Recent Developments in Autoconstructive Evolution

Lee Spector, Hampshire College & UMass Amherst
Eva Moscovici, UMass Amherst

Workshop on Evolutionary Computation for the Automated Design of Algorithms (ECADA)
Genetic and Evolutionary Computation Conference (GECCO)
Berlin, Germany; July, 2017

Outline

- Autoconstructive evolution
- AutoDoG (2016): 4 features and evolution evolves!
- 2 new milestones reached via 2.5 new features
- Future

Motivation

- In nature, the ways in which evolution works ***itself evolves***, through variation and selection of mechanisms for variation and selection
- In evolutionary computation, if the evolutionary process ***can itself evolve***, then it should be capable of solving more and more difficult problems

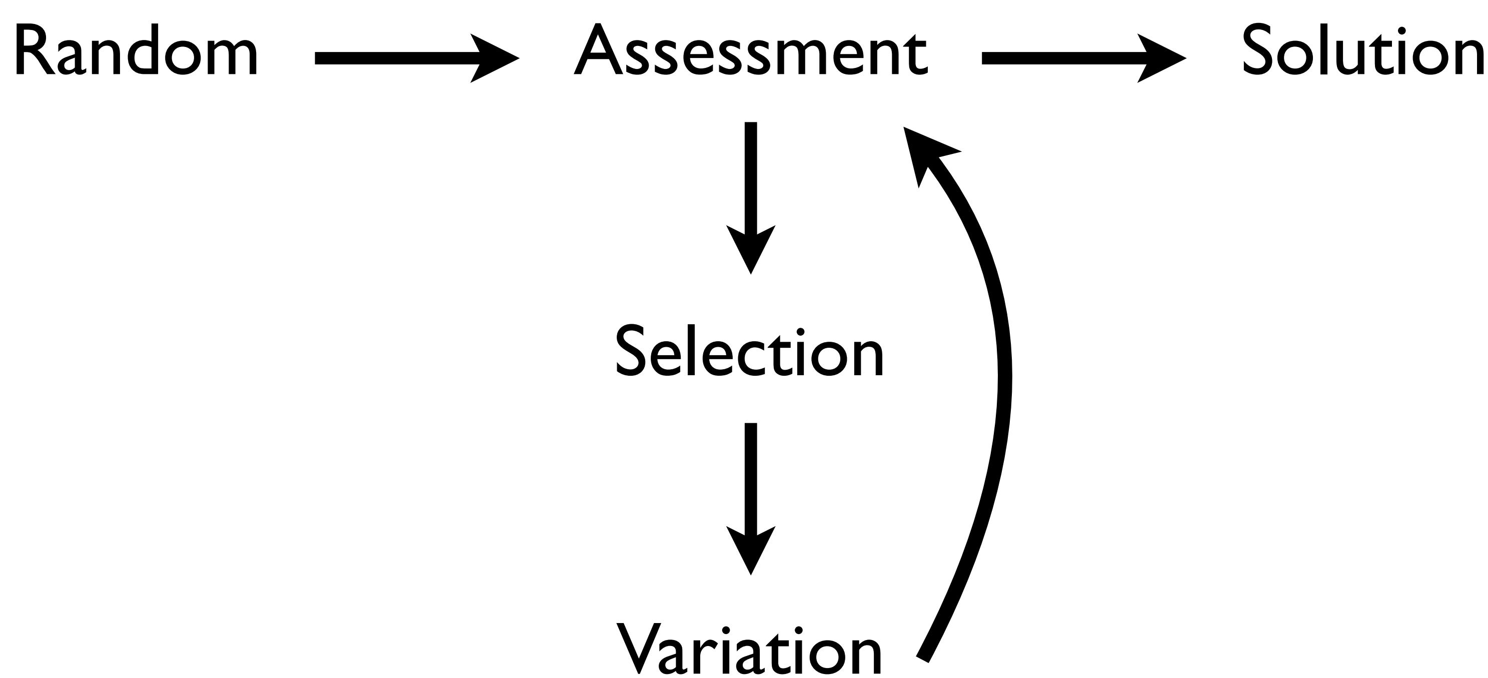
Meta*

- Individuals are GA/GP configurations; fitness test includes a full run of a GA/GP system
- Co-evolving populations of problem-solvers and variation operators

