Genetic Programming with Autoconstructive Evolution

Lee Spector
Hampshire College & UMass Amherst

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.

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

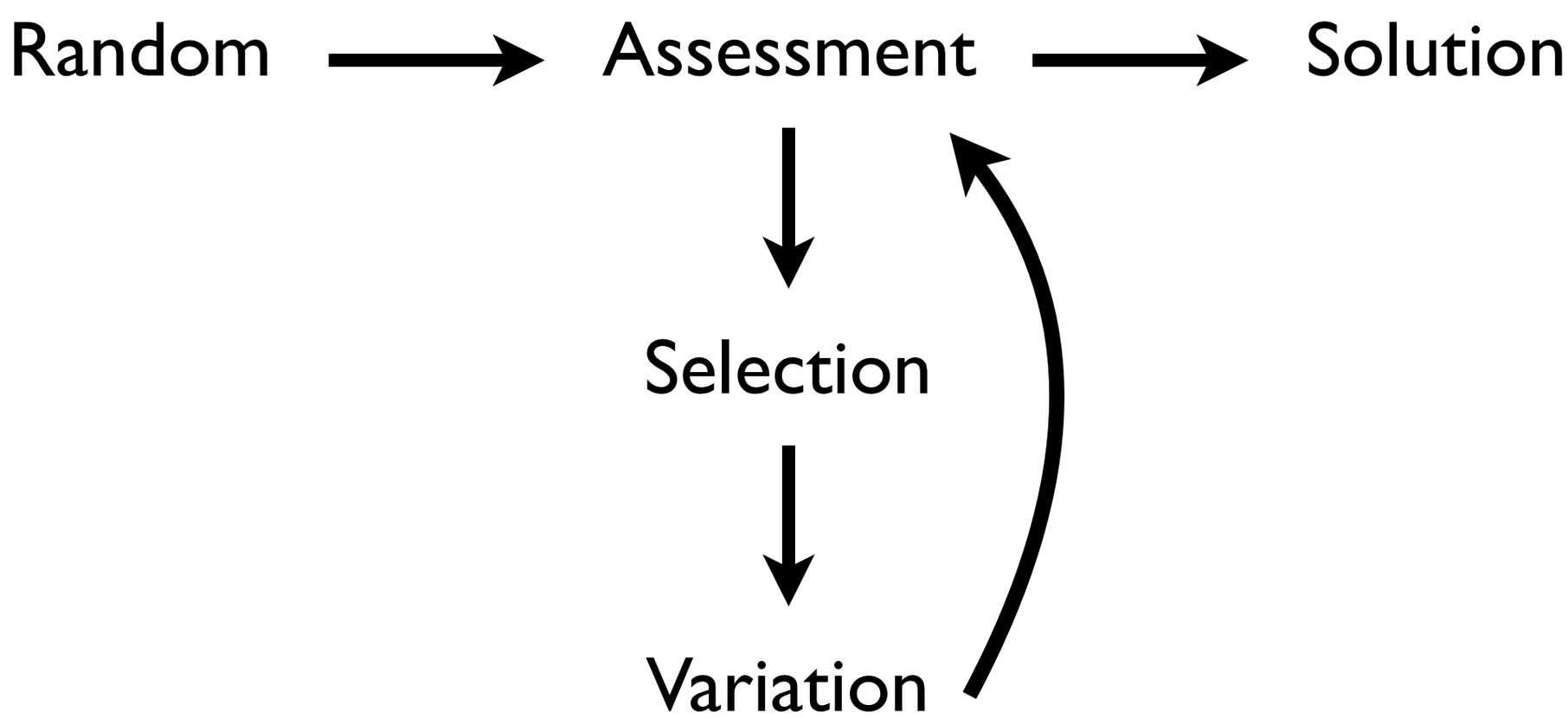
- Co-evolving populations of problem-solvers and variation operators

Autoconstruction

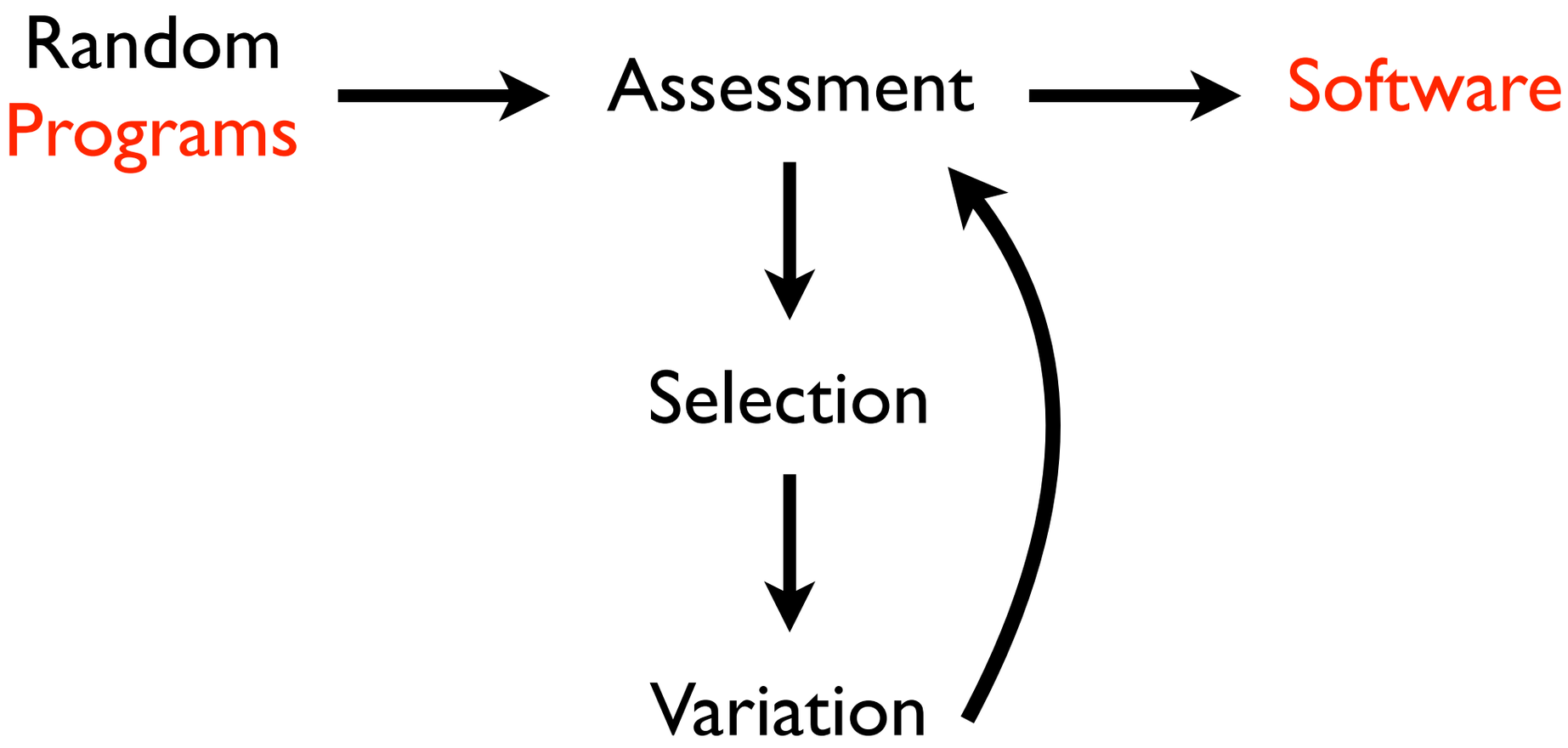
- Individual programs make their own children
- In doing so, they control their own mutation and recombination rates and methods, and in some cases mate selection, etc.

