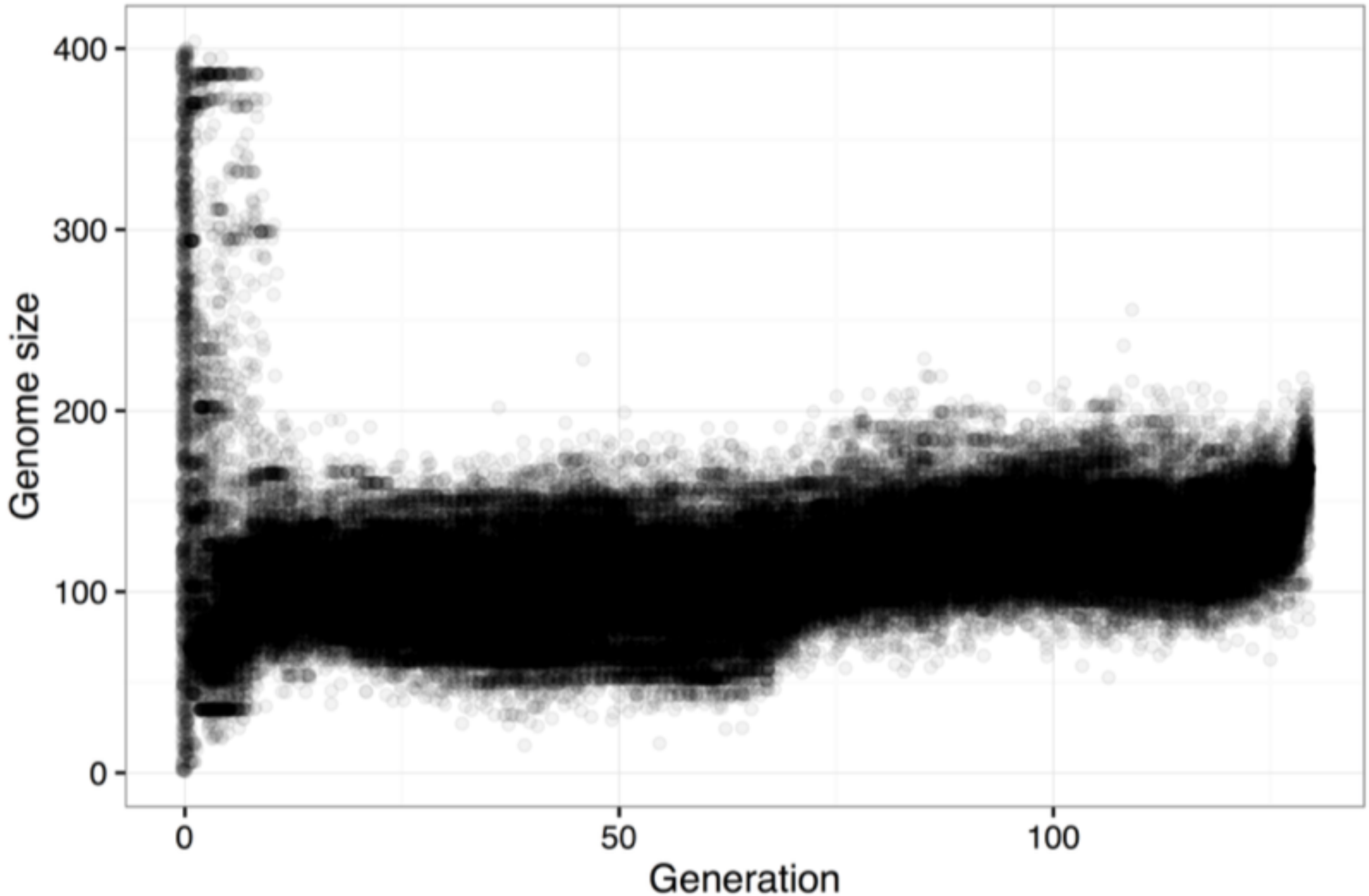


Figure 2: Genome sizes during a single non-autoconstructive run of GP on the Replace Space With Newline problem

