Expressive Genetic Programming

Tutorial
Genetic and Evolutionary Computation Conference (GECCO-2013)
Amsterdam, The Netherlands

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).
ACM 978-1-4503-1964-5/13/07.

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.

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.
Course Agenda

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

Evolutionary Computation

Evolution, the Designer

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

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

\[
(+ (* X Y) \\
(+ 4 (- Z 23)))
\]

\[
(+ (* X Y) \\
(+ 4 (- Z 23)))
\]

\[
(+ (- (+ 2 Z) 2) \\
(+ 4 (- Z 23)))
\]

Recombining Lisp

Parent 1: 
\[
(+ (* X Y) \\
(+ 4 (- Z 23)))
\]

Parent 2: 
\[
(- (* 17 (+ 2 X)) \\
(* (- (* 2 Z) 1)) \\
(+ 14 (/ Y X)))
\]

Child 1: 
\[
(+ (- (* 2 Z) 1) \\
(+ 4 (- Z 23)))
\]

Child 2: 
\[
(- (* 17 (+ 2 X)) \\
(* (*) X Y) \\
(+ 14 (/ Y X)))
\]

Symbolic Regression

A simple example

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

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

Fitness = error (smaller is better)
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

$$(- (* (\% * 0.1 \times X \times)) (- (\% 0.1 0.1) (* X X))) 0.1)$$

Best Program, Gen 5

$$(- (* (* (\% X 0.1) (* 0.1 X)) (- X (\% 0.1 X)))) 0.1)$$
Best Program, Gen 12

Best Program, Gen 22

Expressiveness

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

Evolvability

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.
Modularity

- 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.

Modularity is Everywhere

Modularity in Software

- 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.

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 (1)

- Specialize GP techniques to directly support human programming language abstractions
- Strongly typed genetic programming
- Module acquisition/encapsulation systems
- Automatically defined functions
- Automatically defined macros
- Architecture altering operations

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))

Structure via GP (2)

- Specialize GP techniques to \textbf{indirectly} support human programming language abstractions
- Constrain genetic change, or repair after genetic change, to satisfy abstraction syntax
- Map from unstructured genomes to programs in languages that support abstraction (e.g. via grammars)

Structure via GP (3)

- Develop new program encodings, represented most generally as graphs
- Develop abstraction mechanisms for these representations
- Specialize GP techniques to directly or indirectly support abstraction in these new program encodings

Structure via GP (4)

- 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
  - \textbf{Push}
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.
\[( \text{2 3 INTEGER.}* 4.1 \ 5.2 \text{ FLOAT.}+ \text{ TRUE FALSE BOOLEAN.OR} )\]
**Same Results**

\[
( 2 3 \text{ INTEGER.}* 4.1 5.2 \text{ FLOAT.} + \\
\text{ TRUE FALSE BOOLEAN.OR } )
\]

\[
( 2 \text{ BOOLEAN.AND} 4.1 \text{ TRUE INTEGER.}/ \text{ FALSE} \\
3 5.2 \text{ BOOLEAN.OR INTEGER.}* \text{ FLOAT.} + )
\]
( 3.14 CODE.REVERSE CODE.CDR IN IN 5.0 FLOAT.> (CODE.QUOTE FLOAT.*) CODE.IF )

IN=4.0

exec  code  bool  int  float

3.14
CODE.REVERSE
CODE.CDR
IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF

3.14
exec  code  bool  int  float
exec  code  bool  int  float

exec  code  bool  int  float

exec  code  bool  int  float

exec  code  bool  int  float

CODE.IF  (CODE.QUOTE FLOAT.*)  FLOAT.> 5.0 IN  (CODE.CDR  CODE.REVERSE 3.14)  FALSE  3.14

exec  code  bool  int  float

exec  code  bool  int  float

exec  code  bool  int  float

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

IN=4.0

exec code bool int float

exec code bool int float

exec code bool int float

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>3.13</td>
<td>FLOAT.*</td>
<td>10.0</td>
<td>FLOAT./</td>
<td>(IN EXEC DUP (3.13 FLOAT.*) 10.0 FLOAT./)</td>
</tr>
<tr>
<td>10.0</td>
<td>FLOAT./</td>
<td>3.13</td>
<td>FLOAT.*</td>
<td>(IN EXEC DUP (3.13 FLOAT.*) 10.0 FLOAT./)</td>
</tr>
<tr>
<td>39.1876</td>
<td>FLOAT./</td>
<td>10.0</td>
<td>FLOAT.*</td>
<td>(IN EXEC DUP (3.13 FLOAT.*) 10.0 FLOAT./)</td>
</tr>
</tbody>
</table>
The Odd Problem