- The machinery of reproduction and diversification (i.e., the machinery of evolution) evolves

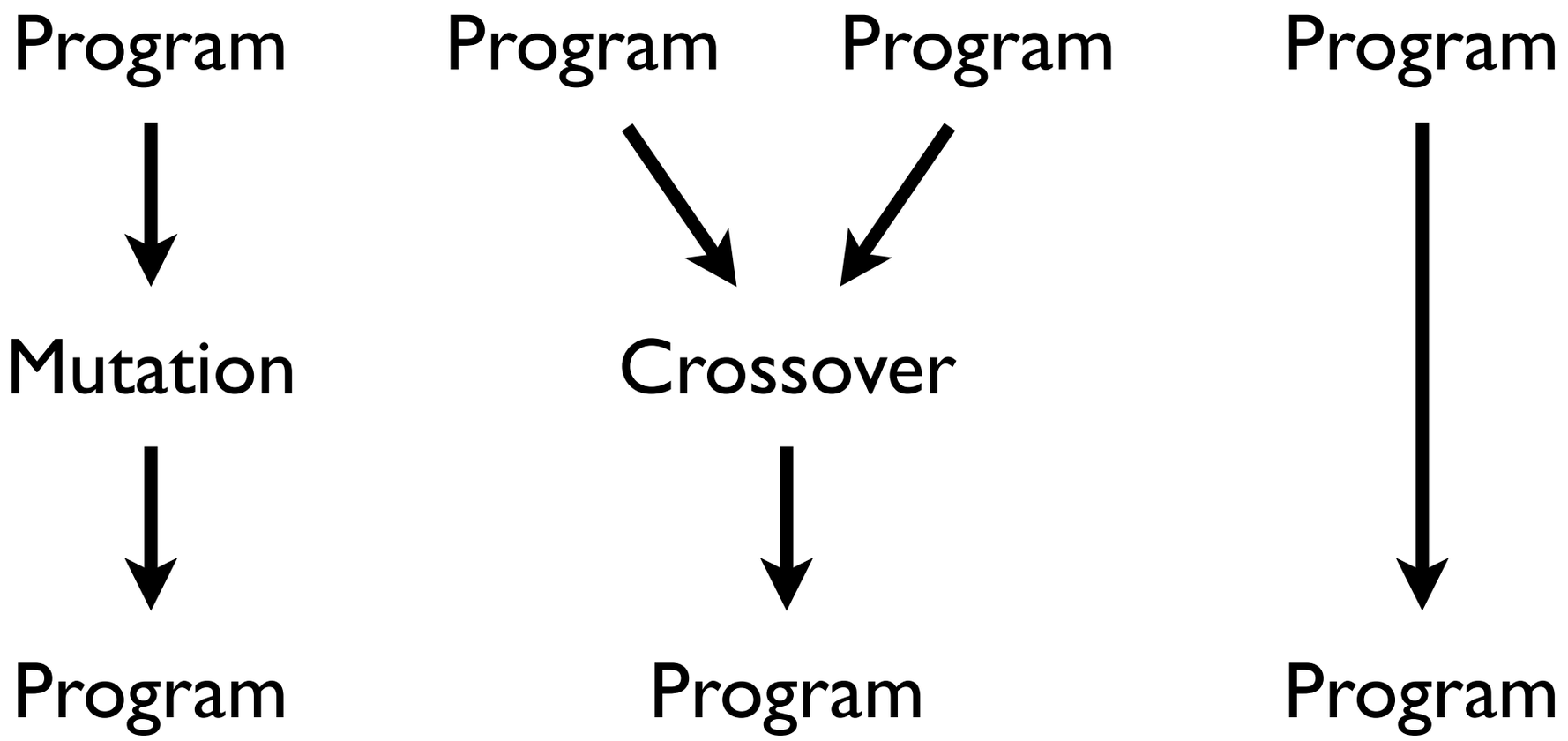
Evolutionary Computing



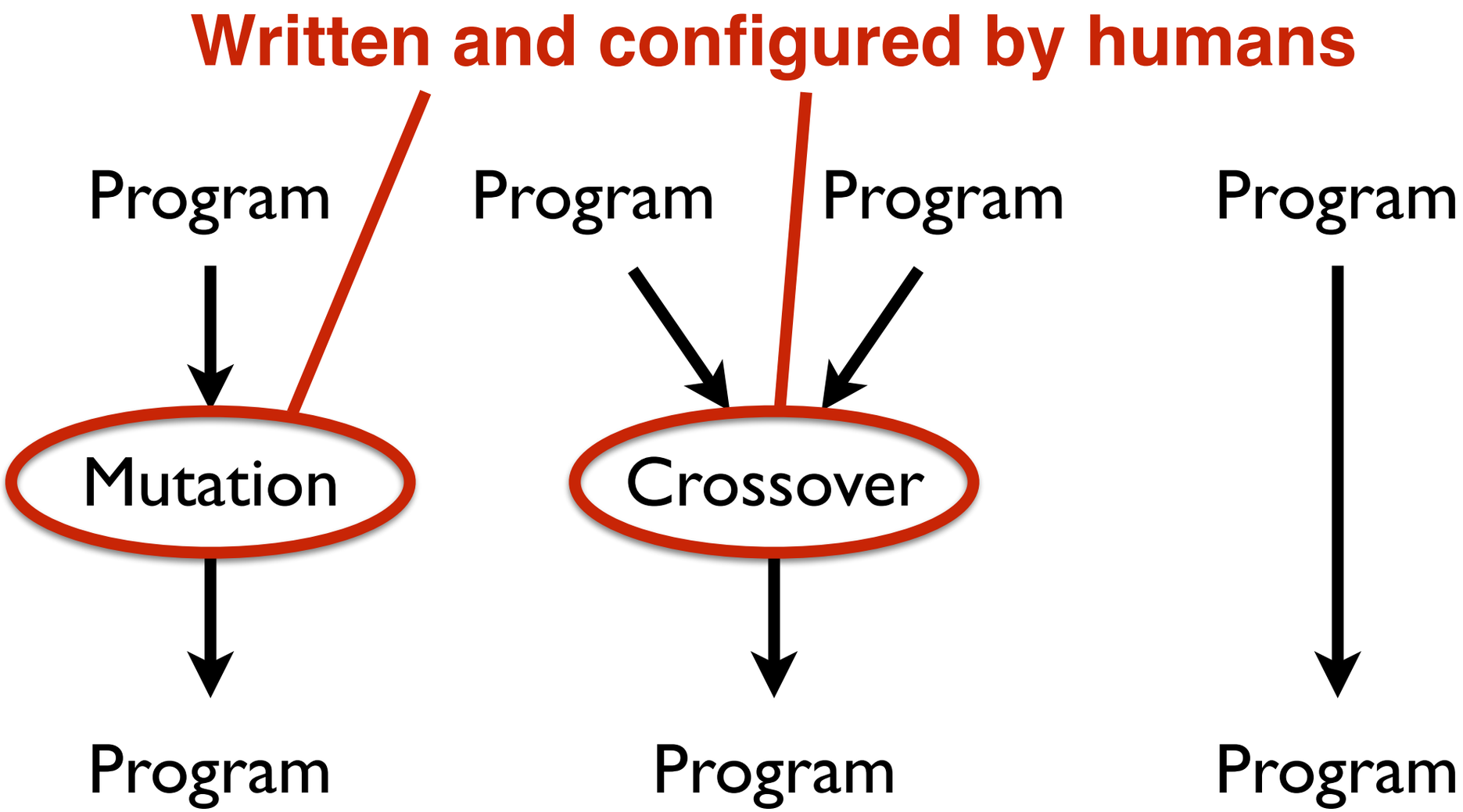
Genetic Programming



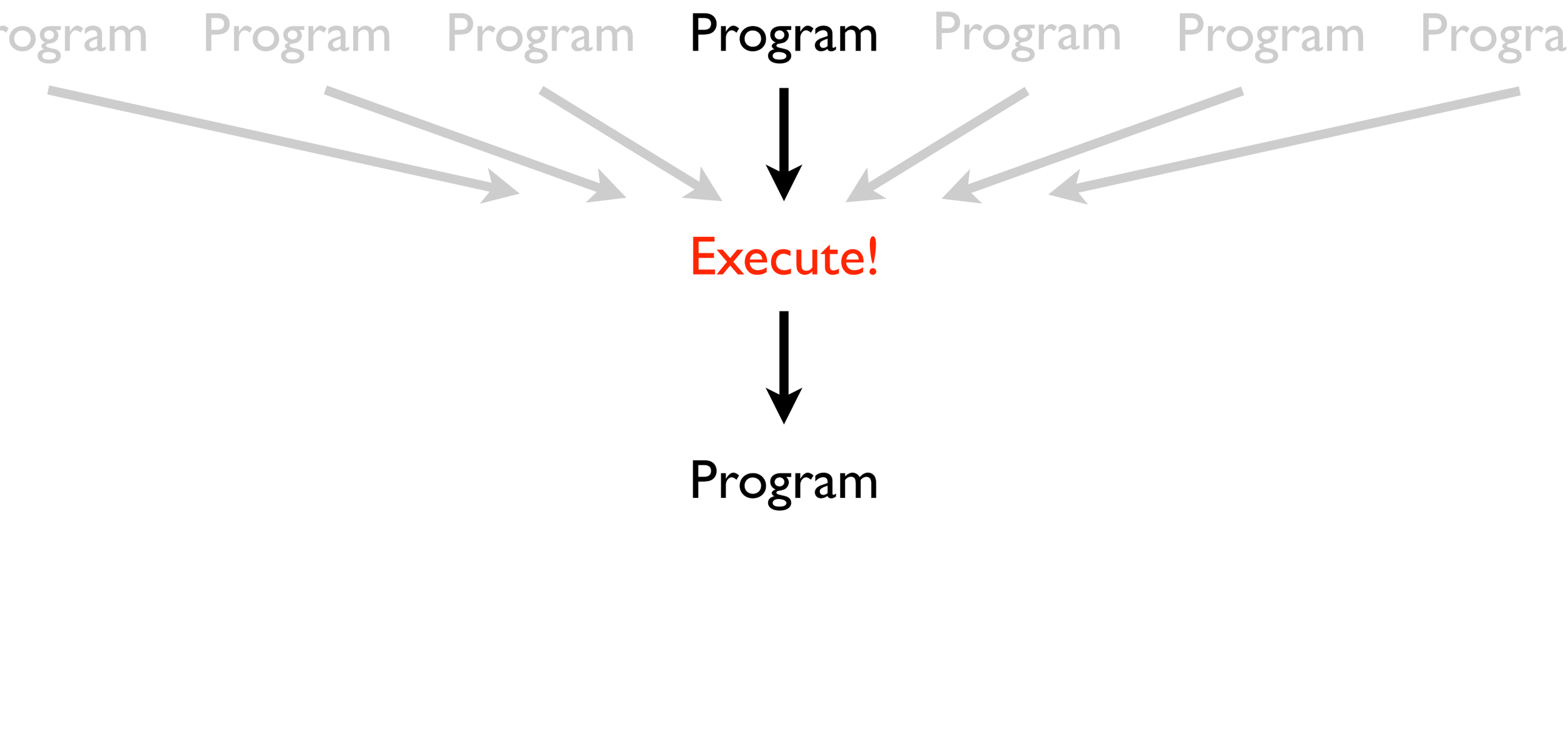
Variation in GP



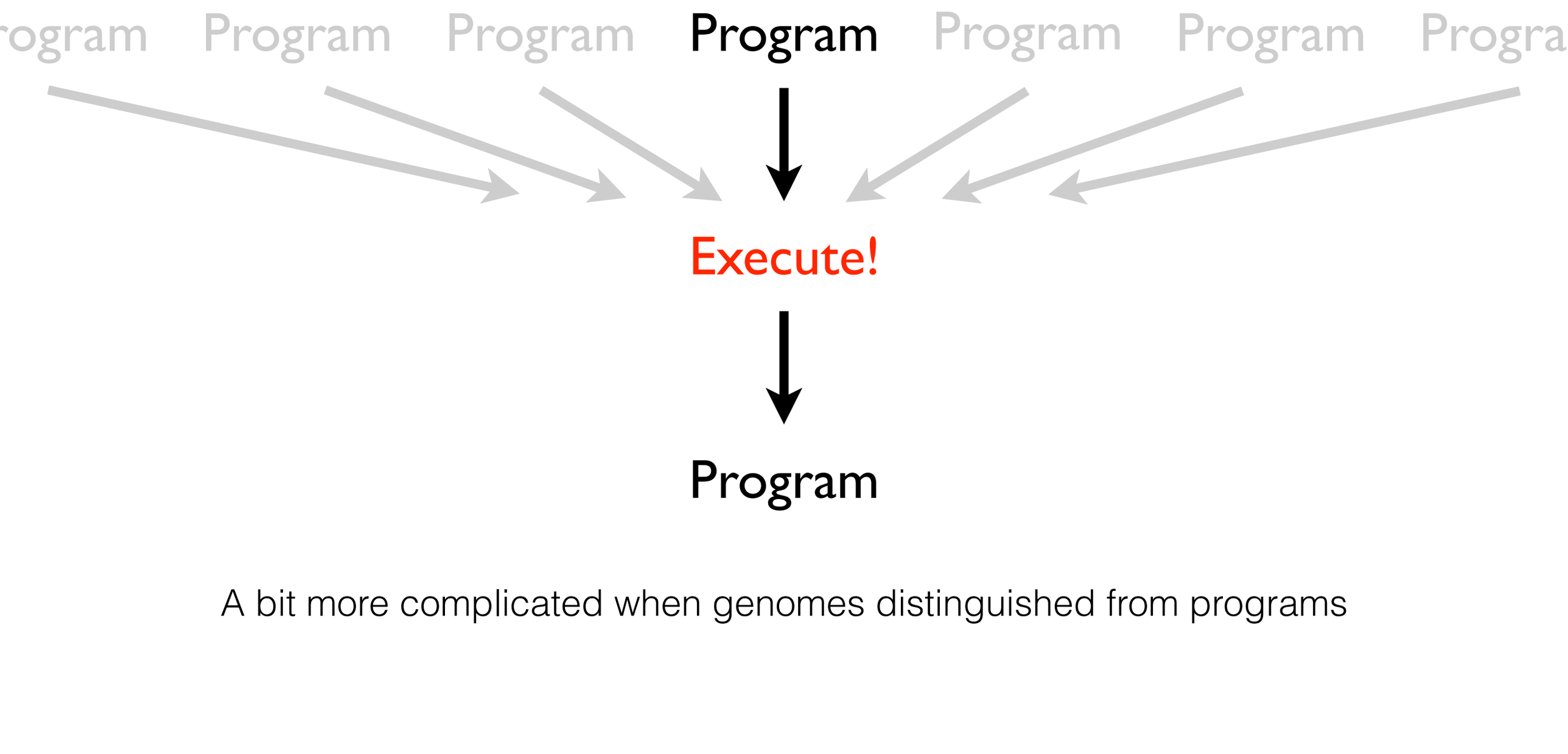
Variation in GP



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

Push (2)

- One stack per type:
  integer, float, boolean, string, code, exec, vector, ...

- `program → instruction | literal | ( program* )`

- Turing complete

- Missing argument? NOOP

- Time/step limits ensure termination, with results available from stacks in all cases

Clojush

Lee Spector (lspector@hampshire.edu), started 20100227 See version history. Older version history is in `old-version-history.txt`.

This is the README file accompanying Clojush, an implementation of the Push programming language and the PushGP genetic programming system in the Clojure programming language. Among other features this implementation takes advantage of Clojure's facilities for multi-core concurrency.

Availability

https://github.com/lspector/Clojush/

Requirements

To use this code you must have a Clojure programming environment; see http://clojure.org/. The current version of Clojush requires Clojure 1.7.0.

Clojure is available for most OS platforms. A good starting point for obtaining and using Clojure.

Quickstart

Using Leiningen you can run an example from the OS command line (in the Clojush directory) with a call like:

