Expressive Genetic Programming

2012 Genetic and Evolutionary Computation Conference (GECCO-2012)

Lee Spector
School of Cognitive Science
Hampshire College
Amherst, MA 01002 USA
lspector@hampshire.edu
http://hampshire.edu/lspector

Copyright is held by the author/owner(s).
GECCO'12 Companion, July 7–11, 2012, Philadelphia, PA, USA.
ACM 978-1-4503-1178-6/12/07.

Tutorial Description (1)

The language in which evolving programs are expressed can have significant impacts on the problem-solving capabilities of a genetic programming system. These impacts stem both from the absolute computational power of the languages that are used, as elucidated by formal language theory, and from the ease with which various computational structures can be produced by random code generation and by the action of genetic operators. Highly expressive languages can facilitate the evolution of programs for any computable function using, when appropriate, multiple data types, evolved subroutines, evolved control structures, evolved data structures, and evolved modular program and data architectures. In some cases expressive languages can even support the evolution of programs that express methods for their own reproduction and variation (and hence for the evolution of their offspring).

Tutorial Description (2)

This tutorial will begin with a comparative survey of approaches to the evolution of programs in expressive programming languages ranging from machine code to graphical and grammatical representations. Within this context it will then provide a detailed introduction to the Push programming language, which was designed specifically for expressiveness and specifically for use in genetic programming systems. Push programs are syntactically unconstrained but can nonetheless make use of multiple data types and express arbitrary control structures, supporting the evolution of complex, modular programs in a particularly simple and flexible way. The Push language will be described and ten years of Push-based research, including the production of human-competitive results, will be briefly surveyed. The tutorial will conclude with a discussion of recent enhancements to Push that are intended to support the evolution of complex and robust software systems.

Instructor

Lee Spector is a Professor of Computer Science in the School of Cognitive Science at Hampshire College in Amherst, Massachusetts, and an adjunct professor in the Department of Computer Science at the University of Massachusetts, Amherst. He received a B.A. in Philosophy from Oberlin College in 1984 and a Ph.D. from the Department of Computer Science at the University of Maryland in 1992. His areas of teaching and research include genetic and evolutionary computation, quantum computation, and a variety of intersections between computer science, cognitive science, evolutionary biology, and the arts. He is the Editor-in-Chief of the journal Genetic Programming and Evolvable Machines (published by Springer) and a member of the editorial board of Evolutionary Computation (published by MIT Press). He is also a member of the SIGEVO executive committee and he was named a Fellow of the International Society for Genetic and Evolutionary Computation.
Course Agenda

- Genetic Programming refresher
- Why evolve programs in expressive languages?
- Expressivity and evolvability
- Expressive trees, bits, graphs, grammars, stacks
- Push
- Expressing the future

Evolution, the Designer

“Darwinian evolution is itself a designer worthy of significant respect, if not religious devotion.” Boston Globe OpEd, Aug 29, 2005

Evolutionary Computation

- Expression of the future

Genetic Programming (GP)

- 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, ...).

Mutating Lisp

\[
\begin{align*}
\text{Parent 1:} & \quad (+ \ (* \ X \ Y) \\
& \quad (+ \ 4 \ (- \ Z \ 23)) \\
\text{Parent 2:} & \quad (- \ (* \ 17 \ (+ \ 2 \ X)) \\
& \quad (* \ (- \ (* \ 2 \ Z) \ 1) \\
& \quad (+ \ 14 \ (/ \ Y \ X))) \\
\text{Child 1:} & \quad (+ \ (- \ (* \ 2 \ Z) \ 1) \\
& \quad (+ \ 4 \ (- \ Z \ 23))) \\
\text{Child 2:} & \quad (- \ (* \ 17 \ (+ \ 2 \ X)) \\
& \quad (* \ (* \ X \ Y) \\
& \quad (+ \ 14 \ (/ \ Y \ X))))
\end{align*}
\]

Recombining Lisp

Given a set of data points, evolve a program that produces \( y \) from \( x \).

Primordial ooze: +, -, *, %, x, 0.1

Fitness = error (smaller is better)

Symbolic Regression
GP Parameters

Maximum number of Generations: 51
Size of Population: 1000
Maximum depth of new individuals: 6
Maximum depth of new subtrees for mutants: 4
Maximum depth of individuals after crossover: 17
Fitness-proportionate reproduction fraction: 0.1
Crossover at any point fraction: 0.3
Crossover at function points fraction: 0.5
Selection method: FITNESS-PROPORTIONATE
Generation method: RAMPED-HALF-AND-HALF
Randomizer seed: 1.2

Evolving  \( y = x^3 - 0.2 \)

Best Program, Gen 0

\[ (- \left( \% \left( * \ 0.1 \ \left( * \ X \ X \right) \right) \right) \]

Best Program, Gen 5

\[ (- \left( * \ \left( \% \ \left( * \ 0.1 \ \left( * \ X \ X \right) \right) \right) \right) \]
Expressiveness

• Turing machine tables
• Lambda calculus expressions
• Register machine programs
• Partial recursive functions
• etc.

