The calculator problem and the evolutionary synthesis of arbitrary software

CREST Open Workshop on Genetic Programming for Software Engineering

October 14, 2013

Lee Spector
Hampshire College
Amherst, MA USA
Outline

- Arbitrary software
- Requirements and ways to meet them
- Tags, uniform variation, and lexicase selection
- The calculator problem
- Other problems and prospects
Arbitrary Software

- OS utilities
- Word processors
- Web browsers
- Accounting systems
- Image processing systems
- Everything
Arbitrary Software

• May be stateful, with multiple entry points
• May have a variety of interfaces involving a variety of types
• May require arbitrary Turing-computable functions
• Can be specified with behavioral tests
Requirements

• Represent and evolve arbitrary computable functions on arbitrary types (Push, uniform variation)
• Represent and evolve arbitrary computational architectures (modules; tags, tagged entry points)
• Drive evolution with performance tests (lexicase selection)
Evolutionary Computation

Diagram:

- Random Generation
- Assessment
  - Solution
  - Selection
  - Variation
Genetic Programming

- Evolutionary computing to produce executable computer programs
- Programs are assessed by executing them
- Automatic programming; producing software
- Potential (?): evolve software at all scales, including and surpassing the most ambitious and successful products of human software engineering
Program Representations

- Lisp-style symbolic expressions (Koza, ...).
- Purely functional/lambda expressions (Walsh, Yu, ...).
- Linear sequences of machine/byte code (Nordin et al., ...).
- Artificial assembly-like languages (Ray, Adami, ...).
- Stack-based languages (Perkis, Spector, Stoffel, Tchernev, ...).
- Graph-structured programs (Teller, Globus, ...).
- Object hierarchies (Bruce, Abbott, Schmutter, Lucas, ...)
- Fuzzy rule systems (Tunstel, Jamshidi, ...)
- Logic programs (Osborn, Charif, Lamas, Dubossarsky, ...).
- Strings, grammar-mapped to arbitrary languages (O’Neill, Ryan, ...).
Evolvability

The fact that a computation can be expressed in a formalism does not imply that a correct program can be produced in that formalism by a human programmer or by an evolutionary process.
Data/Control Structure

• Data abstraction and organization
  
  Data types, variables, name spaces, data structures, ...

• Control abstraction and organization
  
  Conditionals, loops, modules, threads, ...
Structure via GP (I)

- Specialize GP techniques to support human programming language abstractions
- Strongly typed genetic programming
- Automatically defined functions/macros
- **Architecture altering operations**
- Map from unstructured genomes to programs in languages that support abstraction (e.g. via grammars)
Structure via GP (2)

- Forget about human programming abstractions (mostly)
- Evolve programs in a minimal-syntax language that nonetheless supports a full range of data and control abstractions
- For example: orchestrate data flows via stacks, not via syntax
- Push
A programming language developed specifically for evolutionary computation, as the language in which evolving programs are expressed

Intended to maximize the evolvability of arbitrary computational processes
Push

- Stack-based postfix language with one stack per type
- Types include: integer, float, boolean, code, exec, vector, matrix, quantum gate, [add more as needed]
- Missing argument? NO-OP
- Minimal syntax:
  
  \[
  \text{program} \rightarrow \text{instruction} | \text{literal} | ( \text{program}^* )
  \]
Why Push?

• Highly expressive: data types, data structures, variables, conditionals, loops, recursion, modules, ...

• Elegant: minimal syntax and a simple, stack-based execution architecture

• Elegance simplifies a variety of things ranging from uniform variation to meta-evolution

• Evolvable

• Extensible
## Sample Push Instructions

<table>
<thead>
<tr>
<th>Category</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack manipulation instructions (all types)</td>
<td>POP, SWAP, YANK, DUP, STACKDEPTH, SHOVE, FLUSH, =</td>
</tr>
<tr>
<td>Math (INTEGER and FLOAT)</td>
<td>+, -, /, *, &gt;, &lt;, MIN, MAX</td>
</tr>
<tr>
<td>Logic (BOOLEAN)</td>
<td>AND, OR, NOT, FROMINTEGER</td>
</tr>
<tr>
<td>Code manipulation (CODE)</td>
<td>QUOTE, CAR, CDR, CONS, INSERT, LENGTH, LIST, MEMBER, NTH, EXTRACT</td>
</tr>
<tr>
<td>Control manipulation (CODE and EXEC)</td>
<td>DO*, DO<em>COUNT, DO</em>RANGE, DO*TIMES, IF</td>
</tr>
</tbody>
</table>
Push(3) Semantics

• To execute program $P$:

  1. Push $P$ onto the EXEC stack.

  2. While the EXEC stack is not empty, pop and process the top element of the EXEC stack, $E$:

     (a) If $E$ is an instruction: execute $E$ (accessing whatever stacks are required).

