Flight of the FINCH through the Java Wilderness

Michael Orlov and Moshe Sipper

orlovm, sipper@cs.bgu.ac.il

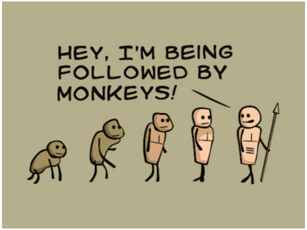
Department of Computer Science
Ben-Gurion University, Israel

GPTP-2010 Workshop
20-22 May 2010, University of Michigan

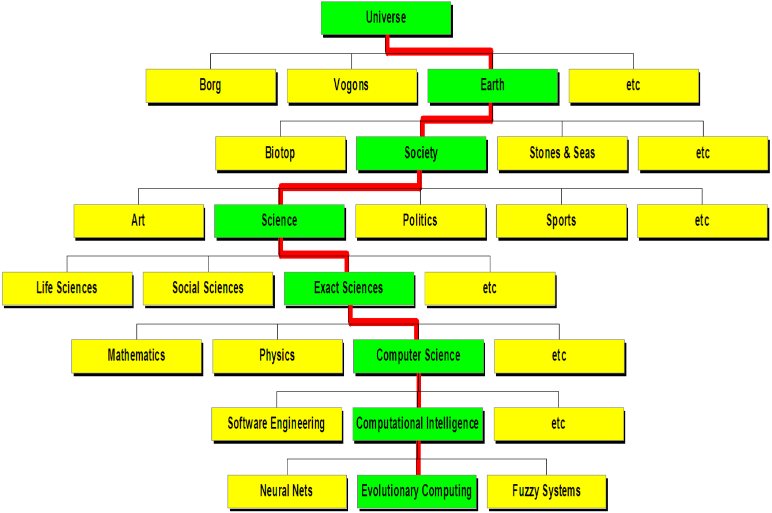
Evolutionary Algorithms

A class of probabilistic optimization algorithms inspired by the biological process of

Evolution by Natural Selection



The Universe. . .



Source: Eiben & Smith, 2003

Turing on Evolution

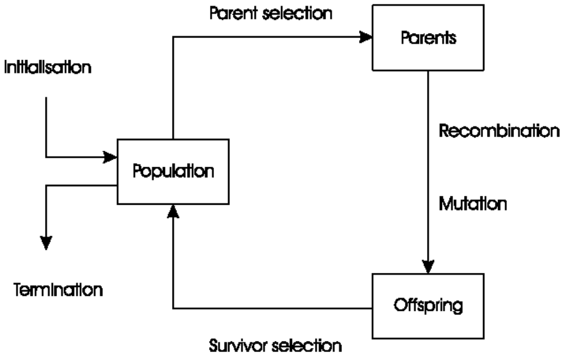
Instead of trying to produce a programme to simulate the adult mind, why not rather try to produce one which simulates the child's ? If this were then subjected to an appropriate course of education one would obtain the adult brain. Presumably the child-brain is something like a note-book as one buys it from the stationers. Rather little mechanism, and lots of blank sheets. (Mechanism and writing are from our point of view almost synonymous.) Our hope is that there is so little mechanism in the child-brain that something like it can be easily programmed. The amount of work in the education we can assume, as a first approximation, to be much the same as for the human child.

We have thus divided our problem into two parts. The child-programme and the education process. These two remain very closely connected. We cannot expect to find a good child-machine at the first attempt. One must experiment with teaching one such machine and see how well it learns. One can then try another and see if it is better or worse. There is an obvious connection between this process and evolution, by the identifications

Structure of the child machine	=	Hereditary material
Changes ,, ,,	=	Mutations
Natural selection	=	Judgment of the experimenter

A. M. Turing, "Computing machinery and intelligence," Mind, 59(236), 433-460, Oct. 1950

General Scheme of EAs



Source: Eiben & Smith, 2003

Pseudocode of Typical EA

Initialize population with random candidate solutions
Evaluate the fitness each candidate

while termination condition not met **do**
    Select parents
    Recombine pairs of parents
    Mutate the resulting offspring
    Evaluate the new candidates
end while

Stochastic Operators

- **Fitness** value
  - computed for each individual
- **Selection**
  - probabilistically selects fittest individuals
- **Recombination**
  - decomposes two distinct solutions
  - randomly mixes their parts to form novel solutions
- **Mutation**
  - randomly perturbs a candidate solution