Pragmatics

The fact that a computation can be expressed in a formalism does not imply that a correct expression can be produced in that formalism by a human programmer or by an evolutionary process.
Tricks

- Cars, airplanes, and other complex engineered artifacts...
- Evolved biological organisms...
- Large-scale software systems...

... are each composed of millions of specialized parts, chosen, in each case, from a portfolio of domain-specialized components and processes.

Code Tricks

- Data abstraction and organization
  Data types, variables, name spaces, data structures, ...
- Control abstraction and organization
  Conditionals, loops, modules, threads, ...

Tricks via GP (1)

- Specialize GP techniques to directly support “code trick” syntax of human programming languages
- Strongly typed genetic programming
- Automatically defined functions
- Automatically defined macros
- Architecture altering operations

Tricks via GP (2)

- Specialize GP techniques to *indirectly* support “code trick” syntax from human programming languages
- Repair
- Genotype/phenotype mapping
- Grammars
Tricks via GP (3)

- Develop new program encodings, represented most generally as graphs
- Develop analogs of code tricks for these representations
- Specialize GP techniques to directly or indirectly support “code trick” syntax for these new program encodings

Tricks via GP (4)

- Evolve programs in a minimal-syntax language that nonetheless supports a full range of “code tricks”
- For example: orchestrate data flows via stacks, not via syntax
- Push

Modularity is Everywhere

![Modularity Diagram]

- Pervasive and widely acknowledged to be essential
- Modules may be functions, procedures, methods, classes, data structures, interfaces, etc.
- Modularity measures include coupling, cohesion, encapsulation, composability, etc.
**Modules via GP**

- Automatically-defined functions
- Automatically-defined macros
- Architecture-altering operations
- Module acquisition/encapsulation systems
- Grammars for languages with modules
- Instructions that build/execute modules

**Evolving Modular Programs**

With “automatically defined functions”

- All programs in the population have the same, pre-specified architecture
- Genetic operators respect that architecture
- Significant implementation costs
- Significant pre-specification
- Architecture-altering operations: more power and higher costs

**ADMs**

- Macros implement control structures
- ADMs can be implemented via small tweaks to any system that supports ADFs
- Similar pros and cons to ADFs, but provide additional expressive power

**Control Structures (1)**

Multiple evaluation

```
(defmacro do-twice (code)
  `(progn ,code ,code))
```

```
(do-twice (incf x))
```
Control Structures (2)

Conditional evaluation

(defmacro numeric-if (exp neg zero pos)
  `(if (< ,exp 0)
     ,neg
     (if (< 0 ,exp) ,pos ,zero)))

(numeric-if (foo) (bar) (baz) (bix))

Push

- Stack-based postfix language with one stack per type
- Types include: integer, float, Boolean, name, code, exec, vector, matrix, quantum gate, [add more as needed]
- Missing argument? NOOP
- Minimal syntax:
  
  program → instruction | literal | ( program* )

Why Push?

- Highly expressive: data types, data structures, variables, conditionals, loops, recursion, modules, ...
- Elegant: minimal syntax and a simple, stack-based execution architecture
- Evolvable
- Extensible
- Supports several forms of meta-evolution

Sample Push Instructions

| Stack manipulation instructions (all types) | POP, SWAP, YANK, DUP, STACKDEPTH, SHOVE, FLUSH, = |
| Math (INTEGER and FLOAT) | +, -, /, *, >, <, MIN, MAX |
| Logic (BOOLEAN) | AND, OR, NOT, FROMINTEGER |
| Code manipulation (CODE) | QUOTE, CAR, CDR, CONS, INSERT, LENGTH, LIST, MEMBER, NTH, EXTRACT |
| Control manipulation (CODE and EXEC) | DO*, DO*COUNT, DO*RANGE, DO*TIMES, IF |
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.
<table>
<thead>
<tr>
<th>exec</th>
<th>code</th>
<th>bool</th>
<th>int</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>BOOLEAN.OR</td>
<td>(2 INTEGER + 4 5 2 FLOAT + TRUE FALSE BOOLEAN.OR)</td>
<td>6</td>
</tr>
<tr>
<td>FALSE</td>
<td>BOOLEAN.OR</td>
<td>(2 INTEGER + 4 5 2 FLOAT + TRUE FALSE BOOLEAN.OR)</td>
<td>TRUE</td>
<td>6</td>
</tr>
</tbody>
</table>
Same Results