     (b) If $E$ is a literal: push $E$ onto the appropriate stack.

     (c) If $E$ is a list: push each element of $E$ onto the EXEC stack, in reverse order.
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )
exec code bool int float

2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR

( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )
<table>
<thead>
<tr>
<th>INTEGER.*</th>
<th>4.1</th>
<th>5.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOAT.+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRUE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FALSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOOLEAN.OR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )

<table>
<thead>
<tr>
<th>exec</th>
<th>code</th>
<th>bool</th>
<th>int</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>exec</td>
<td>code</td>
<td>bool</td>
<td>int</td>
<td>float</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>4.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOAT.+</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRUE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FALSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOOLEAN.OR</td>
<td>( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>exec</td>
<td>code</td>
<td>bool</td>
<td>int</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
</tr>
<tr>
<td>5.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOAT.+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRUE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FALSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOOLEAN.OR</td>
<td>(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>exec</td>
<td>code</td>
<td>bool</td>
<td>int</td>
<td>float</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>FLOAT.+</td>
<td>TRUE</td>
<td>FALSE</td>
<td>BOOLEAN.OR</td>
<td></td>
</tr>
<tr>
<td>( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )</td>
<td>6</td>
<td></td>
<td>5.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.1</td>
<td></td>
</tr>
<tr>
<td>exec</td>
<td>code</td>
<td>bool</td>
<td>int</td>
<td>float</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>BOOLEAN.OR</td>
<td>( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )</td>
<td>6</td>
</tr>
<tr>
<td>exec</td>
<td>code</td>
<td>bool</td>
<td>int</td>
<td>float</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>FALSE</td>
<td>BOOLEAN.OR</td>
<td>( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )</td>
<td>TRUE</td>
<td>6</td>
</tr>
<tr>
<td>exec</td>
<td>code</td>
<td>bool</td>
<td>int</td>
<td>float</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>BOOLEAN. OR</td>
<td>( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN. OR )</td>
<td>TRUE</td>
<td>6</td>
<td>9.3</td>
</tr>
<tr>
<td>exec</td>
<td>code</td>
<td>bool</td>
<td>int</td>
<td>float</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TRUE</td>
<td>6</td>
<td>9.3</td>
</tr>
</tbody>
</table>

( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )
No Time to Show How

- Push enables a trivial form of auto-simplification
- Push programs are often robust to reordering and other changes, producing a search space with high neutrality
- Push programs that modify their own code and/or the execution stack dynamically can thereby implement arbitrary control structures and several forms of modularity
Calculator Test Cases

Keys pressed => number, error flag

- Digit entry tests
- Digit entry pair tests
- Double digit float entry tests
- Single digit math tests
- Single digit incomplete math tests
- Single digit chained math tests
- Division by zero tests
Digit Entry Tests

- :zero => 0.0, false
- :one => 1.0, false
- :two => 2.0, false
- :three => 3.0, false
- ...

Digit Entry Pair Tests

- :zero :zero => 0.0, false
- :zero :one => 1.0, false
- :two :three => 23.0, false
- :nine :nine => 99.0, false
- ...

Float Entry Tests

- :zero :point :nine => 0.9, false
- :zero :point :two => 0.2 false
- :seven :point :nine => 7.9, false
- :three :point :two => 3.2, false
- ...

Single Digit Math Tests

- :zero :plus :nine :equals => 9.0, false
- :three :times :four :equals => 12.0, false
- :three :minus :nine :equals => -6.0, false
- :three :divided-by :four :equals => 0.75, false
- ...
Incomplete Math Tests

- :three :plus :four => 4.0, false
- :seven :plus => 7.0, false
- ...

Chained Math Tests

- \(3 + 9 - 5 = 7.0\), false
- \(3 \times 2 \div 8 = 0.75\), false
- \(\frac{3}{9} - 5 = -4.6666665\), false
- \(\frac{3}{9} - 5 = -4.6666665\), false
- ...
Division by Zero Tests

• zero :divided-by :zero :equals => 0.0, true
• seven :divided-by :zero :equals => 0.0, true
• three :divided-by :zero :equals => 0.0, true
• ...