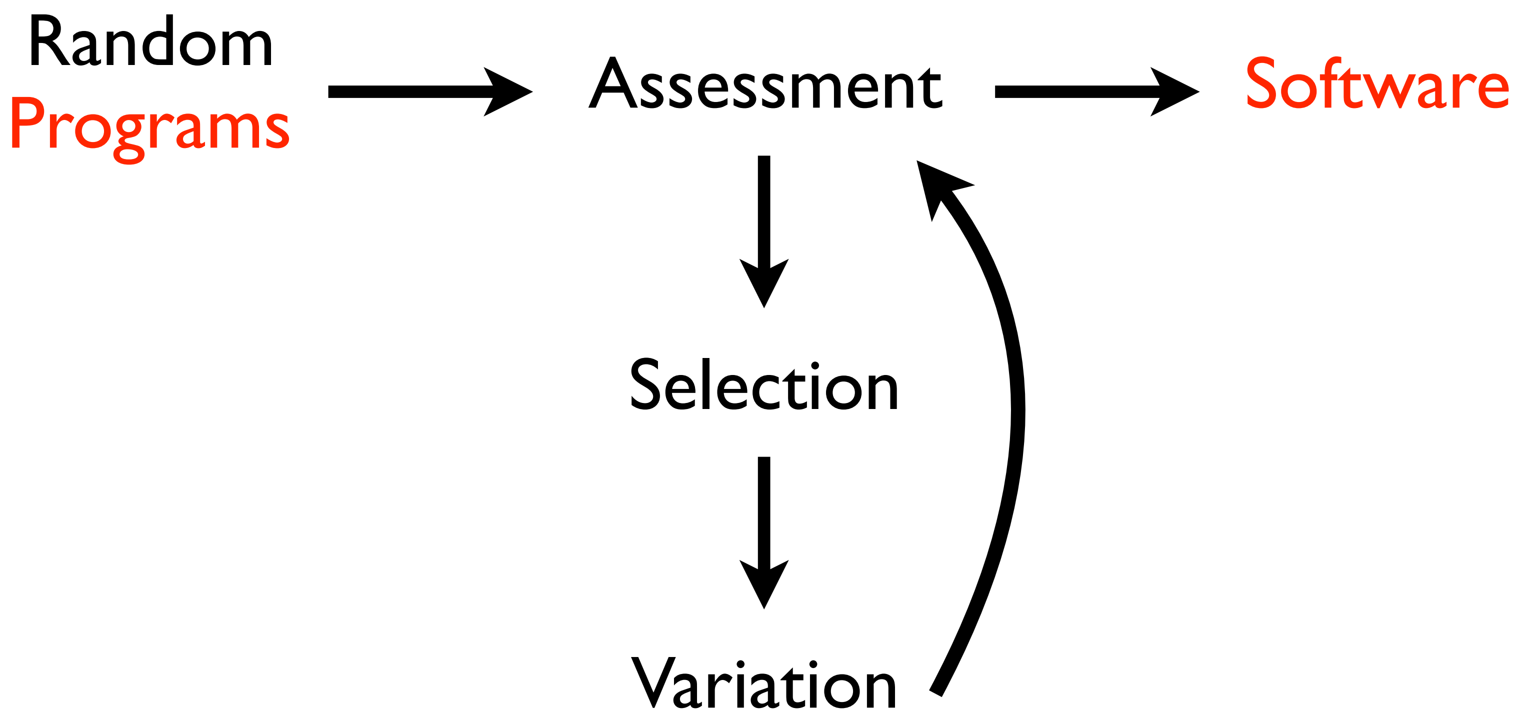
Autoconstruction

- 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

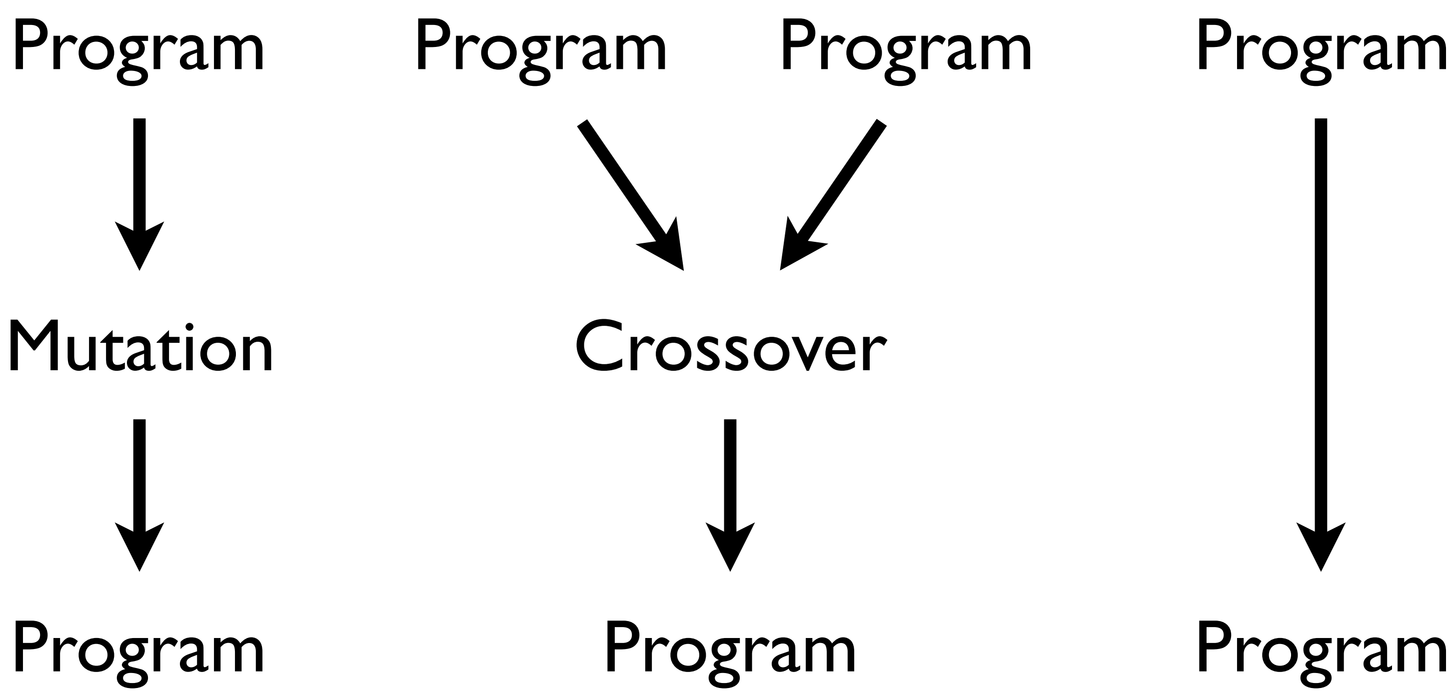
Evolutionary Computing



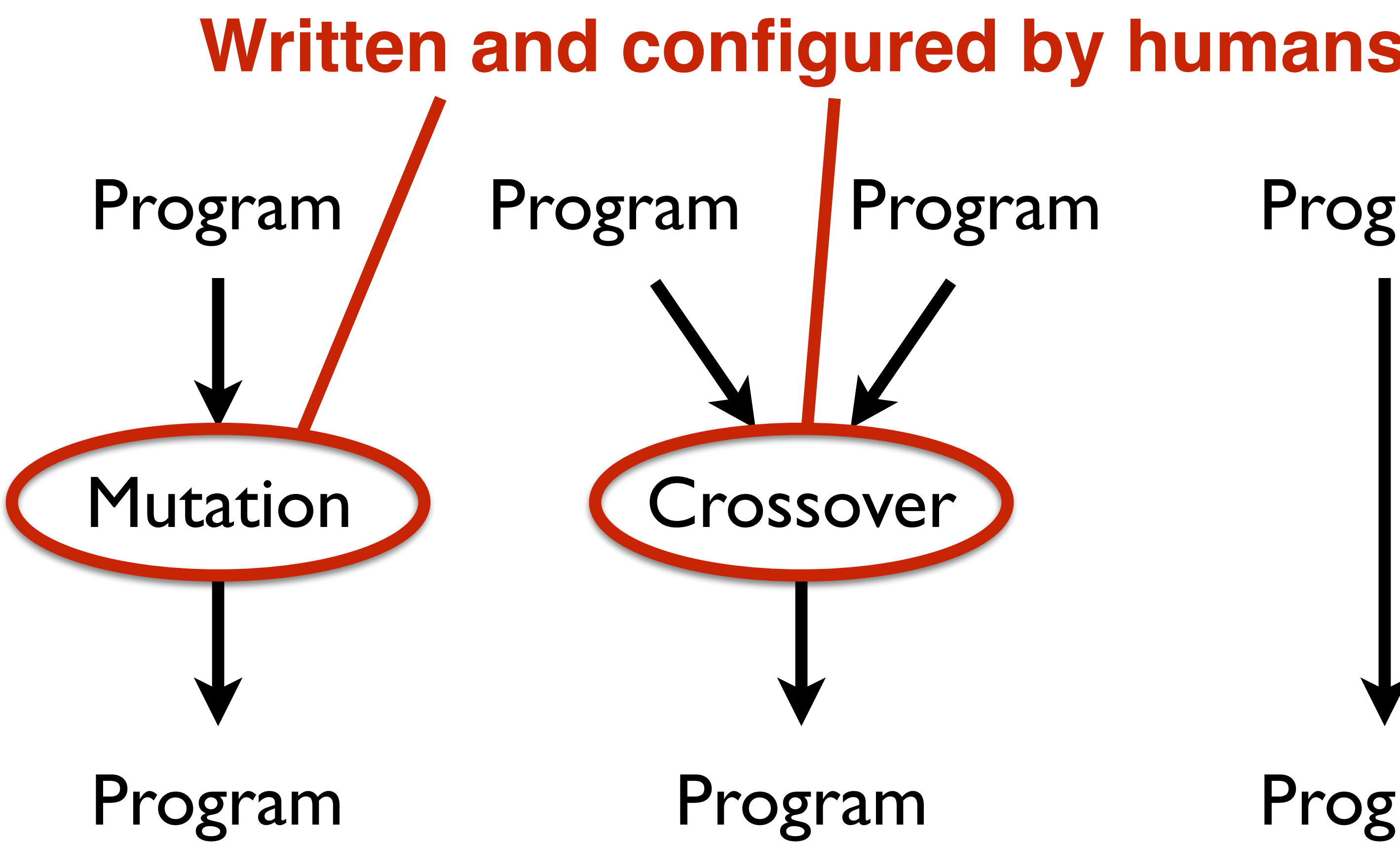
Genetic Programming



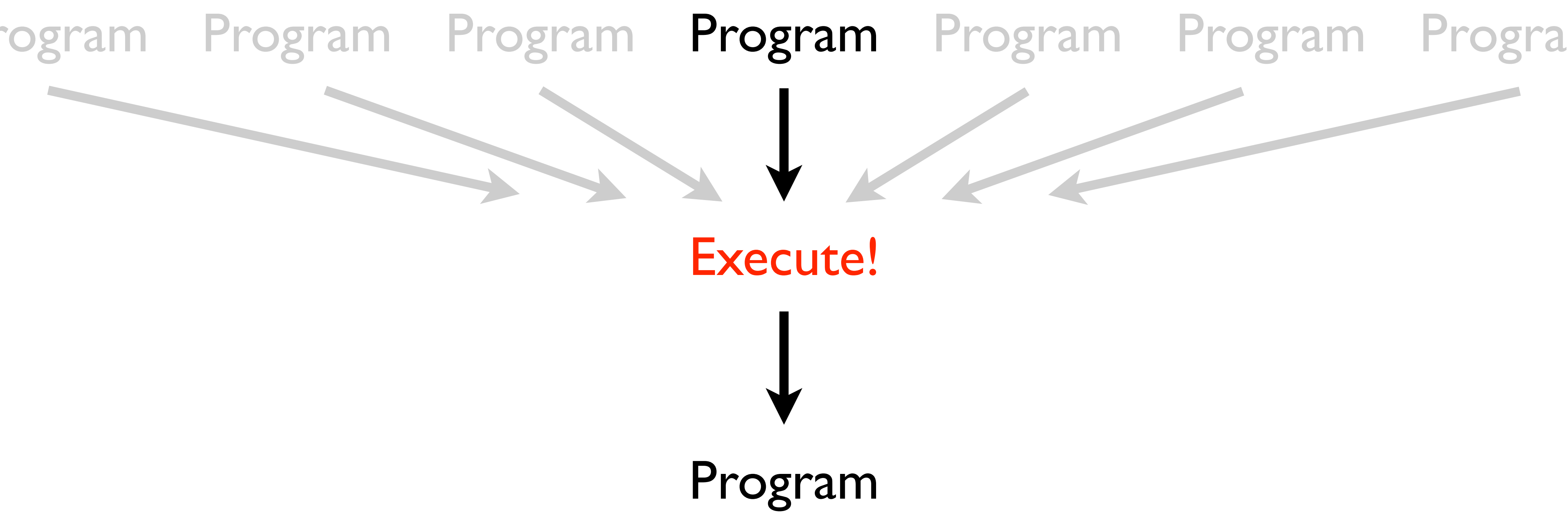
Variation in GP



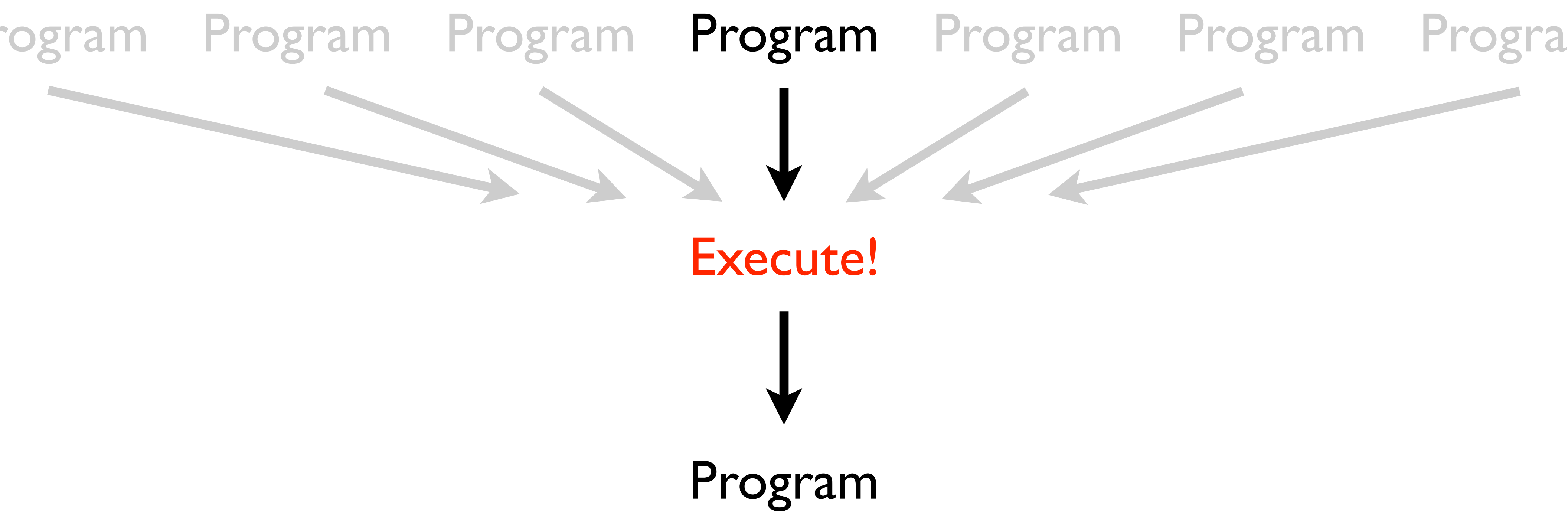
Variation in GP



Autoconstruction



Autoconstruction



A bit more complicated when genomes distinguished from programs

Autoconstructive Evolution

- 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

- A 15 year old project (building on older and broader-based ideas)
- Like genetic programming, but harder and less successful! But with greater potential?
- GECCO-2016: AutoDoG, sometimes solve significant problems, intriguing patterns of **evolving evolution**
- **Push** makes it easy and natural

Push

- Programming language for programs that evolve
- Data flows via per-type stacks, not syntax
- Trivial syntax, rich data and control structures
- PushGP: GP system that evolves Push programs
- C++, Clojure, Common Lisp, Elixir, Java, Javascript, Python, Racket, Ruby, Scala, Scheme, Swift
- http://pushlanguage.org

Early Autoconstruction

- 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

GECCO-2016 (ECADA)

Evolution Evolves with Autoconstruction

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

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

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

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

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

AutoDoG (GECCO-2016)

Autoconstructive **D**iversification **o**f **G**enomes

1. Construct genomes, not programs
2. Distinct mode/phase for construction of offspring
3. Select combinatorially, not on aggregate error
4. Enforce diversification constraints

[1. Construct genomes, not programs]

- Previous: Push programs, on code stacks, Lisp-inspired code-manipulation instructions
- AutoDoG: Plush genomes, linear with epigenetic markers, translated to Push programs prior to running

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

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

[2. Distinct mode/phase for construction of offspring]

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

[3. Select combinatorially, not on aggregate error]

- Previous: Parents selected using standard, error aggregating methods (tournament selection)
- 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

- Selected parent may be specialist, not great on average, but lead to generalists later

- Epsilon for floats; **leaky** in experiments below

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 o
programs during evolution. While some classic genetic r
gramming problems, such as the artificial ant problem a
lawnmower problem [1], involve only single test ca
others involve large numbers of tests. There are s
in which a genetic programming system can
test cases into consideration during parent s
when determining which individuals to us
when producing offspring for the nex
best choice may depend on the type

For some problems it may be be
that seek "compromises" amor

Manuscript received Novembe
5, 2014. This material is based
Foundation under Grants Nc
findings, and conclusions
are those of the authors
Science Foundation.