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

( 2 BOOLEAN.AND 4.1 TRUE INTEGER./ FALSE 3 5.2 BOOLEAN.OR INTEGER.* FLOAT.+ )
<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>3.14</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.14</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.14</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.0</td>
<td></td>
</tr>
</tbody>
</table>

The code snippet appears to be a fragment of a programming language, possibly LISP or a similar dialect. It contains conditional statements (`CODE.IF`) and comparison operators (e.g., `FLOAT.>`). The values `3.14` and `4.0` suggest it might be dealing with floating-point numbers and comparisons.
<table>
<thead>
<tr>
<th>exec</th>
<th>code</th>
<th>bool</th>
<th>int</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOAT.&gt; (CODE:QUOTE FLOAT.*)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODE:IF (CODE:QUOTE FLOAT.*) FLOAT.&gt; 5.0 IN IN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODE:CDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

exec code bool int float

<table>
<thead>
<tr>
<th>exec</th>
<th>code</th>
<th>bool</th>
<th>int</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOAT.* (CODE:QUOTE FLOAT.*)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODE:IF (CODE:QUOTE FLOAT.*) FLOAT.&gt; 5.0 IN IN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODE:CDR</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>3.14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

exec code bool int float

<table>
<thead>
<tr>
<th>exec</th>
<th>code</th>
<th>bool</th>
<th>int</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE:QUOTE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOAT.* (CODE:QUOTE FLOAT.*)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODE:IF (CODE:QUOTE FLOAT.*) FLOAT.&gt; 5.0 IN IN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODE:CDR</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>3.14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

exec code bool int float

<table>
<thead>
<tr>
<th>exec</th>
<th>code</th>
<th>bool</th>
<th>int</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE:QUOTE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOAT.* (CODE:QUOTE FLOAT.*)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODE:IF (CODE:QUOTE FLOAT.*) FLOAT.&gt; 5.0 IN IN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODE:CDR</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>3.14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

exec code bool int float
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)

IN=4.0
The Odd Problem

- Integer input
- Boolean output
- Was the input odd?
- \((\text{code.nth} \text{ code.atom})\)

Combinators

- **Standard \(K, S, \text{ and } Y\) combinators:**
  - \(\text{EXEC}.K\) removes the second item from the \(\text{EXEC}\) stack.
  - \(\text{EXEC}.S\) pops three items (call them A, B, and C) and then pushes \((B \ C), C, \text{ and then } A\).
  - \(\text{EXEC}.Y\) inserts \((\text{EXEC}.Y \ T)\) under the top item \(T\).

- A \(Y\)-based “while” loop:
  \[
  \begin{align*}
  \text{(EXEC.Y} \\
  \text{( <BODY/CONDITION> EXEC.IF} \\
  \text{ ( ) EXEC.POP ) )}
  \end{align*}
  \]
Iterators

CODE.DO*TIMES, CODE.DO*COUNT, CODE.DO*RANGE
EXEC.DO*TIMES, EXEC.DO*COUNT, EXEC.DO*RANGE

Additional forms of iteration are supported through code manipulation (e.g. via
CODE.DUP CODE.APPEND CODE.DO)

Named Subroutines

( TIMES2 EXEC.DEFINE ( 2 INTEGER.* ) )

Auto-simplification

Loop:

Make it randomly simpler
If it's as good or better: keep it
Otherwise: revert

Problems Solved by PushGP in the GECCO-2005 Paper on Push3

- Reversing a list
- Factorial (many algorithms)
- Fibonacci (many algorithms)
- Parity (any size input)
- Exponentiation
- Sorting
Autoconstructive Evolution

- Individuals make their own children
- Agents thereby control their own mutation rates, sexuality, and reproductive timing
- The machinery of reproduction and diversification (i.e., the machinery of evolution) evolves
- Radical self-adaptation

Related Work

- MetaGP: but (1) programs and reproductive strategies dissociated and (2) generally restricted reproductive strategies
- ALife systems such as Tierra, Avida, SeMar: but (1) hand-crafted ancestors, (2) reliance on cosmic ray mutation, and (3) weak problem solving
- Evolved self-reproduction: but generally exact reproduction, non-improving (exception: Koza, but very limited tools for problem solving and for construction of offspring)
Pushpop

- A soup of evolving Push programs
- Reproductive procedures emerge ex nihilo:
  - No hand-designed “ancestor”
  - Children constructed by any computable process
  - No externally applied mutation procedure or rate
  - Exact clones are prohibited, but near-clones are permitted.
- Selection for problem-solving performance