```
lein run clojush.problems.demos.simple-regression
```

Pysh

Push Genetic Programming in Python. For the most complete documentation, refer to the ReadTheDocs.

http://pysh2.readthedocs.io/en/latest/

Push Genetic Programming

Push is programming language that plays nice with evolutionay computing / genetic programming. It is a stack-based language that features 1 stack per data type, including code. Programs are represented by lists of instructions, which modify the values on the stacks. Instuctions are executed in order.

More information about PushGP can be found on the Push Redux and the Push Homepage.

For the most cutting edge PushGP framework, see the Clojure implementaion called Clojush.

Installing Pysh

> Pysh is compatale with python `2.7.x` and `3.5.x`

Install from pip

Coming whith first beta release of `pyshgp`. Check the Roadmap to get a sense of how far off this is.

Why Push?

- Expressive: data types, data structures, variables, conditionals, loops, recursion, modules, ...
- Elegant: minimal syntax and a simple, stack-based execution architecture
- Supports several forms of meta-evolution
- Evolvable? At minimum, supports investigation of relations between expressiveness and evolvability

Push Program Execution

- Push the program onto the exec stack.
- While exec isn't empty, pop and **do** the top:
  - If it's an instruction, execute it.
  - If it's a literal, push it onto the appropriate stack.
  - If it's a list, push its elements back onto the exec stack one at a time.