Architectural Requirements

- Every key press is an entry point
- Answers (a floating point number and a boolean value) should be provided after every key press
- State must be maintained between key presses
- Stacks + tags provide an elegant way to meet these requirements
Holland’s Tags

- Initially arbitrary identifiers that come to have meaning over time
- Matches may be inexact
- Appear to be present in some form in many different kinds of complex adaptive systems
- Examples range from immune systems to armies on a battlefield
- A general tool for the support of emergent complexity
Tag-based Modules

- Include instructions that tag code (modules)
- Include instructions that recall and execute modules by closest matching tag
- If a single module has been tagged then all tag references will recall modules
- The number of tagged modules can grow incrementally over evolutionary time
- Expressive and evolvable
Calculator Architecture

- Run program once to tag modules
- Clear stacks
- For each pressed key, execute the module that best matches the corresponding tag, maintaining stacks across key presses
- The top of the float stack is the number output; the top of the boolean stack is the error flag output
And?

With Push and the tagged-entry-point architecture we can run GP on the calculator problem.

And it fails miserably:

- Large programs are required
- Must allow growth without bloating
- Must allow arbitrary recombination
Uniform Variation

• All genetic material that a child inherits should be \(\approx\) likely to be mutated

• Parts of both parents should be \(\approx\) likely to appear in children (at least if they are \(\approx\) in size), and to appear in a range of combinations
Why Uniformity?

• No hiding from mutation
• All parts of parents subject to variation and recombination
• Biological genetic variation, while not fully uniform, has uniformity properties that prevent some of the problems we see in GP; e.g. just having more genes doesn’t generally “protect” any of them
Prior Work

• Point mutations or “uniform crossovers” that replace/swap nodes but only in restricted ways; cannot change structure, has depth biases (McKay et al, 1995; Page et al, 1998; Poli and Langdon, 1998; Poli and Page, 2000; Semenkin and Semenkina, 2012)

• Uniform mutation via size-based numbers of tree replacements; depth biases, little demonstrated benefit (McKay et al, 1995; Van Belle and Ackley, 2002)
ULTRA

- Achieve uniformity by treating genomes as linear sequences, even if they are hierarchically structured
- Repair after transform to ensure structural validity
The ULTRA Operator

- **U**niform **L**inear **T**ransformation with **R**epair and **A**lternation

- Linearize 2 parents, treating “(” and “)” as ordinary tokens

- Start at the beginning of one parent and copy tokens to the child, switching parents stochastically (according to the *alternation rate*, and subject to an *alignment deviation*)

- Post-process with uniform mutation (according to a *mutation rate*) and repair
Parents:

( a b ( c ( d ) ) e ( f g ) )
( 1 ( 2 ( 3 4 ) 5 ) 6 )

Result of alternation:

( a b 2 ( 3 4 d ) 6 )

Result of repair:

( a ( b 2 ( 3 4 d ) 6 ) )
ULTRA on the bioavailability problem

Fig. 1 Results from the bioavailability problem. We conducted 100 runs for each choice of operators. The RMSE of the best individuals on the training fitness cases (left) and on the test fitness cases (right). In each plot, subtree replacement 81/9/10 is plotted first, followed by subtree replacement 45/45/10 and then ULTRA. In each box plot, the box stretches from the first quartile to the third quartile with a line for the median in the middle. The whiskers extend to the furthest value within 1.5 times the inter-quartile range. Points beyond the whiskers are outliers, plotted as points. Note that in the right plot, 8 outliers on the 81/9/10 set, 7 outliers on the 45/45/10 set, and 3 outliers on the ULTRA set fell outside the of the visible plot.

Fig. 2 Program sizes for the bioavailability problem.

Table 3 Results on the Pagie-1 problem. We conducted 100 runs for each choice of operators. MBF is the mean best fitness of the run. Note that the reported fitnesses are the mean errors over test cases, not the summed errors.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Successes</th>
<th>MBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtree Replacement</td>
<td>0</td>
<td>0.363</td>
</tr>
<tr>
<td>80/10/10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtree Replacement</td>
<td>0</td>
<td>0.319</td>
</tr>
<tr>
<td>45/45/10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULTRA</td>
<td>15</td>
<td>0.036</td>
</tr>
</tbody>
</table>

Table 3 presents the results of our experiments on the Pagie-1 problem. PushGP using ULTRA found perfect solutions in 15 out of 100 runs, whereas runs with subtree replacement found none with either parameter setting. The difference in MBF between subtree replacement 80/10/10 and ULTRA, as well as subtree replacement 45/45/10 and ULTRA, is statistically significant based on an unpaired t-test at $p = 0.01$.