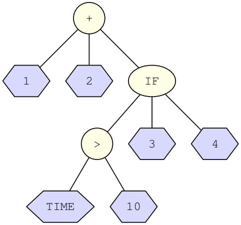
Different Types of EAs

- Historically, different flavors of EAs have been associated with different representations
  - Binary strings → *Genetic Algorithms*
  - Real-valued vectors → *Evolution Strategies*
  - Finite-state machines → *Evolutionary Programming*
  - LISP trees → *Genetic Programming*
- These differences are largely irrelevant, best strategy
  - design **representation** to suit problem
  - design **variation operators** to suit representation

Genetic Programming (GP)

- EA, with individuals in population represented as computer programs
- "Classical" GP uses LISP (S-Expressions)

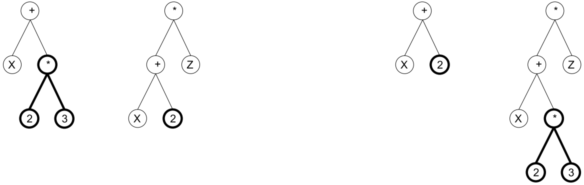
$$(+\ 1\ 2\ (\mathrm{IF}\ (>\ \mathrm{TIME}\ 10)\ 3\ 4))$$



Genetic Programming *(cont'd)*

Before **After**

Crossover



Mutation

Blurb...

artificial evolution is highly simplified relative to biology

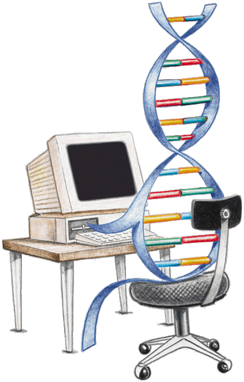
BUT

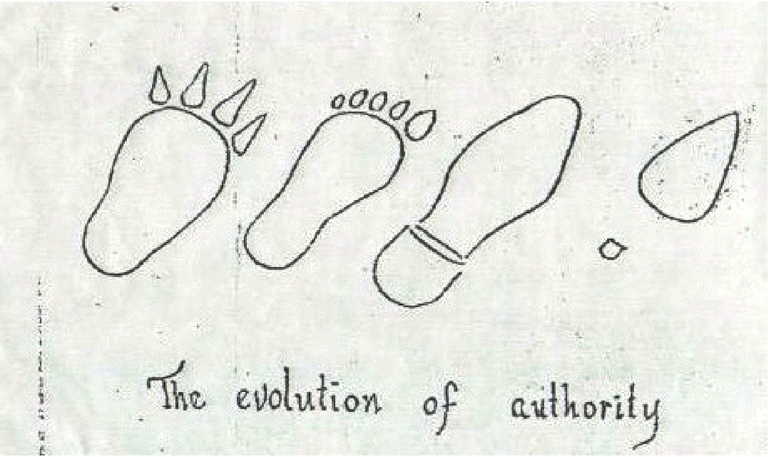
repeatedly produces
complex, **interesting**, and **useful** solutions

EA in Full Bloom

Anecdotal listing of (successful) application areas:

- Acoustics
- Aerospace engineering
- Astronomy and astrophysics
- Chemistry
- Electrical engineering
- Financial markets
- Game playing
- Geophysics
- Materials engineering
- Mathematics and algorithmics
- Military and law enforcement
- Molecular biology
- Pattern recognition and data mining
- Robotics
- Routing and scheduling
- Systems engineering




The evolution of authority

GP: Programs or Representations?

"While it is common to describe GP as evolving **programs**, GP is not typically used to evolve programs in the familiar Turing-complete languages humans normally use for software development."

A Field Guide to Genetic Programming
[Poli, Langdon, and McPhee, 2008]

GP: Programs or Representations?

"While it is common to describe GP as evolving **programs**, GP is not typically used to evolve programs in the familiar Turing-complete languages humans normally use for software development."

"It is instead more common to evolve programs
(or expressions or formulae)
in a **more constrained** and often **domain-specific language**."

A Field Guide to Genetic Programming
[Poli, Langdon, and McPhee, 2008]

Michael Orlov
Moshe Sipper

Our Goals

From programs. . .

Evolve actual programs

written in Java

. . . to software!

Improve (existing) software

written in unrestricted Java

Our Goals

From programs. . .

Evolve actual programs

written in Java

. . . to software!