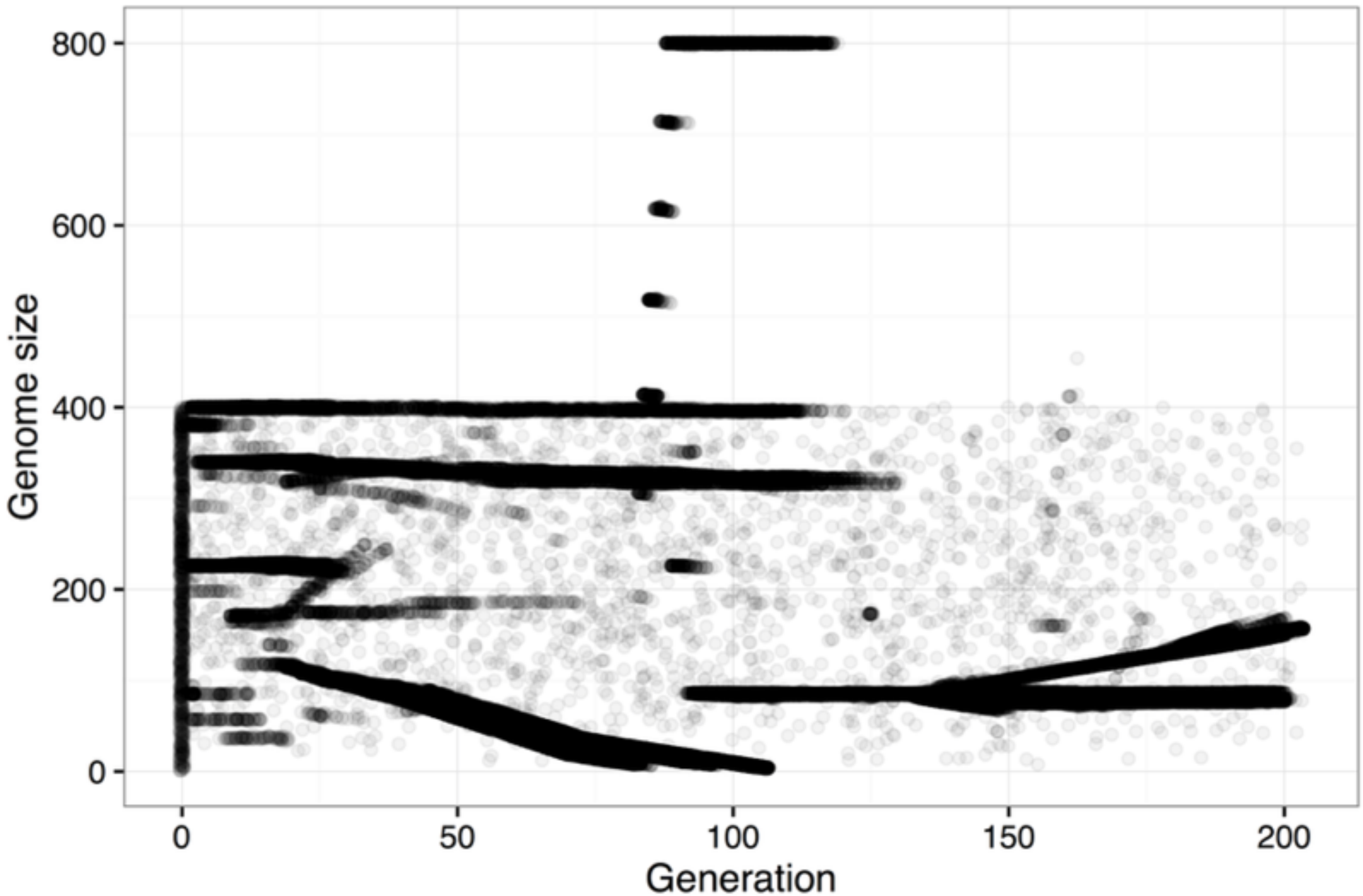
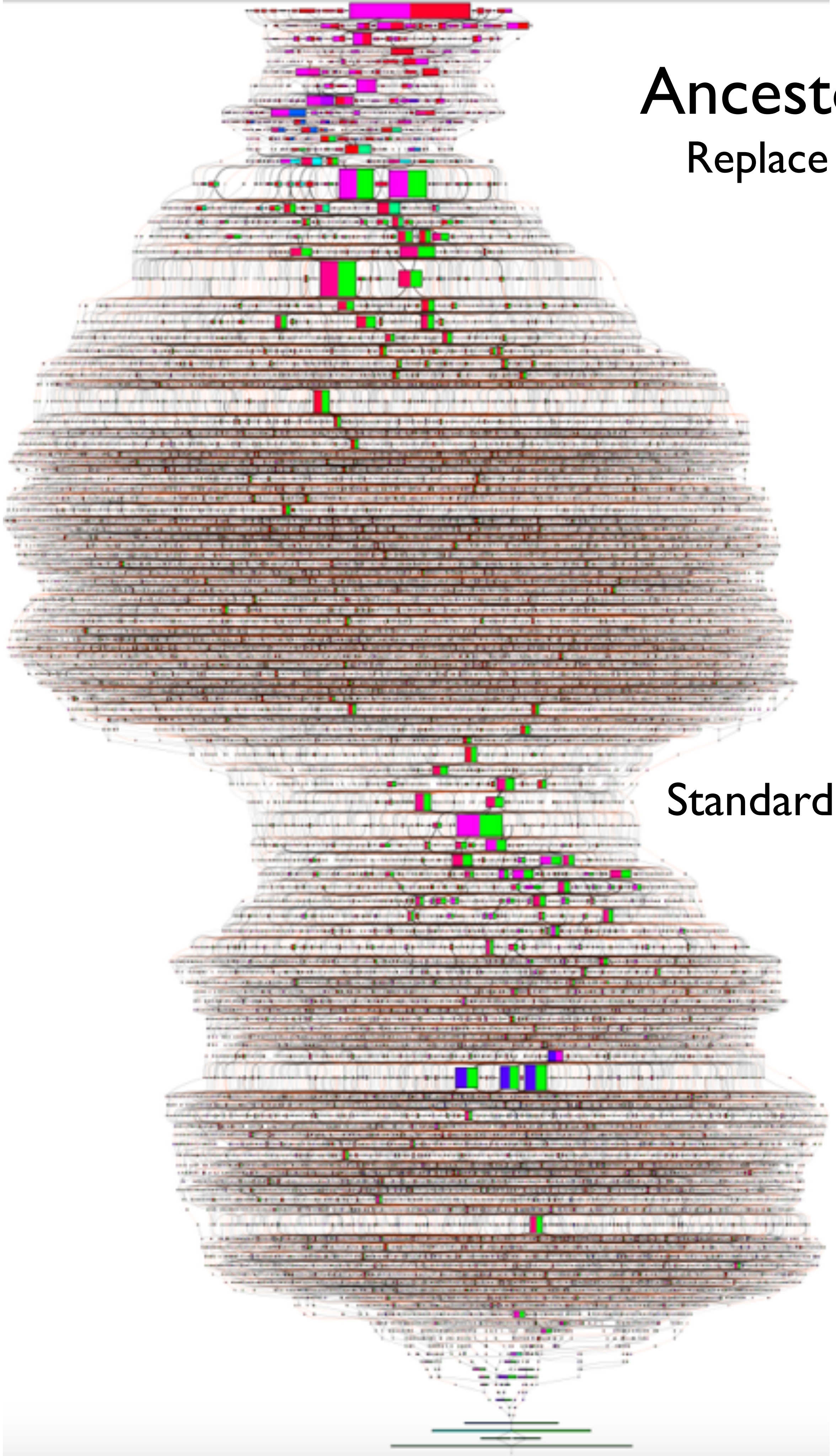