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

Combinators

- Standard $K$, $S$, and $Y$ combinators:
  - $\text{EXEC.K}$ removes the second item from the EXEC stack.
  - $\text{EXEC.S}$ pops three items (call them $A$, $B$, and $C$) and then pushes $(B \ C), C$, and then $A$.
  - $\text{EXEC.Y}$ inserts $\text{EXEC.Y} T$ under the top item ($T$).
- A $Y$-based “while” loop:
  
  ```
  \begin{verbatim}
  ( \text{EXEC.Y}
  \begin{cases}
  \text{<BODY/CONDITION>} & \text{EXEC.IF} \\
  \text{ }) & \text{EXEC.POP} \\
  \end{cases}
  \end{verbatim}
  ```

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.* ) )

The ULTRA Operator

- Uniform Linear Transformation with Repair and Alternation
- 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

Auto-simplification

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

ULTRA on the bioavailability problem

Figure 1 gives two box plots from our sets runs of the bioavailability problem, conducted 100 runs for each choice of operators. The results from the Pagie-1 problem. We conducted 100 runs for each choice of operators. The mean program sizes in our Pagie-1 experiments are given in Figure 3. Runs using ULTRA found perfect solutions in 15 out of 100 runs, whereas runs using 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 on the RMSE of the best program as measured on the training set. The right plot shows where each set contains 300 runs. The left plot shows the root mean square error outside the of the visible plot. Outliers on the ULTRA set fell on the 45/45/10 set, and 3 outliers on the 81/9/10 set, 7 outliers that in the right plot, 8 outliers, plotted as points. Note beyond the whiskers are outliers. The mean program sizes with respect to evolutionary time are plotted in Figure 2. The runs using subtree replacement show steady growth in program sizes, whereas those using ULTRA quickly fall at the beginning of the run and then remain relatively steady. The lower program sizes of ULTRA runs may contribute to its ability to not overfit the data. The runs using subtree replacement show good models of the training data than subtree replacement without running into problems of overfitting the data, which would lead to worse performance on the test set. Results on the Pagie-1 problem. We conducted 100 runs for each choice of operators. Table 3 presents the results of our experiments on the Pagie-1 problem. PushGP

#### Table 3

<table>
<thead>
<tr>
<th>Operators</th>
<th>Mean Program Size</th>
<th>Test RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtree Replacement 45/45/10</td>
<td>500</td>
<td>0.01</td>
</tr>
<tr>
<td>Subtree Replacement 81/9/10</td>
<td>500</td>
<td>0.01</td>
</tr>
<tr>
<td>ULTRA</td>
<td>500</td>
<td>0.01</td>
</tr>
</tbody>
</table>

As recommended in (Luke and Panait, 2002; McDermott et al, 2002) we use unpaired t-tests to compare the differences in MBF for different conditions. For the Pagie-1 problem we use mean error across fitness cases, and do not use a test of whether the RMSE results of two runs come from the same distribution using the Kruskal-Wallis one-way analysis of variance at an unpaired t-test at p = 0.01.

Results from the experiments conducted 100 runs for each choice of operators. The mean best fitness (MBF) is the mean of the best individual fitnesses attained in each run. The fitnesses given here are the mean errors across test cases, not the summed errors. As recommended in (Luke and Panait, 2002; McDermott et al, 2002) we use unpaired t-tests to compare the differences in MBF for different conditions.