Instructions for Most Types

- `<type>_dup`
- `<type>_empty`
- `<type>_eq`
- `<type>_flush`
- `<type>_pop`
- `<type>_rot`
- `<type>_shove`
- `<type>_stackdepth`
- `<type>_swap`
- `<type>_yank`
- `<type>_yankdup`

Selected Integer Instructions

```
integer_add integer_dec integer_div
integer_gt integer_fromstring integer_min
integer_mult integer_rand
```

Selected Boolean Instructions

```
boolean_and boolean_xor boolean_frominteger
```

Selected String Instructions

```
string_concat string_contains string_length
string_removechar string_replacechar
```

Selected Exec Instructions

Conditionals:

```
exec_if exec_when
```

General loops:

```
exec_do*while
```

“For” loops:

```
exec_do*range exec_do*times
```

Looping over structures:

```
exec_do*vector_integer exec_string_iterate
```

Combinators:

```
exec_k exec_y exec_s
```

More Types and Instructions

code_atom code_car print_newline integer_sub integer_inc boolean_stackdepth return_exec_pop vector_integer_eq autoconstructive_integer_rand boolean_pop genome_close_inc string_fromchar vector_string_shove zip_yankdup genome_new vector_float_yankdup exec_yankdup vector_integer_shove integer_yankdup string_flush boolean_swap zip_empty exec_shove vector_boolean_yank code_eq exec_y boolean_yank integer_eq genome_silence string_butlast code_contains string_conjchar code_do*count vector_float_last genome_pop string_substring integer_mult code_length vector_integer_dup boolean_or code_position boolean_empty zip_fromcode print_vector_string vector_boolean_swap return_frominteger vector_float_pushall char_iswhitespace code_cdr exec_do*vector_integer integer_rand vector_string_replacefirst string_first vector_boolean_first exec_do*while exec_string_iterate string_indexofchar vector_float_replace integer_fromstring code_list code_swap char_frominteger genome_gene_randomize vector_integer_emptyvector vector_string_eq vector_float_butlast exec_empty zip_end? exec_fromzipnode string_shove vector_boolean_pushall zip_insert_left_fromcode exec_rot vector_string_concat vector_float_indexof code_pop vector_string_subvec vector_integer_swap code_subst char_pop return_string_pop zip_yank exec_dup vector_integer_butlast vector_float_rest vector_string_flush boolean_fromfloat code_fromziprights float_sin boolean_flush char_isdigit float_lte exec_fromziproot vector_integer_empty print_code vector_string_stackdepth string_reverse exec_k vector_integer_yank float_frominteger char_rot print_char vector_integer_stackdepth vector_boolean_concat boolean_xor integer_gte genome_yankdup vector_float_shove vector_integer_take code_quote string_replacefirst return_fromstring exec_fromziplefts vector_integer_yankdup boolean_shove float_lt vector_string_dup vector_string_occurrencesof vector_integer_replace zip_branch? vector_float_reverse float_mod vector_float_subvec string_last print_boolean boolean_rot vector_string_rest integer_div vector_float_remove integer_fromfloat integer_lte code_fromzipchildren environment_end vector_integer_rot integer_mod string_concat vector_string_butlast genome_swap code_null exec_do*count vector_float_emptyvector vector_string_yankdup integer_rot float_yankdup vector_string_rot zip_replace_fromexec vector_string_take integer_add vector_integer_occurrencesof integer_shove genome_dup return_code_pop char_swap integer_max return_fromexec code_wrap return_float_pop code_flush genome_yank zip_shove vector_integer_flush vector_integer_subvec vector_boolean_indexof vector_float_pop vector_string_remove vector_integer_contains zip_remove code_append vector_float_eq vector_integer_conj string_eq zip_leftmost code_yankdup code_rot integer_stackdepth float_max vector_boolean_set zip_append_child_fromexec zip_next vector_float_conj zip_fromexec string_take zip_left zip_replace_fromcode char_stackdepth return_fromchar genome_eq vector_integer_replacefirst float_stackdepth code_fromziproot float_fromchar float_gt boolean_dup float_fromboolean code_fromzipnode genome_rot vector_float_replacefirst vector_boolean_conj vector_boolean_dup vector_integer_indexof vector_string_swap exec_eq string_emptystring string_swap integer_yank exec_while float_empty print_vector_boolean integer_min exec_swap genome_rotate integer_fromchar vector_string_yank string_stackdepth code_do*range string_replacechar char_allfromstring vector_integer_rest vector_boolean_length char_yank vector_float_empty code_fromfloat genome_parent2 return_fromcode string_pop float_eq vector_boolean_empty zip_insert_child_fromexec vector_string_last string_nth code_do* return_zip_pop vector_string_pop zip_rot vector_integer_nth exec_do*range exec_if char_shove zip_down zip_insert_left_fromexec code_frominteger vector_boolean_remove vector_integer_remove boolean_invert_first_then_and genome_flush print_string integer_fromboolean char_yankdup code_do vector_string_first boolean_frominteger string_setchar vector_integer_last char_isletter genome_gene_dup vector_integer_concat print_integer code_map boolean_eq float_gte return_fromfloat genome_gene_copy string_occurrencesofchar string_replacefirstchar print_float boolean_rand integer_flush float_shove string_replace char_dup float_pop char_eq vector_float_nth vector_string_conj integer_gt return_integer_pop float_sub vector_integer_length vector_float_set vector_string_indexof vector_boolean_rest code_dup vector_boolean_shove zip_eq float_min boolean_not float_mult float_fromstring genome_unsilence code_if vector_integer_pop vector_boolean_last exec_do*times zip_pop zip_rightmost float_dec vector_float_contains genome_gene_copy_range environment_new exec_do*vector_string code_nthcdr string_empty char_empty exec_pop vector_integer_set autoconstructive_boolean_rand vector_float_rot string_yankdup exec_do*vector_float string_removechar code_extract vector_string_replace vector_float_first genome_parent1 return_tagspace char_flush vector_float_occurrencesof vector_string_emptyvector float_add code_stackdepth exec_s zip_insert_right_fromexec float_dup vector_string_nth zip_stackdepth vector_integer_reverse print_vector_integer char_fromfloat code_do*times code_noop zip_swap code_yank integer_lt vector_boolean_eq genome_stackdepth code_fromziplefts noop_open_paren string_containschar string_yank char_rand zip_flush vector_boolean_rot float_swap exec_fromziprights vector_string_pushall vector_string_set vector_boolean_flush exec_noop code_size vector_boolean_stackdepth vector_integer_pushall vector_boolean_reverse integer_swap string_split vector_boolean_contains string_fromboolean return_boolean_pop vector_float_dup vector_boolean_replace integer_dup vector_boolean_nth vector_string_length string_rest zip_insert_child_fromcode float_tan string_rot string_rand exec_yank string_parse_to_chars integer_pop integer_empty vector_float_flush vector_float_yank noop_delete_prev_paren_pair print_exec zip_append_child_fromcode genome_gene_delete code_empty float_inc zip_right vector_float_length float_rand integer_dec string_contains return_fromboolean vector_float_concat vector_float_stackdepth exec_do*vector_boolean vector_integer_first genome_shove code_rand print_vector_float float_rot return_char_pop vector_string_contains vector_boolean_occurrencesof genome_empty zip_prev genome_toggle_silent vector_string_reverse zip_dup code_cons code_member exec_stackdepth float_flush boolean_and vector_boolean_butlast string_length float_cos string_frominteger exec_flush vector_string_empty exec_when vector_float_swap genome_close_dec code_insert vector_boolean_pop float_div zip_insert_right_fromcode code_fromboolean vector_boolean_take code_shove environment_begin vector_float_take boolean_invert_second_then_and code_container code_nth vector_boolean_subvec float_yank zip_up vector_boolean_emptyvector vector_boolean_replacefirst string_fromfloat vector_boolean_yankdup string_dup boolean_yankdup exec_fromzipchildren

```
;; https://github.com/lspector/Clojush/

=> (run-push '(1 2 integer_add) (make-push-state))

:exec ((1 2 integer_add))
:integer ()

:exec (1 2 integer_add)
:integer ()

:exec (2 integer_add)
:integer (1)

:exec (integer_add)
:integer (2 1)

:exec ()
:integer (3)
```

```
=> (run-push '(2 3 integer_mult 4.1 5.2 float_add
                 true false boolean_or)
               (make-push-state))

:exec ()
:integer (6)
:float (9.3)
:boolean (true)
```

In other words

- Put 2×3 on the integer stack
- Put $4.1 + 5.2$ on the float stack
- Put *true* $\vee$ *false* on the boolean stack

```
=> (run-push '(2 boolean_and 4.1 true integer_div
               false 3 5.2 boolean_or integer_mult
               float_add)
             (make-push-state))

:exec ()
:integer (6)
:float (9.3)
:boolean (true)
```

Same as before, but

- Several operations (e.g., `boolean_and`) become NOOPs
- Interleaved operations

```
=> (run-push
     '(4.0 exec_dup (3.13 float_mult) 10.0 float_div)
     (make-push-state))

:exec ((4.0 exec_dup (3.13 float_mult) 10.0 float_div))
:float ()

:exec (4.0 exec_dup (3.13 float_mult) 10.0 float_div)
:float ()

:exec (exec_dup (3.13 float_mult) 10.0 float_div)
:float (4.0)

:exec((3.13 float_mult) (3.13 float_mult) 10.0 float_div)
:float (4.0)

...

:exec ()
:float (3.91876)
```