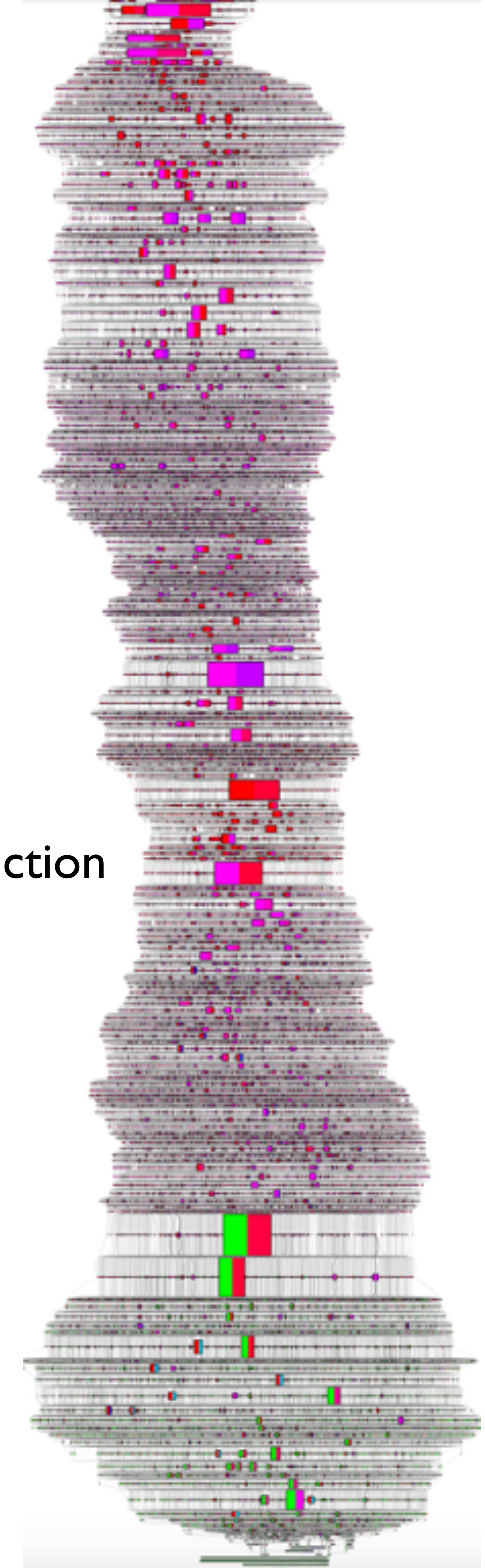
Figure 4: Genome sizes during a single autoconstructive run of GP on the Replace Space With Newline problem

Ancestors of Solutions

Replace Space with Newlines



Standard Operators



Autoconstruction

Conclusions and Prospects

- Autoconstructive evolution can now solve reasonably hard problems, at least some of the time
- So far, it takes longer, because it must evolve evolution along with solutions
- Can it solve problems that can't be solved by ordinary genetic programming? Possibly, because it evolves
- Studying how/why it works may help us to improve it
- Studying how/why it works may help us to understand the evolution of biological evolution

Thanks

- Members of the Hampshire College Computational Intelligence Lab for discussions
- Anonymous reviewers for both corrections and insightful comments
- Josiah Erikson for systems support
- Hampshire College for support for the Hampshire College Institute for Computational Intelligence
- This material is based upon work supported by the National Science Foundation under Grants No. 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.