Note that the results for subtree replacement 45/45/10 are only over 98 runs, with data from 2 runs yet to come.

The mean program sizes in our Pagie-1 experiments are given in Figure 3. Runs using subtree replacement experienced quick code growth, reaching mean sizes near the maximum program size of 500 within the first 50 generations. After this point, it is difficult for the genetic operators to make changes to large programs without exceeding the program size limit. On the other hand, the mean program sizes of ULTRA runs quickly drop to around size 50, and then climb to approach 100. In these runs, it is unlikely that many genetic operations will exceed the size limit.
With Push, the tagged-entry-point architecture, and ULTRA... we still fail. But not quite as miserably.

Issues:

• Different test cases require qualitatively different modes of response

• Numbers of cases of different types have an undue influence

• Average performance across cases does not guide search appropriately
Lexicase Selection

- Each parent is selected by filtering the entire population, one one case at a time (in random order), keeping only the elite at each stage

- Useful for “modal” problems, which require qualitatively different responses to different inputs

- Useful for “uncompromising” problems, in which solutions must be optimal on each case

- All comparisons are “within case,” so may be useful whenever cases are non-comparable
Lexicase Selection

Initialize:

Candidates = the entire population

Cases = a list of all of the test cases in random order

Loop:

Candidates = the subset of Candidates with exactly the best performance of any current candidate for the first case in Cases

If Candidates or Cases contains just a single element then return a randomly selected individual from Candidates

Otherwise remove the first case from Cases and go to Loop
Finite Algebras

\[
\begin{array}{c|ccc}
A_1 \ast & 0 & 1 & 2 \\
\hline
0 & 2 & 1 & 2 \\
1 & 1 & 0 & 0 \\
2 & 0 & 0 & 1 \\
\end{array}
\]
A1 Mal’cev Term

<table>
<thead>
<tr>
<th>Selection</th>
<th>Successes</th>
<th>CE</th>
<th>MBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tournament Size 2</td>
<td>35</td>
<td>532,000</td>
<td>0.75</td>
</tr>
<tr>
<td>Tournament Size 3</td>
<td>43</td>
<td>420,000</td>
<td>0.70</td>
</tr>
<tr>
<td>Tournament Size 4</td>
<td>31</td>
<td>440,000</td>
<td>0.75</td>
</tr>
<tr>
<td>Tournament Size 5</td>
<td>22</td>
<td>616,000</td>
<td>0.77</td>
</tr>
<tr>
<td>Tournament Size 6</td>
<td>25</td>
<td>750,000</td>
<td>0.90</td>
</tr>
<tr>
<td>Tournament Size 7</td>
<td>23</td>
<td>403,000</td>
<td>0.92</td>
</tr>
<tr>
<td>Tournament Size 8</td>
<td>26</td>
<td>464,000</td>
<td>0.94</td>
</tr>
<tr>
<td>Tournament Size 9</td>
<td>21</td>
<td>550,000</td>
<td>1.06</td>
</tr>
<tr>
<td>Lexicase</td>
<td>94</td>
<td>90,000</td>
<td>0.05</td>
</tr>
</tbody>
</table>
Digital Multiplier

• Evolve a digital circuit to multiply two binary numbers

• $n$-bit digital multiplier: $2 \times n$ bits $\rightarrow 2n$ bits

• Multiple outputs

• Scalable

• Recommended as a GP benchmark problem (McDermott, et al 2012, White et al 2013)
3-bit Digital Multiplier

<table>
<thead>
<tr>
<th>Boolean Stack</th>
<th>and, or, xor, invert_first_then_and, dup, swap, rot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input / Output</td>
<td>in0, ..., in2n, out0, ..., out2n</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Selection</th>
<th>Successes</th>
<th>MBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tournament Size 7</td>
<td>0</td>
<td>0.24</td>
</tr>
<tr>
<td>Lexicase</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

Reported fitneses are the summed errors over test cases. The mean of those errors is the mean best fitness of the run. In our experiments, some test cases will have a much larger impact on the evolved solution than others. Beyond the input and output instructions, we use the boolean stack manipulation instructions found in the top row of Table VI. The boolean stack instructions found in the top row of Table VI are often used in PushGP. The 3-bit digital multiplier problem uses each n-bit number as output. This specific output instruction is never called within the program, so the evolved solution should essentially do the same thing for each test case. Consideration regardless of the magnitude of the errors, we expect it to perform better than tournament selection for this version uses symbolic regression problem with one input and one output, whereas tournament selection never found a perfect solution. Find perfect solutions in every run, whereas tournament selection never found a perfect solution.