Pushpop Results

- In adaptive populations:
  - Species are more numerous
  - Diversification processes are more reliable
  - Selection can promote diversity
  - Provides a possible explanation for the evolution of diversifying reproductive systems
SwarmEvolve 2.0

- Behavior (including reproduction) controlled by evolved Push programs
- Color, color-based agent discrimination controlled by agents
- Energy conservation
- Facilities for communication, energy sharing
- Ample user feedback (e.g. diversity metrics, agent energy determines size)

SwarmEvolve 2.0

- Goals:
  - Superior problem-solving performance
  - Tractable analysis
- Push3
- Asexual
- Children produced on demand (not during fitness testing)
- Constraints on selection and birth
- Still work in progress

AutoPush

<table>
<thead>
<tr>
<th>Instruction(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUP, POP, SWAP, REP, _, NOOP, PULL, PULLUP, CONVERT, CAR, CDR, QUOTE, ATOM, NULL, NTH, +, -, /, &gt;, &lt;, NOT, AND, NAND OR, NOR, DO*, IF</td>
<td>Standard Push instructions (See [11])</td>
</tr>
<tr>
<td>VectorX, VectorY, VectorZ, VPlus, VMinus, VTimes, VDivide, VectorLength, Make-Vector</td>
<td>Vector access, construction, and manipulation</td>
</tr>
<tr>
<td>Randl, RandlF, RandlV, RandIC</td>
<td>Random number, vector, and code generators</td>
</tr>
<tr>
<td>SetServoSetpoint, SetServoGain, Servo</td>
<td>Servo-based persistent memory</td>
</tr>
<tr>
<td>Mutate, Crossover</td>
<td>Stochastic list manipulation (parameters from stacks)</td>
</tr>
<tr>
<td>Spawn</td>
<td>Produce a child with code from stack</td>
</tr>
<tr>
<td>ToFood, FoodIntensity</td>
<td>Vector to energy source, Energy of energy source</td>
</tr>
<tr>
<td>MyAge, MyEnergy, MyHue, MyVelocity, MyLocation, MyProgram</td>
<td>Information about self</td>
</tr>
<tr>
<td>ToFriend, FriendAge, FriendEnergy, FriendHue, FriendVelocity, FriendLocation, FriendProgram</td>
<td>Information about closest agent of similar hue</td>
</tr>
<tr>
<td>ToOther, OtherAge, OtherEnergy, OtherHue, OtherVelocity, OtherLocation, OtherProgram</td>
<td>Information about closest agent of non-similar hue</td>
</tr>
<tr>
<td>FeedFriend, FeedOther</td>
<td>Transfer energy to closest agent of indicated category</td>
</tr>
</tbody>
</table>
Evolving Modular Programs

With Code Manipulation

- Transform code as data on “code” stack
- Execute transformed code with code.do, etc.
- Simple uses of modules can be evolved easily
- Does not scale well to large/complex systems

Evolving Modular Programs

With Execution Stack Manipulation

- Code queued for execution is stored on an “execution stack”
- Allow programs to duplicate and manipulate code that on the stack
- Example: (3 exec.dup (1 integer.+))
- More parsimonious, but same scaling issue

Evolving Modular Programs

With Named Modules

- Uses Push’s “name” stack
- Example:

  (plus1 exec.define (1 integer.+))
  ...
  plus1

- Coordinating definitions/references is tricky
  and this never arises in evolution!

Module Identity

- How are modules recognized by other components of a system?
- Where do module identities come from?
- How can module identity co-evolve with modular architecture?
**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 Altruism**

- Individuals have tags and tag-difference tolerances
- Donate when Δtags ≤ tolerance
- Riolo *et al.* (*Nature*, 2001) showed that tag-based altruism can evolve; Roberts & Sherratt (*Nature*, 2002) claimed it would not evolve under more realistic conditions

**Evolving Modular Programs**

*With tags*

- 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**

---

**Tags in Push**

- Tags are integers embedded in instruction names.
- Instructions like `tag.exec.123` with tag values.
- Instructions like `tagged.456` recall values by the closest matching tag.
- If a single value has been tagged then all tag references will recall (and execute) values.
- The number of tagged values can grow incrementally over evolutionary time.

**Lawnmower Problem**

- Used by Koza to demonstrate utility of ADFs for scaling GP up to larger problems.