T. Helmuth is
ence, Universit
muth@cs.un
L. Spec
lspector
J.

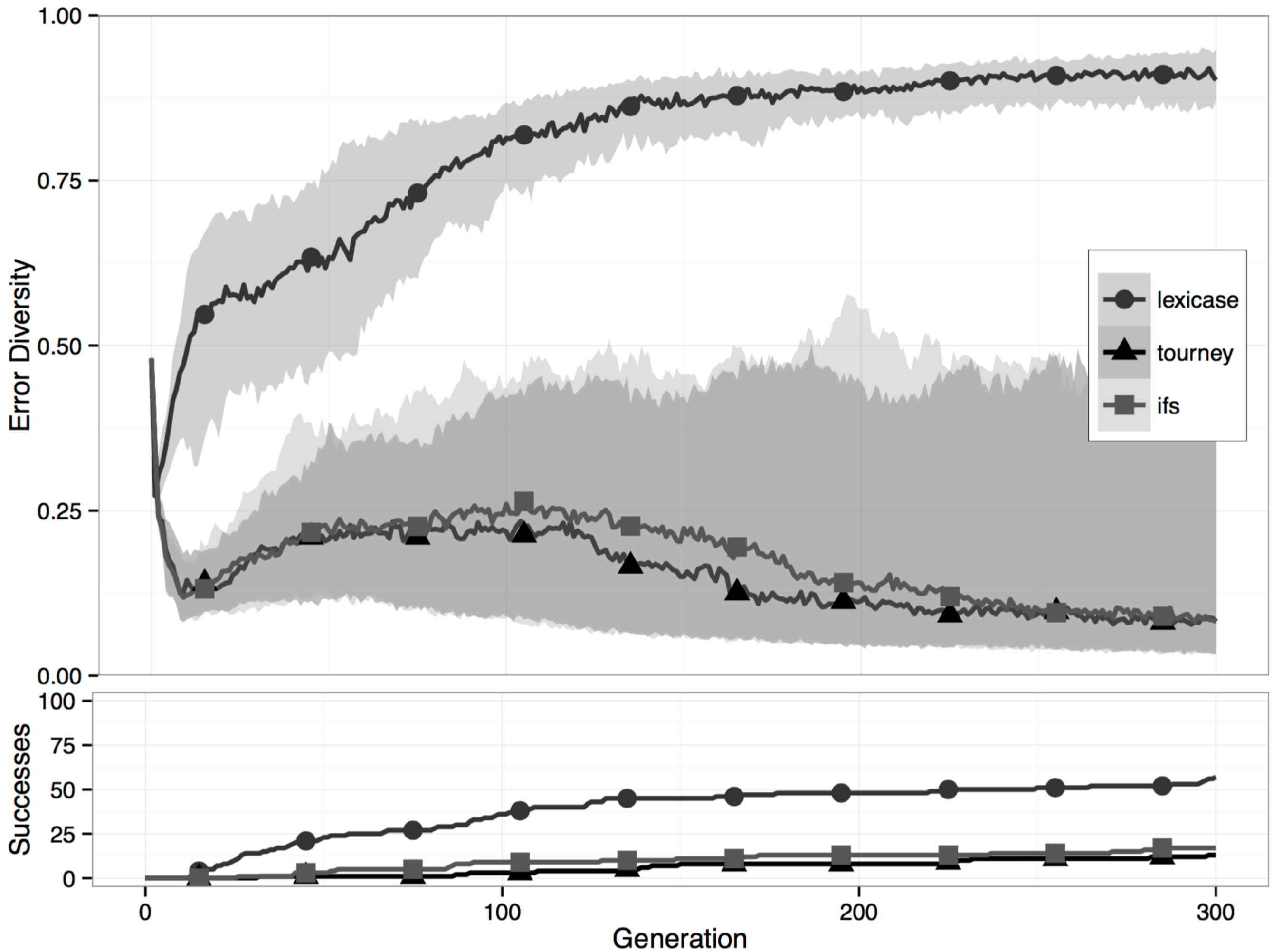
example, we can imagine a problem involving control of
simulated wind turbine in which some test cases focus
performance in low wind conditions while others fo
performance in high wind conditions. It may not be p
optimize performance on all of these test cases sim
and some sort of compromise may therefore
Many common parent selection approaches
ment selection, introduce compromises b
aggregating the performance of an ir
cases into a single fitness value. Th
may be as simple as summing t
squares, into a single error valu
as implicit fitness sharing [
based on population stati

By contrast, we wis
mising" problems: p
must perform as
perform on tha
is a probler
to perfor
for go
prob

GPTP-2015

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

Diversity



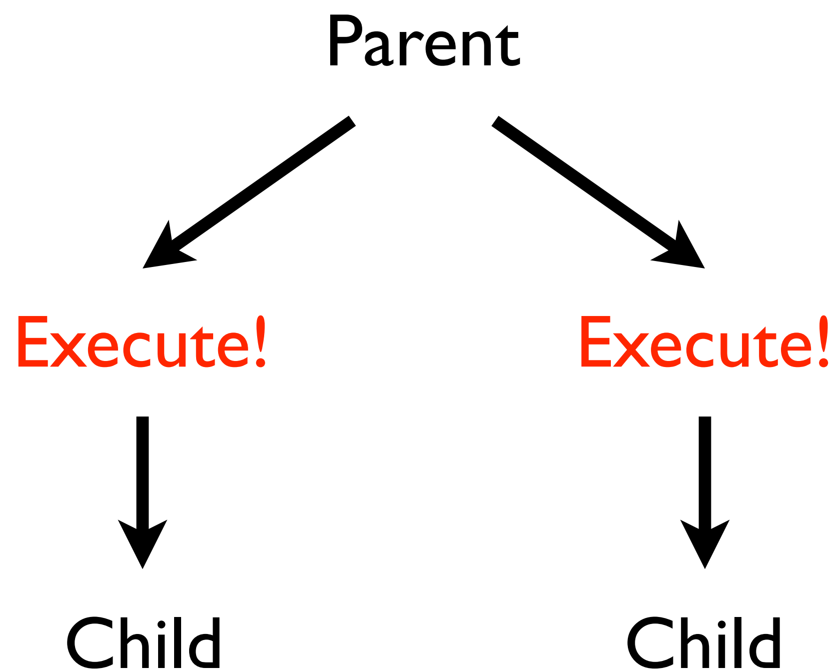
GPTP-2015

Fig. 1 Replace Space With Newline – error diversity

[4. Enforce diversification constraints]

- Previous: Various, including all but clones, or those in lineages making progress
- AutoDoG: Must satisfy diversification constraints on reproductive behavior, determined from a cascade of temporary descendants

Diversification Constraints



- Parent/child program differences positive; not same
- Many variants possible

Software Synthesis Benchmarks (GECCO 2015)

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

Solved with PushGP; only with autoconstruction

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.