Improve (existing) software

written in unrestricted Java

Extending prior work

Existing work uses **restricted subsets** of Java bytecode as **representation language** for GP individuals

We evolve **unrestricted bytecode**

Let's Evolve Java Source Code

- Rely on the building blocks in the initial population
- Defining **genetic operators** is problematic
- How do we define **good** source-code crossover?

Factorial *(recursive)*

```
class F {
  int fact(int n) {
    int ans = 1;

    if (n > 0)
      ans = n *
        fact(n-1);

    return ans;
  }
}
```

$\Leftarrow$

Factorial *(iterative)*

```
class F {
  int fact(int n) {
    int ans = 1;

    for (; n > 0; n--)
      ans = ans * n;


    return ans;
  }
}
```

Oops

- **Source-level crossover** typically produces garbage

Factorial *(recursive $\overleftarrow{\times}$ iterative)*

```
class F {
  int fact(int n) {
    int ans = 1;

    if (n > = 1;
      for (;  n > 0;  n--)
        ans = ans * n;  n-1);

    return ans;
  }
}
```

Oops

- **Source-level crossover** typically produces garbage

Factorial *(recursive* $\overleftarrow{\times}$ *iterative)*

```
class F {
  int fact(int n) {
    int ans = 1;

    if (n > = 1;
      for (;  n > 0;  n--)
        ans = ans * n;  n-1);

    return ans;
  }
}
```

Parse Trees

- Maybe we can design **better** genetic operators?

Parse Trees

- Maybe we can design **better** genetic operators?
- Maybe. . . but too much harsh **syntax**
  Possibly use **parse tree**?

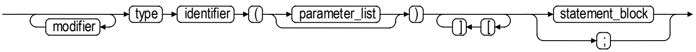
Parse Trees

- Maybe we can design **better** genetic operators?
- Maybe... but too much harsh **syntax**
  Possibly use **parse tree**?

Just one BNF rule *(of many)*

$$
\begin{aligned}
&\texttt{method\_declaration} ::\Longrightarrow \\
&\qquad \texttt{modifier}^{*}\ \texttt{type identifier} \\
&\qquad \text{``(''}\ \texttt{parameter\_list}^{?}\ \text{``)''}\ \text{``[ ]''}^{*} \\
&\qquad \langle\ \texttt{statement\_block}\ |\ \text{``;''}\ \rangle
\end{aligned}
$$

method_declaration



Bytecode

Better than parse trees:

Let's use **bytecode**!

Java Virtual Machine *(JVM)*

- Source code is compiled to **platform-neutral bytecode**
- Bytecode is executed with **fast** just-in-time compiler
- High-order, **simple** yet powerful architecture
- **Stack-based**, supports hierarchical object **types**
- Not limited to Java!

  (Scala, Groovy, Jython, Kawa, Clojure, . . .)

Bytecode (cont'd)
Some basic bytecode instructions

Stack ↔ Local variables

iconst 1	pushes `int` 1 onto operand stack
aload 5	pushes **object** in local variable 5 onto stack *(object **type** is deduced when class is loaded)*
dstore 6	pops two-word `double` to local variables 6–7

Bytecode (cont'd)
Some basic bytecode instructions

Stack ↔ Local variables

iconst 1	pushes `int` 1 onto operand stack
aload 5	pushes **object** in local variable 5 onto stack
	*(object **type** is deduced when class is loaded)*
dstore 6	pops two-word `double` to local variables 6–7

Arithmetic instructions *(affect operand stack)*

imul	pops two `int`s from stack, pushes multiplication result

Bytecode (cont'd)
Some basic bytecode instructions

Stack ↔ Local variables

iconst 1	pushes `int` 1 onto operand stack
aload 5	pushes **object** in local variable 5 onto stack *(object **type** is deduced when class is loaded)*
dstore 6	pops two-word `double` to local variables 6–7

Arithmetic instructions *(affect operand stack)*

imul	pops two `int`s from stack, pushes multiplication result

Control flow *(uses operand stack)*

ifle +13	pops `int`, jumps +13 bytes if value $\leqslant 0$
lreturn	pops two-word `long`, returns to caller's stack

Bytecode (cont'd)
Evolutionary operators

- Java bytecode is **less fragile** than source code

Bytecode (cont'd)
Evolutionary operators

- Java bytecode is **less fragile** than source code
- But, bytecode must be **correct** in order to run **correctly**