---

**Lawnmower Instructions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td><code>left, mow, v8a, frog, R_{08}</code></td>
</tr>
</tbody>
</table>
| Tag       | `left, mow, v8a, frog, R_{08},
  tag.exec.[1000], tagged.[1000]` |
| Exec      | `left, mow, v8a, frog, R_{08},
  exec.dup, exec.pop, exec.rot,
  exec.swap, exec.k, exec.s, exec.y` |

**Lawnmower Effort**

A graph showing computational effort with different conditions: Basic, Tag, and Exec.
Lawnmower Effort

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>8x4</th>
<th>8x6</th>
<th>8x8</th>
<th>8x10</th>
<th>8x12</th>
</tr>
</thead>
<tbody>
<tr>
<td>instr set</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>basic</td>
<td>10000</td>
<td>30000</td>
<td>114000</td>
<td>320000</td>
<td>630000</td>
</tr>
<tr>
<td>tag</td>
<td>7000</td>
<td>2000</td>
<td>29000</td>
<td>&lt;1000</td>
<td>5000</td>
</tr>
<tr>
<td>exec</td>
<td>12000</td>
<td>5000</td>
<td>28000</td>
<td>5000</td>
<td>17000</td>
</tr>
</tbody>
</table>

Dirt-Sensing, Obstacle-Avoiding Robot Problem

Like the lawnmower problem but harder and less uniform

DSOAR Instructions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>if-dirty, if-obstacle, left, mop, v8a, frog, ( R_{v8} )</td>
</tr>
<tr>
<td>Tag</td>
<td>if-dirty, if-obstacle, left, mop, v8a, frog, ( R_{v8} ), tag.exec.[1000], tagged.[1000]</td>
</tr>
<tr>
<td>Exec</td>
<td>if-dirty, if-obstacle, left, mop, v8a, frog, ( R_{v8} ), exec.dup, exec.pop, exec.rot, exec.swap, exec.k, exec.s, exec.y</td>
</tr>
</tbody>
</table>

DSOAR Effort

![Graph showing computational effort vs problem size](image)
**DSOAR Effort**

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>8x4</th>
<th>8x6</th>
<th>8x8</th>
<th>8x10</th>
<th>8x12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr Set</td>
<td>basic</td>
<td>tag</td>
<td>exec</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1584000</td>
<td>216000</td>
<td>450000</td>
<td>430083000</td>
<td>inf</td>
</tr>
<tr>
<td></td>
<td>864000</td>
<td>864000</td>
<td>2125000</td>
<td>inf</td>
<td>inf</td>
</tr>
<tr>
<td></td>
<td>342000</td>
<td>342000</td>
<td>4332000</td>
<td>inf</td>
<td>inf</td>
</tr>
<tr>
<td></td>
<td>2599000</td>
<td>2599000</td>
<td>16644000</td>
<td>inf</td>
<td>inf</td>
</tr>
<tr>
<td></td>
<td>3051000</td>
<td>3051000</td>
<td>7524000</td>
<td>inf</td>
<td>inf</td>
</tr>
</tbody>
</table>

**Evolved DSOAR Architecture (in one environment)**

![Diagram]

**Evolved DSOAR Architecture (in another environment)**

![Diagram]

**Tags in Trees**

- Example:
  
  \[
  \text{Example:} \\
  (\text{progn} \ (\text{tag.123} (+ a b)) \ \\
  (+ \text{tagged.034} \text{tagged.108}))
  \]

- Must do something about endless recursion
- Must do something about return values of tagging operations and references prior to tagging
- Non-trivial to support arguments in a general way
- Utility not clear from experiments conducted to date
Expressiveness and Assessment

• Expressive languages ease representation of programs that over-fit training sets
• Expressive languages ease representation of programs that work only on subsets of training sets
• Lexicase selection may help: Select parents by starting with a pool of candidates and then filtering by performance on individual fitness cases, considered one at a time

Future Work

• Expression of variable scope and local environments
• Expression of concurrency, parallelism, and time-based structures
• Applications for which expressiveness is likely to be essential, e.g. complete software applications and programs for agents in complex, dynamic, heterogeneous environments

Conclusions

• GP in expressive languages may allow for the evolution of complex software
• Minimal-syntax languages can be expressive, and GP systems that evolve programs in such languages can be simple
• Push is expressive, evolvable, successful, and extensible
• Tags appear to allow for the evolvable expression of program modularity

Thanks

This material is based upon work supported by the National Science Foundation under Grant No. 1017817. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation. Thanks also to Thomas Helmuth, Omri Bernstein, Kyle Harrington, Emma Tosch, Kwaku Yeboah Antwi, and Rebecca S. Neimark for discussions related to this work, to Josiah Erikson for systems support, and to Hampshire College for support for the Hampshire College Institute for Computational Intelligence.
References

http://hampshire.edu/spector/push