Computes 4.0 **× 3.13 × 3.13** / 10.0

```
=> (run-push '(1 8 exec_do*range integer_mult)
             (make-push-state))

:integer (40320)
```

Computes 8! in a fairly “human” way

```
=> (run-push '(code_quote
                 (code_quote (integer_pop 1)
                             code_quote (code_dup integer_dup
                                         1 integer_sub code_do
                                         integer_mult)
                             integer_dup 2 integer_lt code_if)
                 code_dup
                 8
                 code_do)
               (make-push-state))

:code ((code_quote (integer_pop 1) code_quote (code_dup
       integer_dup 1 integer_sub code_do integer_mult)
       integer_dup 2 integer_lt code_if))
:integer (40320)
```

A less “obvious” recursive calculation of 8! achieved by code duplication

```
=> (run-push '(0 true exec_while
                  (1 integer_add true))
               (make-push-state))

:exec (1 integer_add true exec_while (1 integer_add
       true))
:integer (199)
:termination :abnormal
```

- An infinite loop
- Terminated by eval limit
- Result taken from appropriate stack(s) upon termination

```
=> (run-push '(in1 in1 float_mult 3.141592 float_mult)
             (push-item 2.5 :input (make-push-state)))

:float (19.63495)
:input (2.5)
```

Computes the area of a circle with the given radius: 3.141592 × in1 × in1

Auto-Simplification

- Loop:
  - Make it randomly simpler
  - Keep simpler if as good or better; otherwise revert

- GECCO-2014 poster: efficiently and reliably reduces the size of the evolved programs

- GECCO-2014 student paper: used as genetic operator

- GECCO-2017 GP best paper nominee: improves generalization

SUCCESS at generation 20

Successful program: (boolean_and boolean_shove exec_do*count (exec_swap (integer_empty char_yank boolean_or integer_fromboolean \space \newline) (exec_dup (char_yank char_iswhitespace string_butlast in1) string_empty boolean_frominteger tagged_275 string_substring) exec_do*times (integer_empty string_dup) string_replacechar print_string string_rot print_char integer_fromboolean string_length integer_eq string_last boolean_swap integer_yankdup) string_swap string_containschar "Wx{ " exec_stackdepth char_empty integer_swap integer_rot string_last boolean_swap integer_yankdup string_swap string_containschar "Wx{ " exec_stackdepth char_empty integer_swap integer_rot integer_fromstring string_pop string_shove char_eq char_empty integer_swap integer_rot integer_fromstring string_pop string_shove char_rot integer_stackdepth integer_min char_yankdup char_eq char_empty tagged_349 exec_yank string_rot exec_dup (boolean_eq string_removechar exec_s (exec_dup (boolean_eq exec_rot (exec_s (string_eq string_fromboolean exec_noop char_eq) () (string_butlast) integer_pop) (char_eq char_empty) (integer_swap integer_rot string_emptystring boolean_stackdepth integer_inc in1 boolean_shove boolean_swap char_isletter integer_gt integer_yankdup) exec_when (string_emptystring string_nth exec_do*range (\space integer_yankdup string_dup exec_shove (integer_swap string_removechar exec_yank string_dup exec_empty) char_eq exec_do*times (tagged_349 boolean_pop exec_when (string_removechar integer_mult integer_inc in1 boolean_shove boolean_swap char_isletter integer_gt string_butlast) integer_mult string_last string_parse_to_chars boolean_frominteger boolean_yank exec_when (string_nth exec_do*range (\space integer_yankdup string_dup exec_shove (integer_swap string_removechar exec_yank integer_yank exec_while (boolean_or)) char_isdigit boolean_swap char_isletter) integer_gt integer_yankdup integer_mult string_last string_parse_to_chars boolean_frominteger char_isletter exec_when (string_nth exec_do*range (\space integer_yankdup string_dup exec_shove (integer_swap string_removechar exec_yank integer_yank integer_mult integer_inc in1 boolean_shove boolean_swap char_isletter integer_gt string_butlast) boolean_invert_second_then_and exec_empty string_rot)) boolean_rot char_iswhitespace integer_yank string_conjchar boolean_dup) integer_add char_dup string_length integer_fromchar string_split char_isdigit boolean_swap boolean_eq char_isdigit exec_shove (boolean_invert_second_then_and string_empty string_conjchar string_shove) string_fromchar boolean_not string_stackdepth exec_y () integer_empty exec_do*range (in1 string_replace)))))) () ()))

Errors: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

Total error: 0.0

Size: 231

Auto-simplifying with starting size: 231

...

step: 5000

program: (\space \newline in1 string_replacechar print_string "Wx{ " string_last in1 string_removechar string_length)

errors: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

Total error: 0.0

Size: 11

Early Autoconstruction

- Instructions leave child code on **`child`** stack
- 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; can be leaky

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 algorithm 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 p 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" amon example, we can imagine a problem involving control of simulated wind turbine in which some test cases focu 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 in cases into a single fitness value. Th may be as simple as summing t squares, into a single error valu as implicit fitness sharing f based on population stati

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

Manuscript received Novembe 5, 2014. This material is based Foundation under Grants No findings, and conclusions are those of the authors Science Foundation.