Correct bytecode requirements

Stack use is **type-consistent**

(e.g., can't multiply an `int` by an **Object***)*

Local variables use is **type-consistent**

(e.g., can't read an `int` after storing an **Object***)*

No stack underflow

No reading from uninitialized variables

Bytecode (cont'd)
Evolutionary operators

- Java bytecode is **less fragile** than source code
- But, bytecode must be **correct** in order to run **correctly**

Correct bytecode requirements

Stack use is **type-consistent**

(e.g., can't multiply an `int` *by an* **Object***)*

Local variables use is **type-consistent**

(e.g., can't read an `int` *after storing an* **Object***)*

No stack underflow

No reading from uninitialized variables

- So, genetic operators are still delicate

Bytecode (cont'd)
Evolutionary operators

- Java bytecode is **less fragile** than source code
- But, bytecode must be **correct** in order to run **correctly**

Correct bytecode requirements

Stack use is **type-consistent**
(e.g., can't multiply an `int` by an **Object***)*
Local variables use is **type-consistent**
(e.g., can't read an `int` after storing an **Object***)*
No stack underflow
No reading from uninitialized variables

- So, genetic operators are still delicate
- Need **good** genetic operators to produce **correct** offspring

Bytecode (cont'd)
Evolutionary operators

- Java bytecode is **less fragile** than source code
- But, bytecode must be **correct** in order to run **correctly**

Correct bytecode requirements

Stack use is **type-consistent**
 (e.g., can't multiply an `int` *by an* **Object***)*
Local variables use is **type-consistent**
 (e.g., can't read an `int` *after storing an* **Object***)*
No stack underflow
No reading from uninitialized variables

- So, genetic operators are still delicate
- Need **good** genetic operators to produce **correct** offspring
- Conclusion: Avoid **bad** crossover and mutation

Evolutionary Operators

Unidirectional bytecode crossover:

Bytecode A

. . .

. . .

. . .

Section α

. . .

. . .

. . .

Bytecode B

. . .

. . .

. . .

Section β

. . .

. . .

. . .

Evolutionary Operators

Unidirectional bytecode crossover:

Bytecode A

. . .

. . .

. . .

Section α

. . .

. . .

. . .

Bytecode B

. . .

. . .

. . .

Section β

. . .

. . .

. . .

Evolutionary Operators

Good and *bad* crossovers

Parent **A**:

Factorial *(recursive)*

```
class F
{
  int fact(int n)
  {
    int ans = 1;

    if (n > 0)
      ans = n * fact(n-1);

    return ans;
  }
}
```

Compiled bytecode

```
 0 iconst_1
 1 istore_2
 2 iload_1
 3 ifle 16
 6 iload_1
 7 aload_0
 8 iload_1
 9 iconst_1
10 isub
11 invokevirtual #2
14 imul
15 istore_2
16 iload_2
17 ireturn
```

Evolutionary Operators
Good and bad crossovers

Parent **B**:

Factorial *(iterative)*

```
class F
{
  int fact(int n)
  {
    int ans = 1;

    for (;  n > 0;  n--)
      ans = ans * n;

    return ans;
  }
}
```

Compiled bytecode

```
 0 iconst_1
 1 istore_2
 2 iload_1
 3 ifle 16
 6 iload_2
 7 iload_1
 8 imul
 9 istore_2
10 iinc 1, -1
13 goto 2
16 iload_2
17 ireturn
```

Evolutionary Operators

Good and bad crossovers

Replace a section in **A** with section from **B**

Bytecode A

```
0  iconst_1
1  istore_2
2  iload_1
3  ifle 16
6  iload_1
7  aload_0
8  iload_1
9  iconst_1
10 isub
11 invokevirtual #2
14 imul
15 istore_2
16 iload_2
17 ireturn
```

⇐

Bytecode B

```
0  iconst_1
1  istore_2
2  iload_1
3  ifle 16
6  iload_2
7  iload_1
8  imul
9  istore_2
10 iinc 1, -1
13 goto 2
16 iload_2
17 ireturn
```

Evolutionary Operators
Good crossover example

Stack use is depth- and type-consistent, variables are initialized.

Bytecode A

```
0  iconst_1
1  istore_2
2  iload_1
3  ifle 16
6  iload_1
7  aload_0
8  iload_1
9  iconst_1
10 isub
11 invokevirtual #2
14 imul
15 istore_2
16 iload_2
17 ireturn
```

⇐

Bytecode B

```
0  iconst_1
1  istore_2
2  iload_1
3  ifle 16
6  iload_2
7  iload_1
8  imul
9  istore_2
10 iinc 1, -1
13 goto 2
16 iload_2
17 ireturn
```

Evolutionary Operators
Good crossover example

Stack use is depth- and type-consistent, variables are initialized.