- Multiple types, looping, multiple tasks
- PushGP can achieve success rates up to ~95% in 300 generations
- AutoDoG 2016 succeeded 5-10%

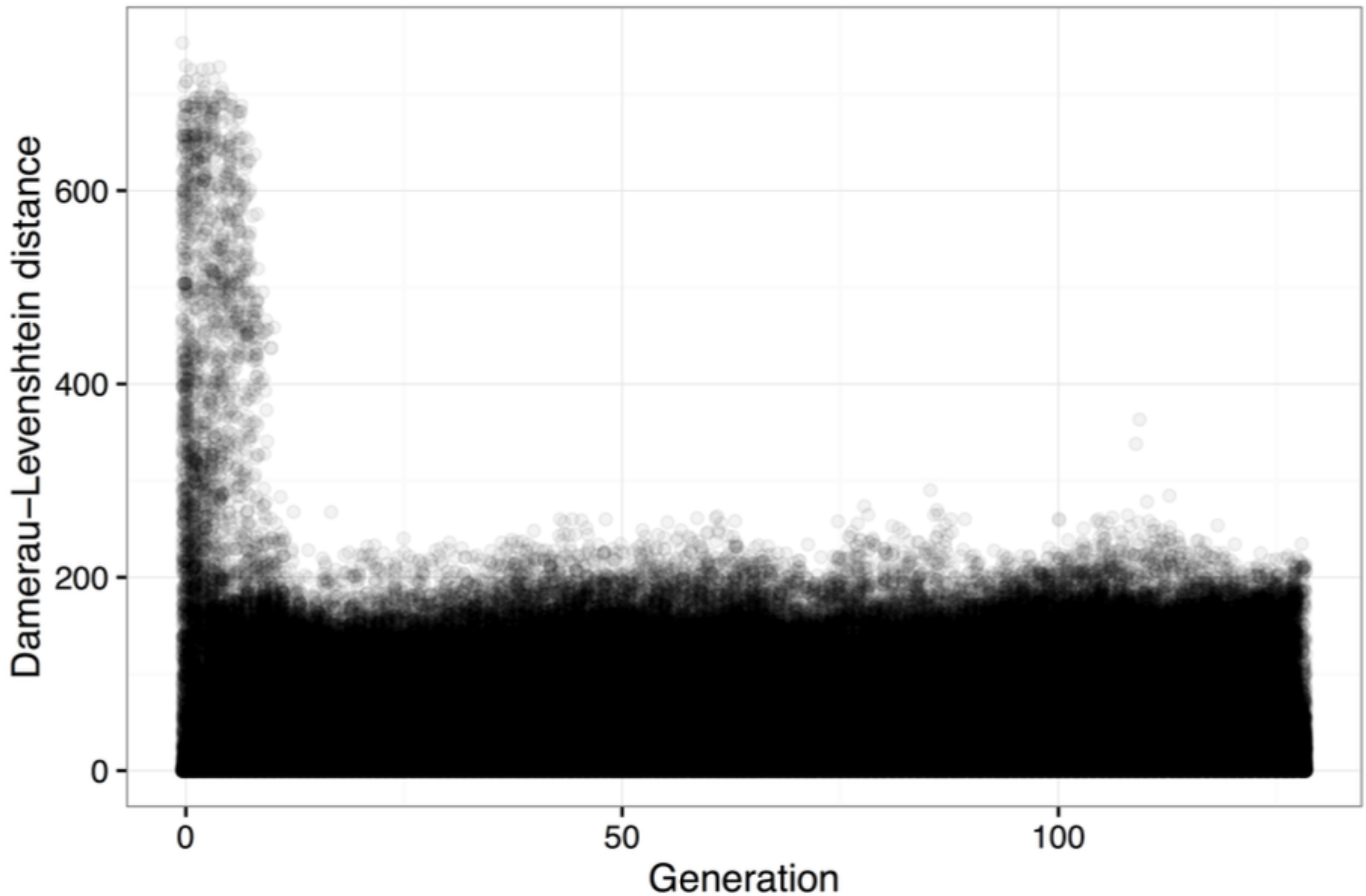


Figure 1: DL-distances between parent and child during a single non-autoconstructive run of GP on the Replace Space With Newline problem