T. Helmuth is ence, Universit math@cs.um

L. Spec Spector

J. ?

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

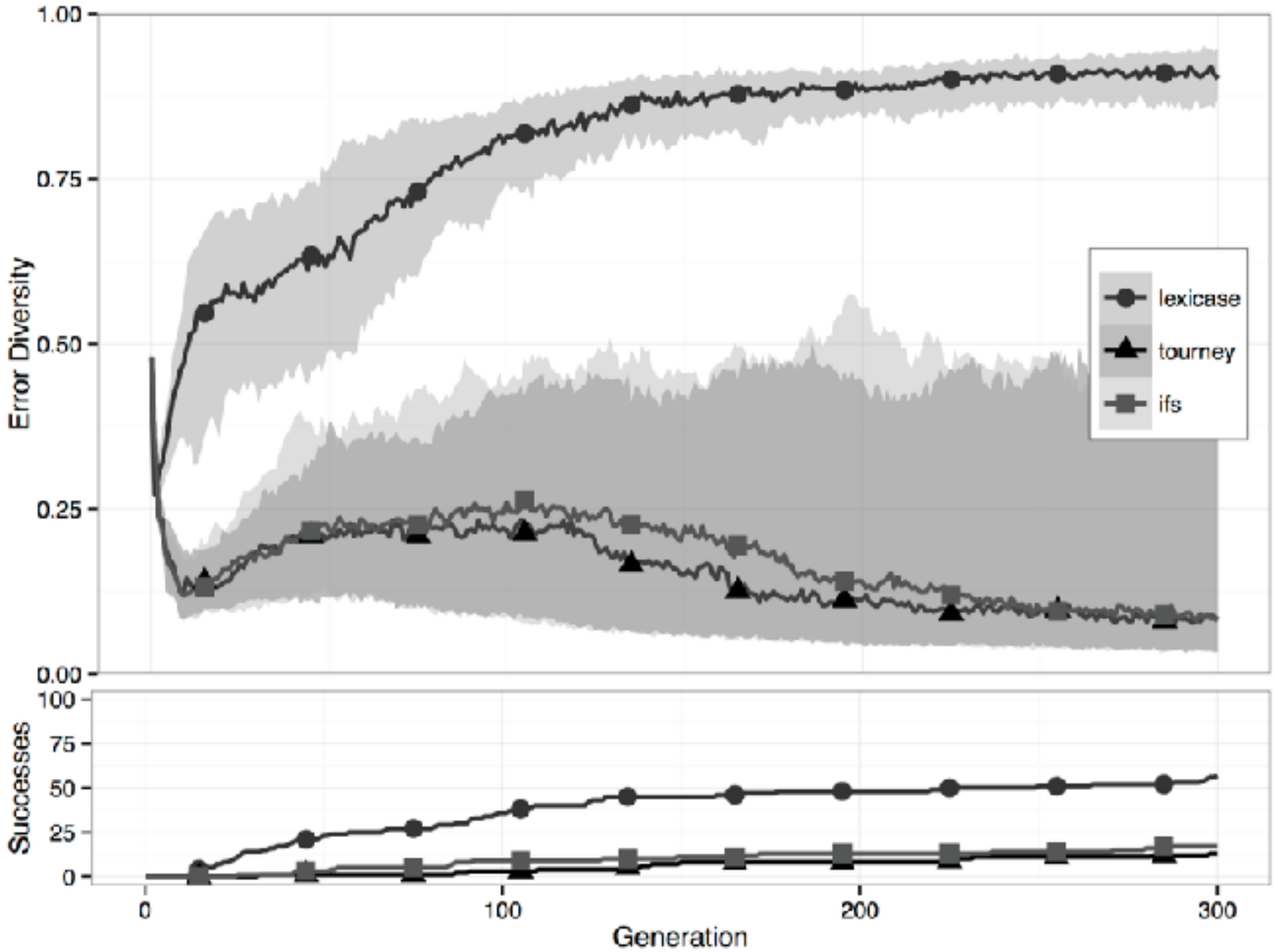


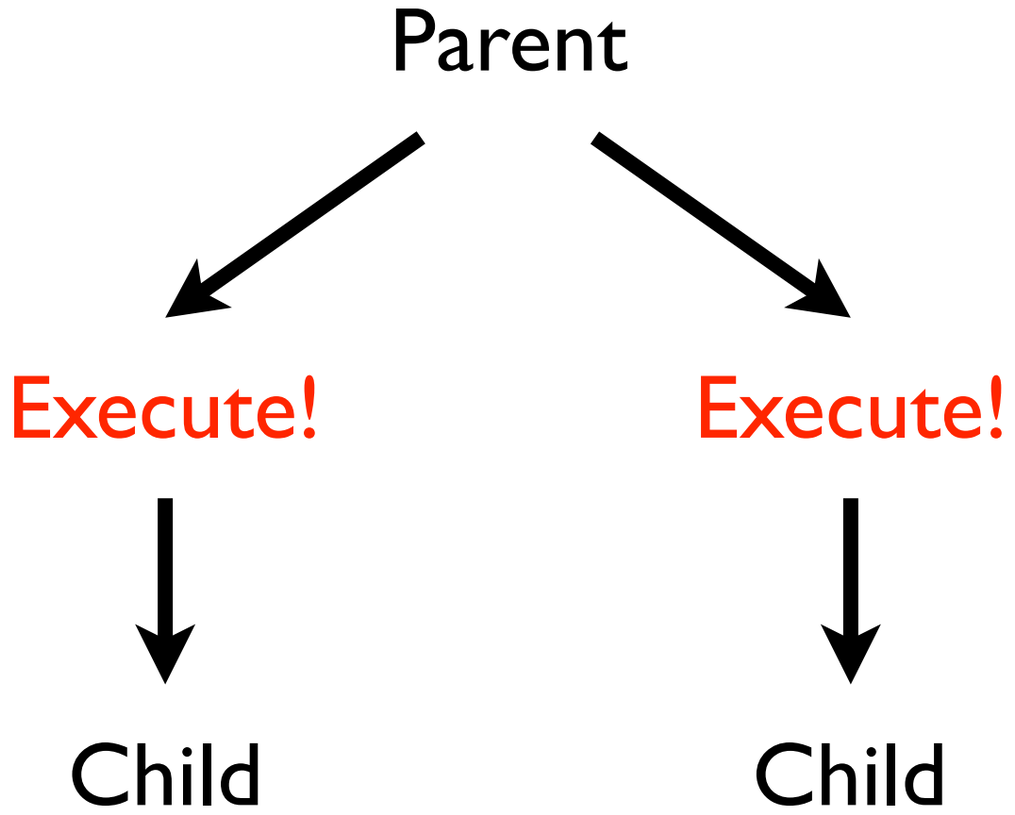
Fig. 1 Replace Space With Newline – error diversity

GPTP-2015

[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%

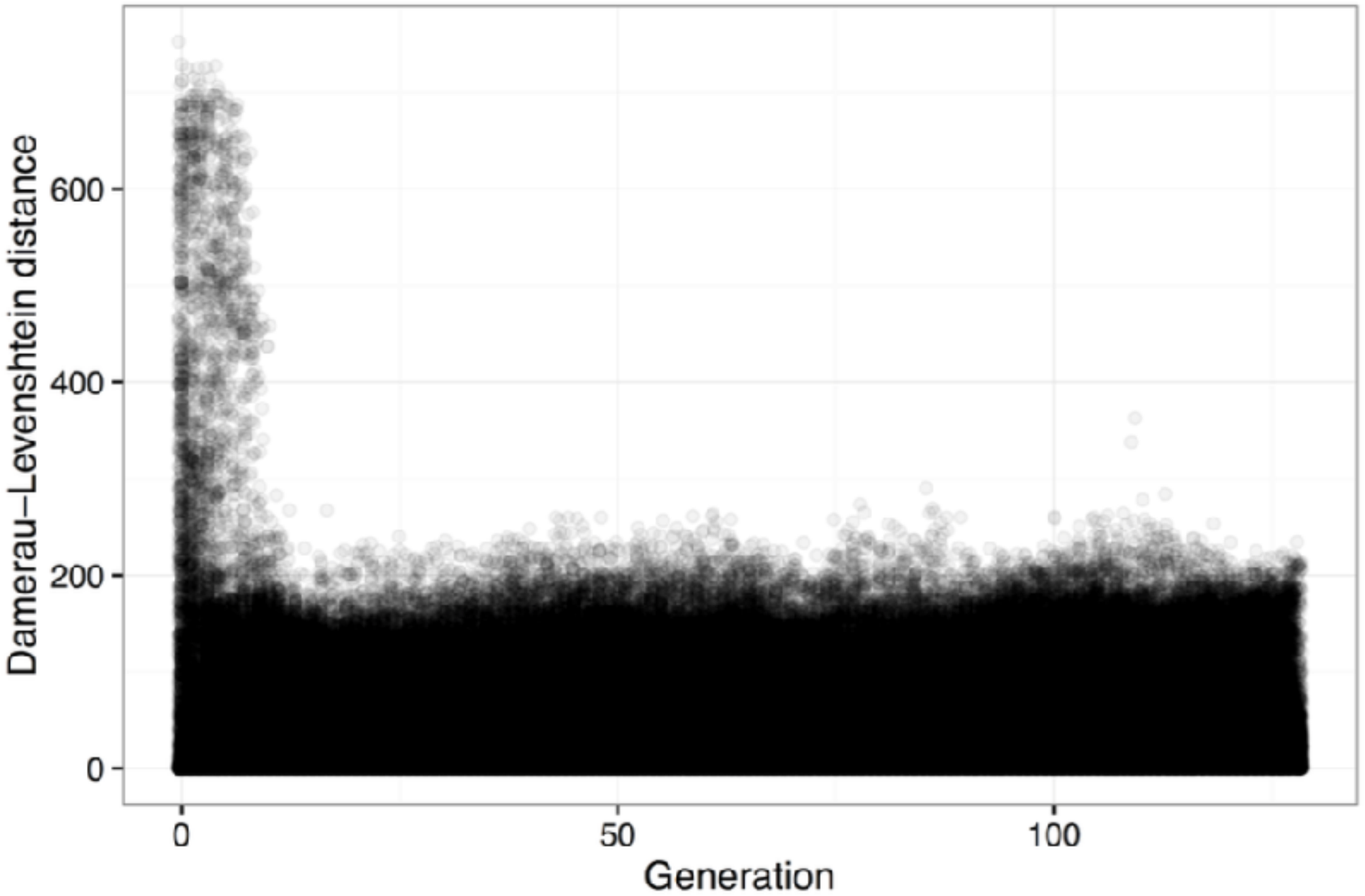


Figure 1: DL-distances between parent and child during a single non-autoconstructive run of GP on the Replace Space With Newline problem