In each run, the fitness is calculated as the sum of the errors for each test case. We present the number of successes and mean best fitnesses for the Pagie-1 bioavailability problem. We conducted 100 runs for each choice of operators. The results from the Pagie-1 problem. We conducted 100 runs for each choice of operators. The mean program sizes in our Pagie-1 experiments are given in Figure 3. Runs using ULTRA found perfect solutions in 15 out of 100 runs, whereas runs using subtree replacement found none.
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

Genetic Programming for Finite Algebras

Lee Spector  
Cognitive Science  
Hampshire College  
Amherst, MA 01002  
lspector@hampshire.edu

David M. Clark  
Mathematics  
SUNY New Paltz  
New Paltz, NY 12561  
clarkd@newpaltz.edu

Ian Lindsay  
Mathematics  
Hampshire College  
Amherst, MA 01002  
iml04@hampshire.edu

Bradford Barr  
Hampshire College  
Amherst, MA 01002  
bradford.barr@gmail.com

Jon Klein  
Hampshire College  
Amherst, MA 01002  
jk@artificial.com

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

- 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

# Species vs. Mother/Child Differences

Note distribution of “+” points: adaptive populations have many species and mother/daughter differences in a relatively high, narrow range (above near-clone levels).
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)

<table>
<thead>
<tr>
<th>Instruction(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUP, POP, SWAP, REP,  , NOOP, PULL, PULLDUP, CONVERT, CAR, CDR, QUOTE, ATOM, NULL, NTH, +, <em>, /, &gt;, &lt;, NOT, AND, NAND OR, NOR, DO</em>, 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>RandI, RandF, RandV, RandC</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 code stack</td>
</tr>
<tr>
<td>ToFood</td>
<td>Vector to energy source</td>
</tr>
<tr>
<td>FoodIntensity</td>
<td>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>

SwarmEvolve 2.0

Winner, Best Paper Award, AAAA Track, GECCO-2003
AutoPush

• 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

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: \((3 \text{ exec.dup (1 integer.+)})\)
• More parsimonious, but same scaling issue

Evolving Modular Programs

With Named Modules

• Uses Push’s “name” stack
• Example:
  \[(\text{plus1 exec.define (1 integer.+))} \]
  \[\ldots\]
  \[\text{plus1}\]
• 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 $\Delta$tags $\leq$ 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 tag values
- Instructions like tagged.456 recall values by 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>left, mow, v8a, frog, $R_{v8}$</td>
</tr>
<tr>
<td>Tag</td>
<td>left, mow, v8a, frog, $R_{v8}$, tag.exec.[1000], tagged.[1000]</td>
</tr>
<tr>
<td>Exec</td>
<td>left, mow, v8a, frog, $R_{v8}$, exec.dup, exec.pop, exec.rot, exec.swap, exec.k, exec.s, exec.y</td>
</tr>
</tbody>
</table>
Lawnmower Effort

Problem Size

Computational Effort

Basic
Tag
Exec

Basic
Tag
Exec

8x4
8x6
8x8
8x10
8x12

Problem Size

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](image1)

**Computational Effort**

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Basic</th>
<th>Tag</th>
<th>Exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>8x4</td>
<td>1584000</td>
<td>216000</td>
<td>450000</td>
</tr>
<tr>
<td>8x6</td>
<td>430083000</td>
<td>864000</td>
<td>2125000</td>
</tr>
<tr>
<td>8x8</td>
<td>inf</td>
<td>3420000</td>
<td>4332000</td>
</tr>
<tr>
<td>8x10</td>
<td>inf</td>
<td>2599000</td>
<td>16644000</td>
</tr>
<tr>
<td>8x12</td>
<td>inf</td>
<td>3051000</td>
<td>7524000</td>
</tr>
</tbody>
</table>

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

![Diagram showing architecture](image2)

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

![Diagram showing architecture](image3)
Tags in Trees

- Example:
  (progn (tag.123 (+ a b))
  (+ tagged.034 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 (implemented in Push, but not yet studied systematically)
- 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 Helmut, Omri Bernstein, Kyle Harrington, Emma Tosch, Kwaku Yeboah Antwi, and Rebecca S. Neimarck 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.


General references on genetic programming