### Table IV

<table>
<thead>
<tr>
<th>Tournament Size</th>
<th>Selection</th>
<th>Successes</th>
<th>MBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Lexicase</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Tournament</td>
<td>9</td>
<td>1.45</td>
</tr>
<tr>
<td>7</td>
<td>Tournament</td>
<td>3</td>
<td>1.59</td>
</tr>
<tr>
<td>8</td>
<td>Tournament</td>
<td>26</td>
<td>0.94</td>
</tr>
<tr>
<td>9</td>
<td>Tournament</td>
<td>10</td>
<td>1.60</td>
</tr>
</tbody>
</table>

### Table VI

<table>
<thead>
<tr>
<th>Boolean Stack</th>
<th>and, or, xor, invert_first_then_and, dup, swap, rot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>Lexicase</td>
</tr>
<tr>
<td>Successes</td>
<td>100</td>
</tr>
<tr>
<td>MBF</td>
<td>0</td>
</tr>
<tr>
<td>Input / Output</td>
<td>in0, ..., in2n, out0, ..., out2n</td>
</tr>
</tbody>
</table>

### Table VII

<table>
<thead>
<tr>
<th>Tournament Size</th>
<th>Selection</th>
<th>Successes</th>
<th>MBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Lexicase</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Tournament</td>
<td>9</td>
<td>1.45</td>
</tr>
<tr>
<td>7</td>
<td>Tournament</td>
<td>3</td>
<td>1.59</td>
</tr>
<tr>
<td>8</td>
<td>Tournament</td>
<td>26</td>
<td>0.94</td>
</tr>
<tr>
<td>9</td>
<td>Tournament</td>
<td>10</td>
<td>1.60</td>
</tr>
</tbody>
</table>

The factorial symbolic regression problem is an integer multiplier. The 3-bit digital multiplier problem uses each n-bit number as output. This specific output instruction is never called within the program, so the evolved solution should essentially do the same thing for each test case. Consideration regardless of the magnitude of the errors, we expect it to perform better than tournament selection for this version uses symbolic regression problem with one input and one output, whereas tournament selection never found a perfect solution. Find perfect solutions in every run, whereas tournament selection never found a perfect solution.
## Factorial

<table>
<thead>
<tr>
<th>Boolean Stack</th>
<th>and, dup, eq, frominteger, not, or, pop, rot, swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Stack</td>
<td>add, div, dup, eq, fromBoolean, greaterThan, lessThan, mod, mult, pop, rot, sub, swap</td>
</tr>
<tr>
<td>Exec Stack</td>
<td>dup, eq, if, noop, pop, rot, swap, when, k, s, y</td>
</tr>
<tr>
<td>Input</td>
<td>in</td>
</tr>
<tr>
<td>Constants</td>
<td>0, 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Selection</th>
<th>Successes</th>
<th>MBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tournament Size 7</td>
<td>0</td>
<td>74,545</td>
</tr>
<tr>
<td>Lexicase</td>
<td>61</td>
<td>28,980</td>
</tr>
</tbody>
</table>
With Push, the tagged-entry-point architecture, ULTRA, and lexicase selection... we succeed!*  
*On some reasonably large sets of tests (not all shown above, yet).  
*But without generalizing.
Continuing Work

• Generative tests for selection and validation
• Refinements to tagging mechanisms, ULTRA, and lexicase selection
• Work on other program synthesis problems:
  • Kata bowling
  • The UNIX wc program
  • CS101 problems
• Insights from non-evolutionary program synthesis work
Conclusions

• Evolutionary synthesis of arbitrary software is hard!

• But we can learn a lot from trying to do it, both for software synthesis and for other GP applications (including others in software engineering, I suspect)

• Push, tags, tagged-entry points, uniform variation methods, and lexicase selection have all demonstrated promise
Thanks

• Thomas Helmuth, Emma Tosch, Kyle Harrington, Kwaku Yeboah Antwi, Jamie Matheson, Daniel Homer, Omri Bernstein, Jake Wisdom, Josiah Erikson

• USA National Science Foundation grants
  Grants No. 1017817 and 1129139.

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.
Genetic Programming and Evolvable Machines

Main editor: L. Spector
ISSN: 1389-2576 (print version)
ISSN: 1573-7632 (electronic version)
Journal no. 10710

Free Preview

2012 Impact Factor 1.333