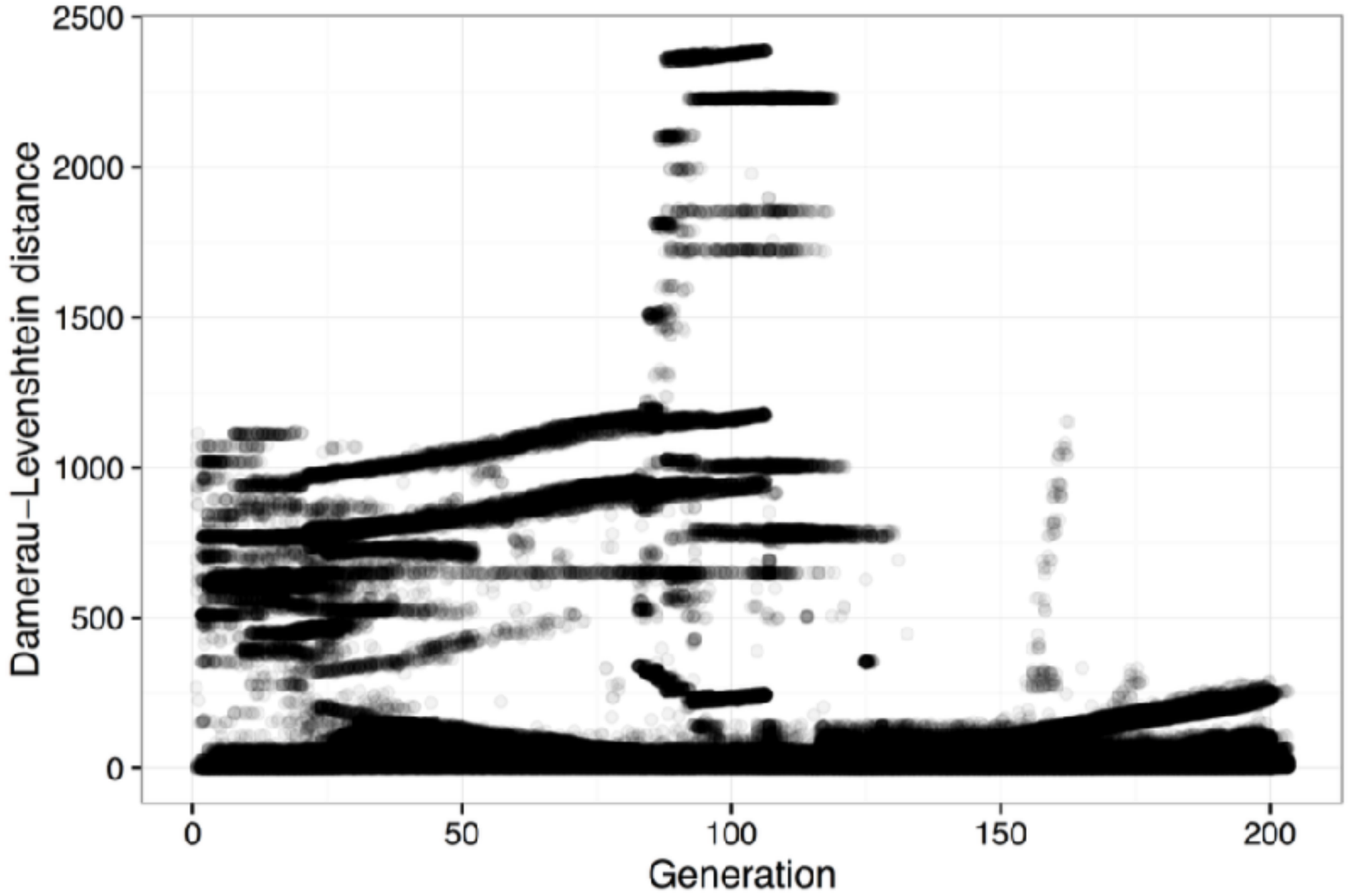


Figure 3: DL-distances between parent and child during a single autoconstructive run of GP on the Replace Space With Newline problem

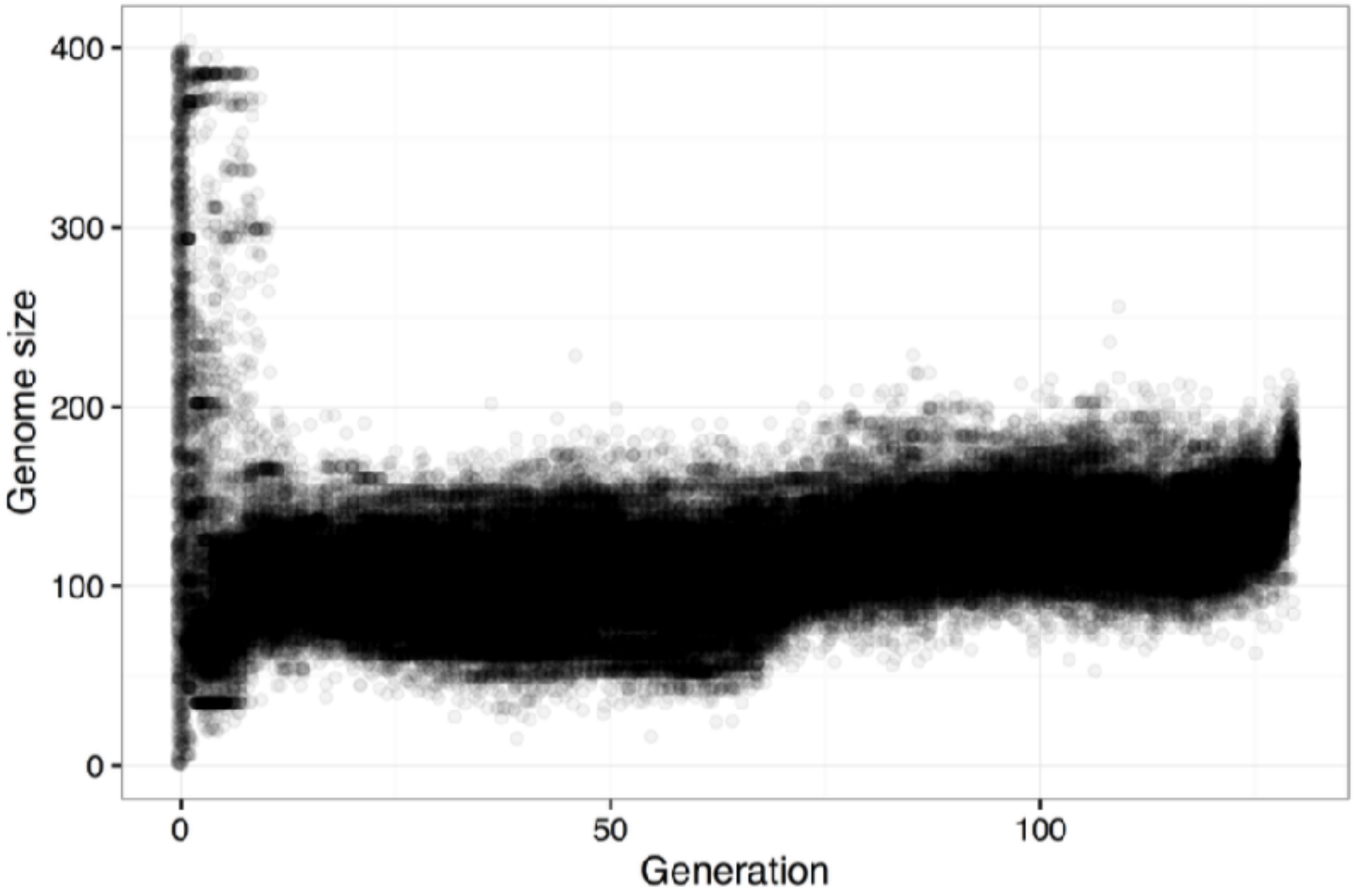


Figure 2: Genome sizes during a single non-autoconstructive run of GP on the Replace Space With Newline problem