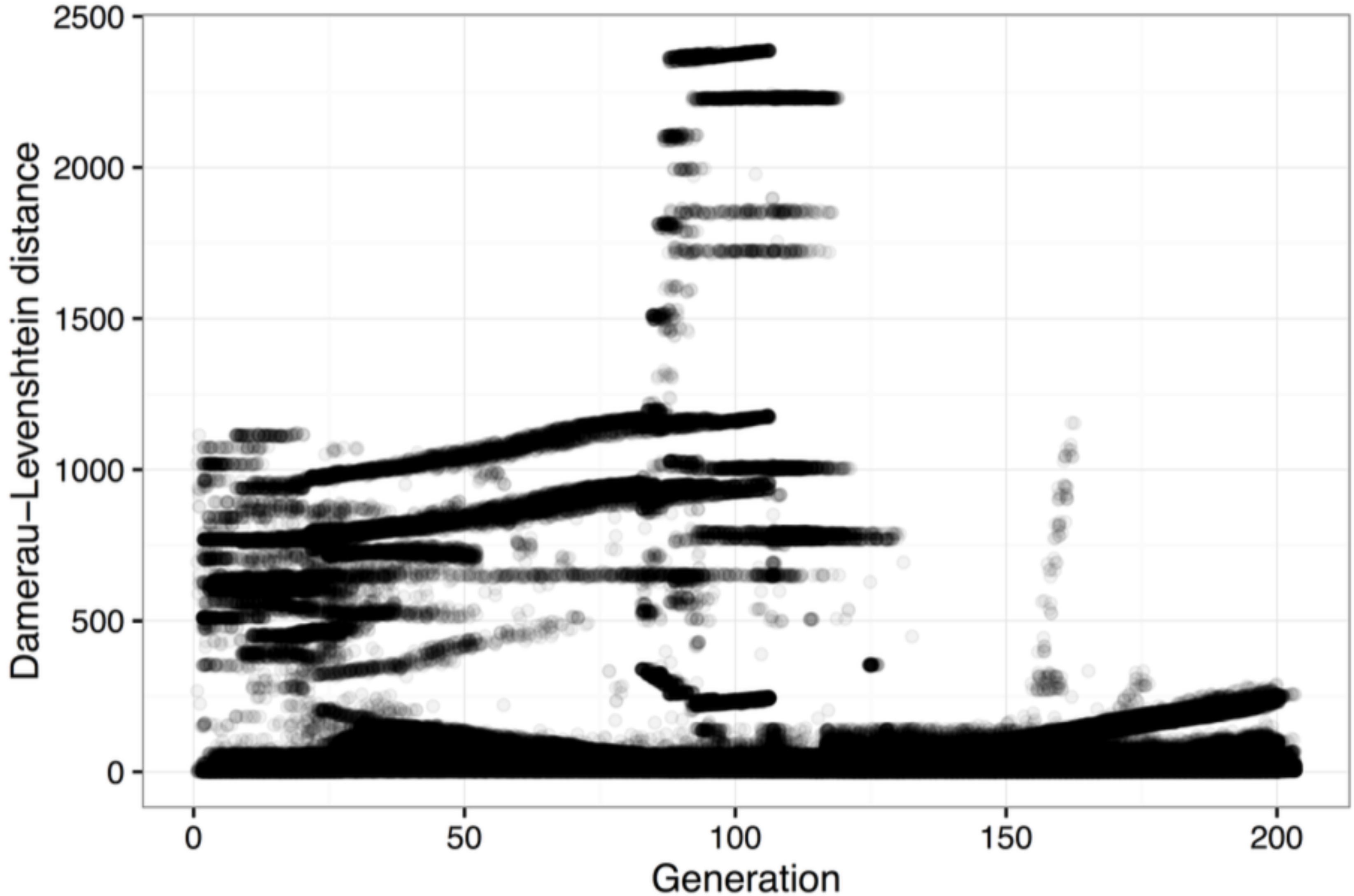


Figure 3: DL-distances between parent and child during a single autoconstructive run of GP on the Replace Space With Newline problem

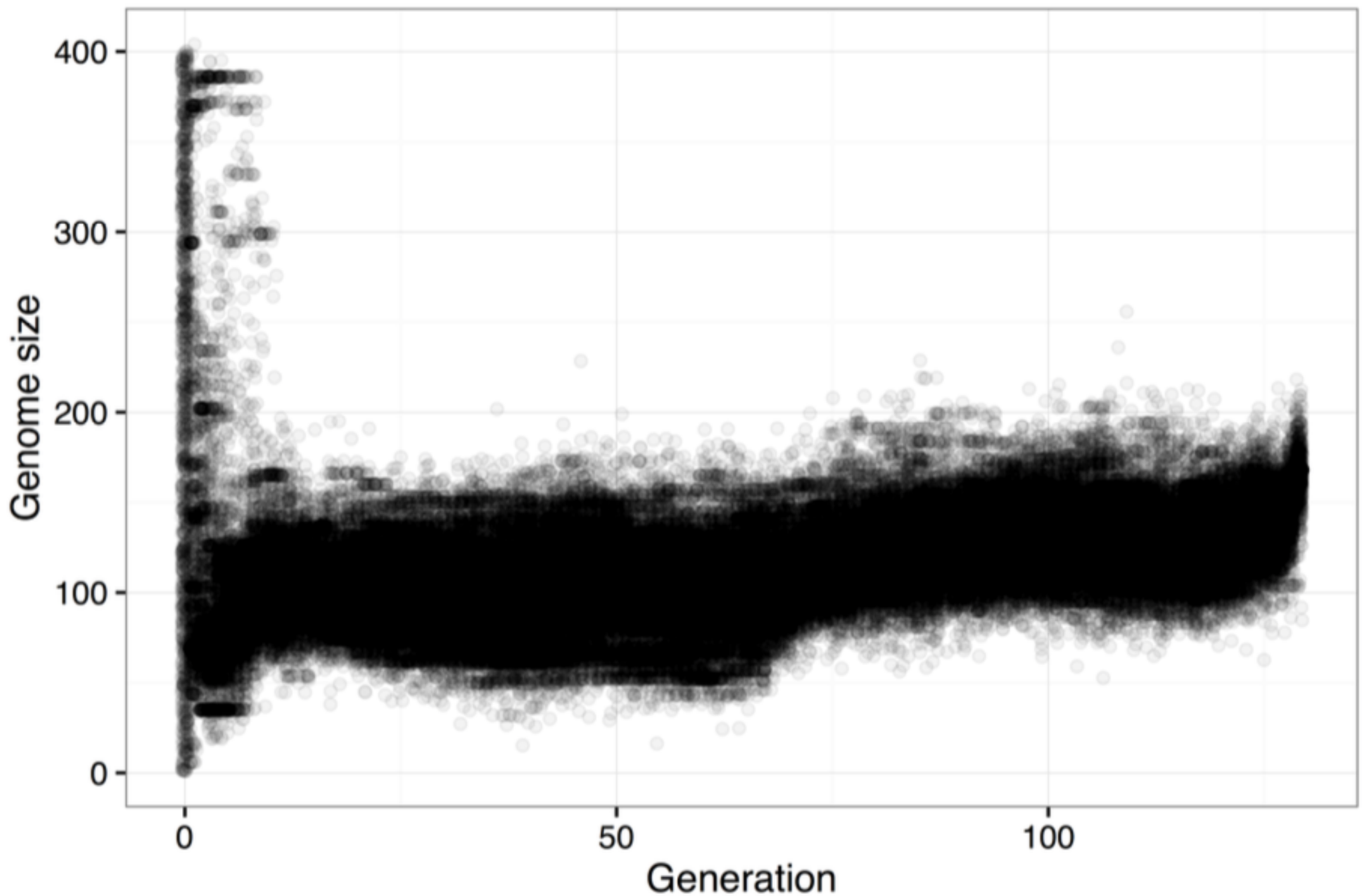


Figure 2: Genome sizes during a single non-autoconstructive run of GP on the Replace Space With Newline problem