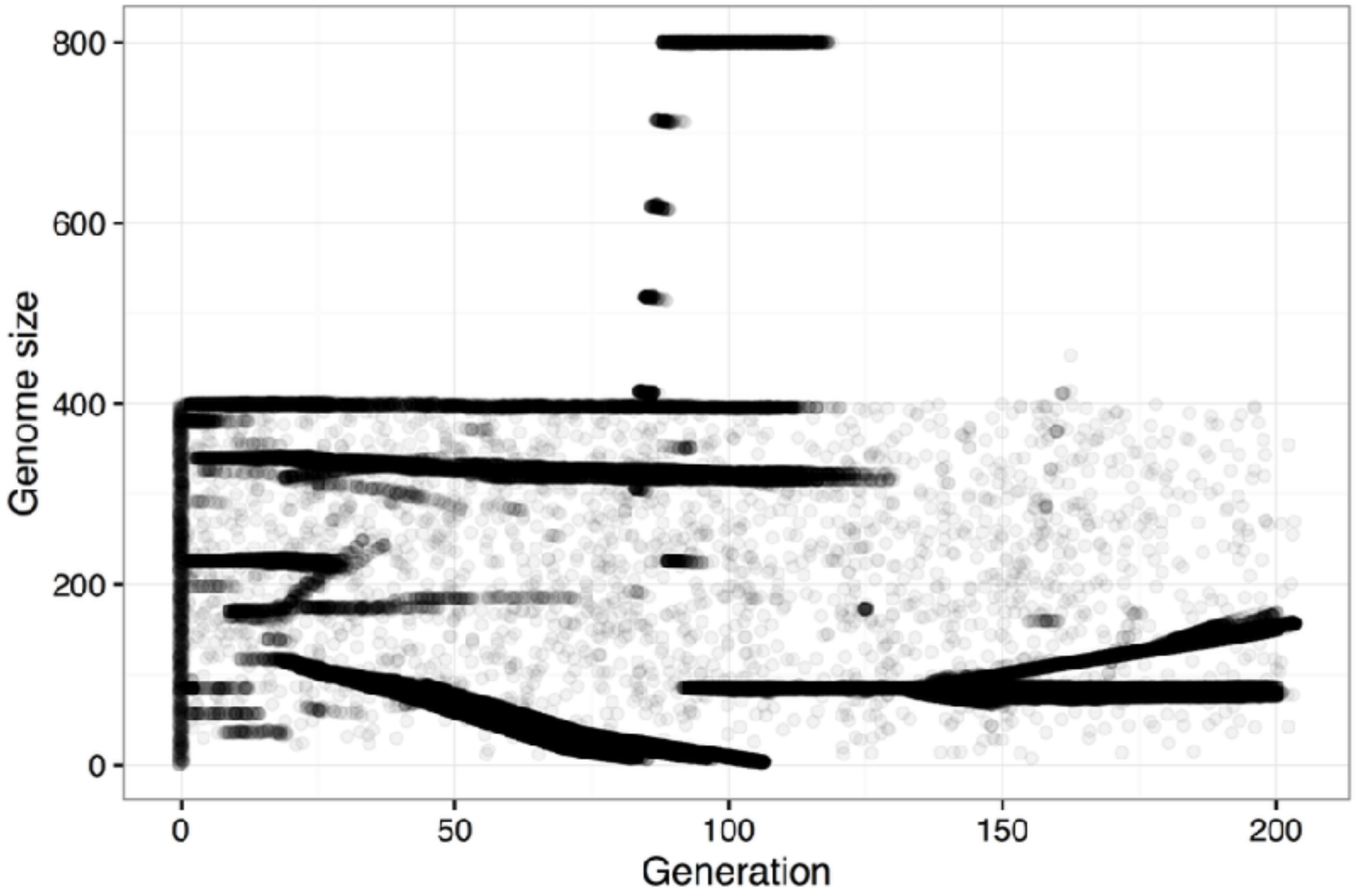
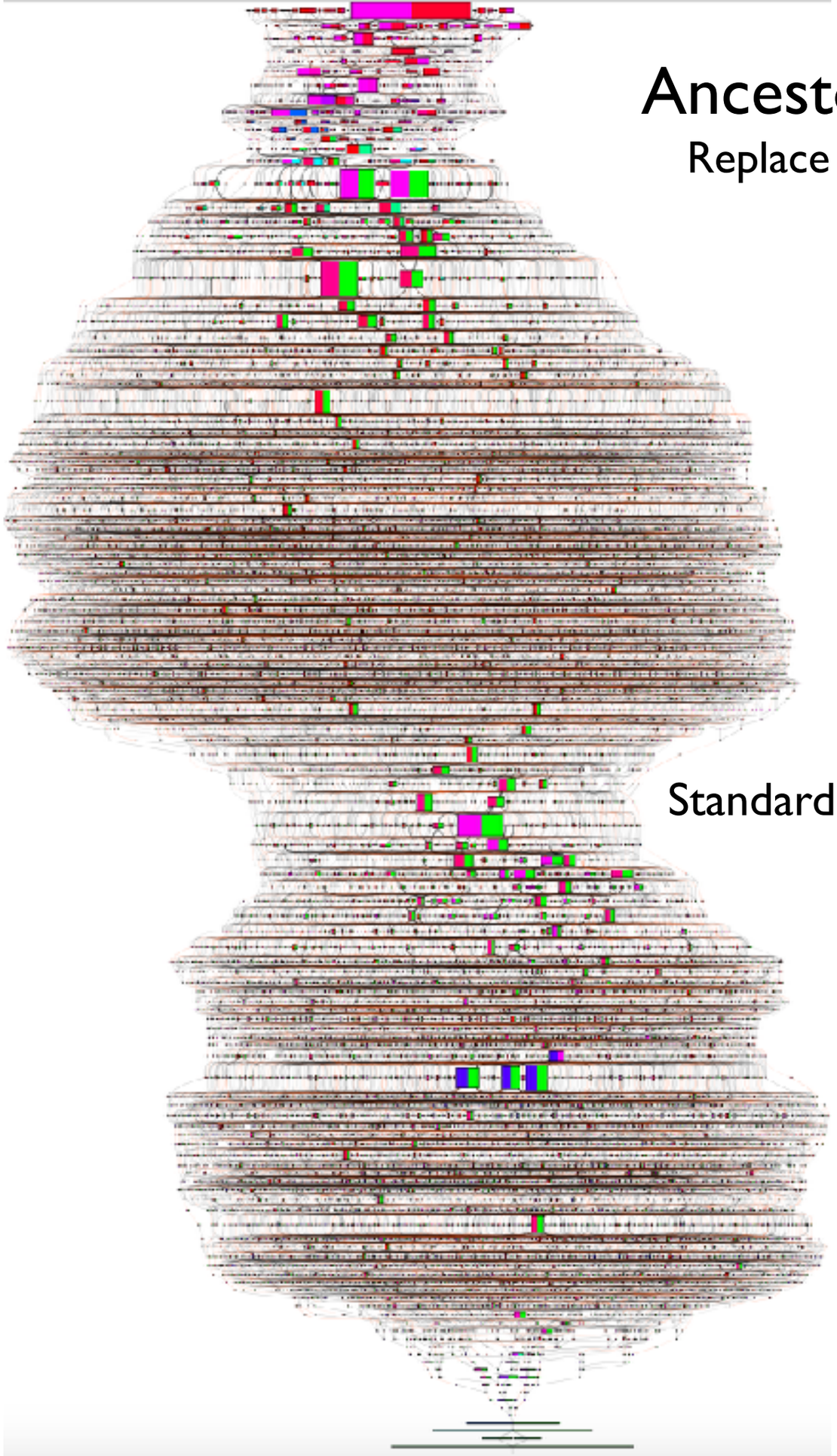


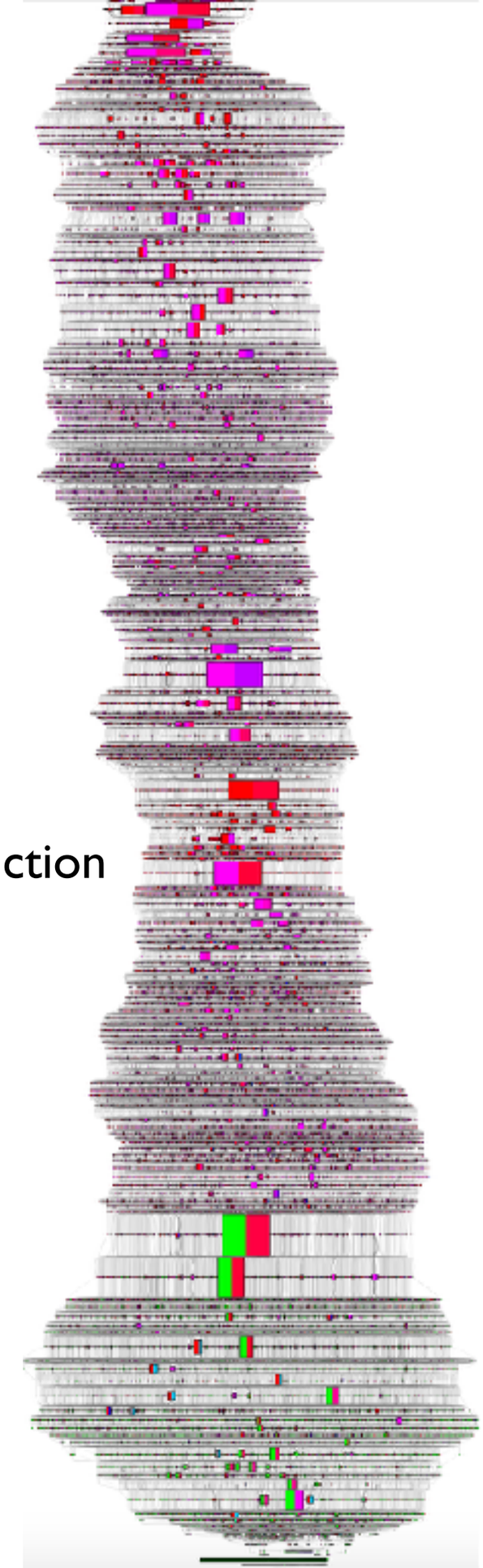
Figure 4: Genome sizes during a single autoconstructive run of GP on the Replace Space With Newline problem

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
- Decay
- 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
```

Decay

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

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 + Decay + 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; *****__update:__ solved once now, with add/delete method discovered by autoconstruciton

- Three autoconstructive solutions so far, with Uniform DSL + Decay

First Evolved Solution

- Makes children using uniform addition, with a rate (~0.0921) close to the decay 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

Broader takeaways

- Push: A flexible and powerful representation for programs that evolve
- Lexicase selection: Don't aggregate; randomly sequence

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.