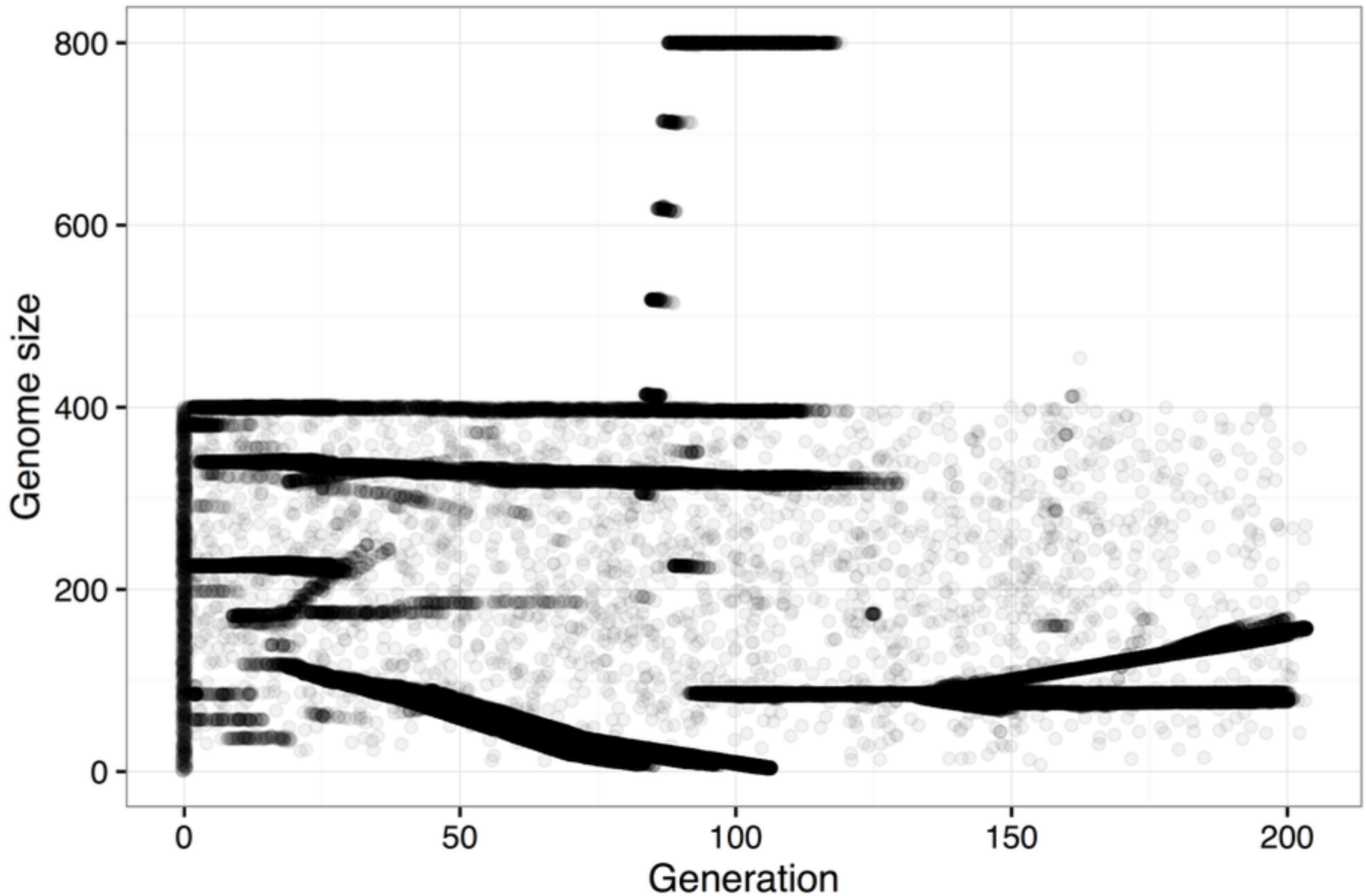
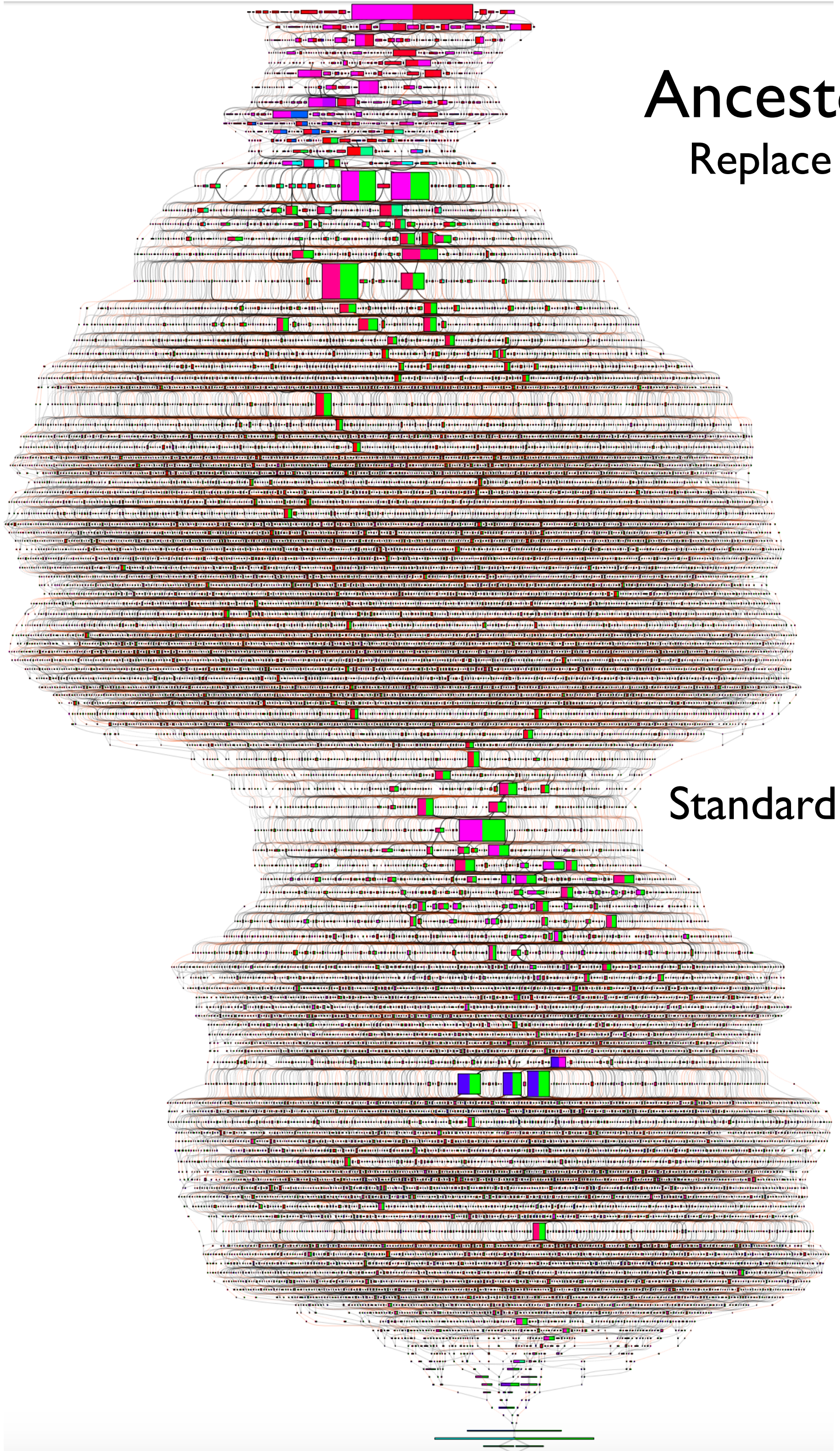


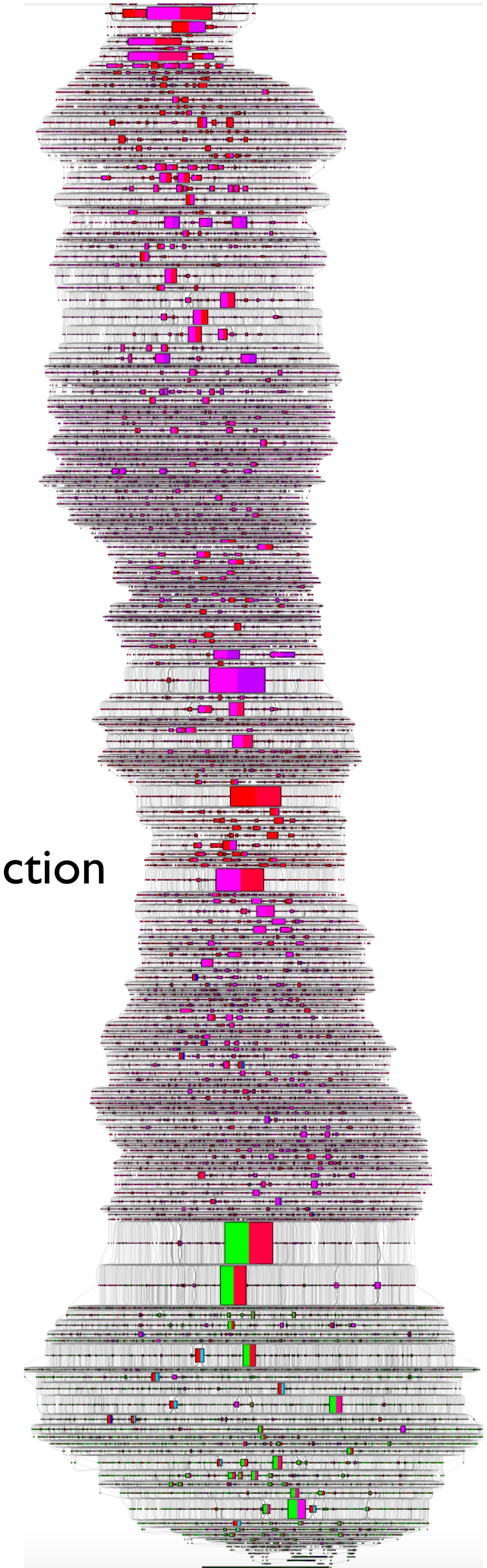
Figure 4: Genome sizes during a single autoconstructive run of GP on the Replace Space With Newline problem

Ancestors of Solutions

Replace Space with Newlines



Standard Operators



Autoconstruction

2 New Milestones

- Autoconstructive evolution can succeed as much and as fast as non-autoconstructive evolution
- Autoconstructive evolution can solve a problem not yet solved without it

2.5 New Features

- DSL for uniform genome manipulation
- Entropy
- Age-Mediated Parent Selection (AMPS)

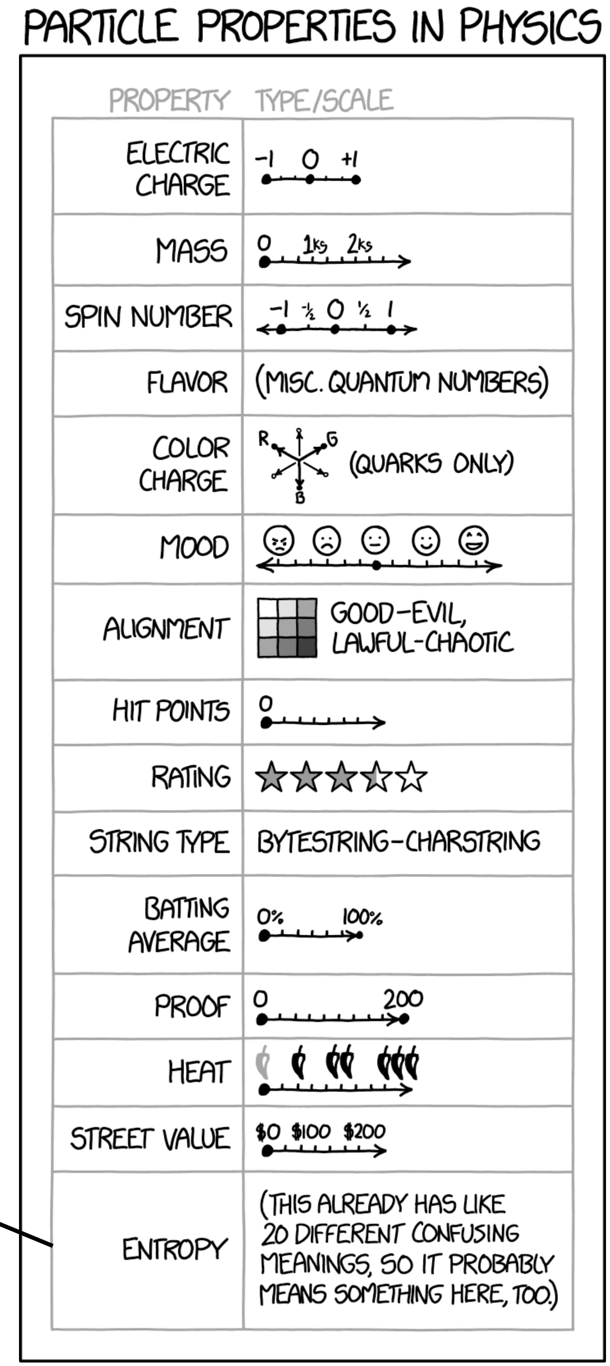
DSL for Uniform Genome Manipulation

```
genome_alternation
genome_genesis
genome_new
genome_parent1
genome_parent2
genome_uniform_addition
genome_uniform_addition_and_deletion
genome_uniform_boolean_mutation
genome_uniform_close_mutation
genome_uniform_combination_and_deletion
genome_uniform_crossover
genome_uniform_deletion
genome_uniform_float_mutation
genome_uniform_instruction_mutation
genome_uniform_integer_mutation
genome_uniform_silence_mutation
genome_uniform_string_mutation
genome_uniform_tag_mutation
genome_dup
genome_empty
genome_eq
genome_flush
genome_pop
genome_rot
genome_rotate
genome_shove
genome_stackdepth
genome_swap
genome_yank
genome_yankdup
```

Entropy

- Random gene deletions after autoconstruction
- Like "cosmic ray mutations" but purely destructive
- All new genetic material must stem from autoconstructive instructions
- Lineages must counteract entropy to survive
- Default rate: 0.1

https://xkcd.com/1862/



Age-Mediated Parent Selection (AMPS)

- Use genealogical age to bias in favor of youth
- Like ALPS (but simpler), and age-fitness Pareto optimization (but for parent selection)
- For each parent, consider only younger than a limit chosen randomly from ages in the population
- Options for age-combining functions; for autoconstruction: age of executing parent + maximum similarity with a parent, scaled to [0,1]

Rivaling Ordinary PushGP

- Uniform DSL + Entropy + AMPS
- In 20 runs, 75% success within 300 generations on Replace Space With Newline (100% by generation 628); 80% on Mirror Image
- Surprisingly, rivals ordinary GP on a problem that ordinary GP can solve

8. **String Differences (P 4.4)** Given 2 strings (without whitespace) as input, find the indices at which the strings have different characters, stopping at the end of the shorter one. For each such index, print a line containing the index as well as the character in each string. For example, if the strings are "dealer" and "dollars", the program should print:

```
1 e o
2 a l
4 e a
```

Extending the Reach of GP

- Without autoconstruction, string difference not yet solved by GP, despite many efforts/configurations
- 3 autoconstructive solutions so far, with Uniform DSL + Entropy

First Evolved Solution

- Makes children using uniform addition, with a rate (~0.0921) close to the entropy rate (0.1)
- Solves problem in general way, with a few clever tricks (like using the depth of the boolean stack to track the comparison index)

Future

- Use autoconstruction to solve other previously unsolved problems
- Study how autoconstruction works, to improve it
- Consider implications for study of evolution of biological evolution

Thanks

- Nic McPhee, Tom Helmuth, Maggie M. Casale, and Julian Oks
- Members of the Hampshire College Computational Intelligence Lab
- 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. 1617087, 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.