Bytecode ($A \overleftarrow{\times} B$)

```
 0 iconst_1
 1 istore_2
 2 iload_1
 3 ifle 12
 6 iload_1
 7 iload_2
 8 iload_1
 9 imul
10 imul
11 istore_2
12 iload_2
13 ireturn
```

Decompiled source

```
class F
{
  int fact(int n)
  {
    int ans = 1;

    if (n > 0)
      ans = n * (ans * n);

    return ans;
  }
}
```

Evolutionary Operators
Bad crossover example

Stack use is depth- and type-inconsistent.

Bytecode A

```
0  iconst_1
1  istore_2
2  iload_1
3  ifle 16
6  iload_1
7  aload_0
8  iload_1
9  iconst_1
10 isub
11 invokevirtual #2
14 imul
15 istore_2
16 iload_2
17 ireturn
```

⇐

Bytecode B

```
0  iconst_1
1  istore_2
2  iload_1
3  ifle 16
6  iload_2
7  iload_1
8  imul
9  istore_2
10 iinc 1, -1
13 goto 2
16 iload_2
17 ireturn
```

Evolutionary Operators
Bad crossover example

Stack use is depth- and type-inconsistent.

Bytecode ($A \overleftarrow{\times} B$)

```
 0 iconst_1
 1 istore_2
 2 iload_1
 3 ifle 13
 6 iload_2
 7 iload_1
 8 invokevirtual #2
11 imul
12 istore_2
13 iload_2
14 ireturn
```

“Decompiled” source

```
class F {
  int fact(int n)
  {
    int ans = 1;

    if (n > 0)
      ans = ans.fact(n) * ?;

    return ans;
  }
}
```

Compatible Crossover
Constraints of unidirectional crossover $\mathbf{A} \overleftarrow{\times} \mathbf{B}$

Good crossover is achieved by respecting bytecode constraints:

(α *is target section in* ***A***, β *is source section in* ***B***)

Operand stack

e.g., β doesn't pop values with types incompatible to those popped by α

Compatible Crossover
Constraints of unidirectional crossover $\mathbf{A} \overleftarrow{\times} \mathbf{B}$

Good crossover is achieved by respecting bytecode constraints:
(α *is target section in* ***A***, β *is source section in* ***B***)

Operand stack

e.g., β doesn't pop values with types incompatible to those popped by α

Local variables

e.g., variables read by β in **B** must be written before α in **A** with compatible types

Compatible Crossover
Constraints of unidirectional crossover $A \overleftarrow{\times} B$

Good crossover is achieved by respecting bytecode constraints:
(α *is target section in* ***A***, β *is source section in* ***B***)

Operand stack

e.g., β doesn't pop values with types incompatible to those popped by α

Local variables

e.g., variables read by β in **B** must be written before α in **A** with compatible types

Control flow

e.g., branch instructions in β have no "outside" destinations

Formal Definition
(Example of operand stack requirement)

α and β have compatible stack frames up to stack depth of β:
pops of α have identical or narrower types as pops of β;
pushes of β have identical or narrower types as pushes of α

Good crossover

	α	β
pre-stack	`**AB`	`**AA`
post-stack	`**B`	`**C`
depth	3	2

Stack pops "`AB`"
(2 stop tack frames) are narrower than "`AA`",
whereas stack push "`C`" is narrower than "`B`"

Types hierarchy: `C` $\rightarrow$ `B` $\rightarrow$ `A`

(see [Orlov and Sipper, 2009, 2010] for full formal definitions)

Formal Definition
(Example of operand stack requirement)

α and β have compatible stack frames up to stack depth of β:
pops of α have identical or narrower types as pops of β;
pushes of β have identical or narrower types as pushes of α

Bad crossover

	α	β
pre-stack	`**AB`	`**Af`
post-stack	`**B`	`**A`
depth	3	2

Stack pops “`AB`” are not narrower than “`Af`”
(B and f are incompatible);
stack push “`A`” is not narrower than “`B`”

Types hierarchy: `B` $\rightarrow$ `A`; `f` is a `float`

(see [Orlov and Sipper, 2009, 2010] for full formal definitions)

Parameters

- Objective: symbolic regression, $x^4 + x^3 + x^2 + x$
- Fitness: sum of errors on 20 random data points in $[-1, 1]$
- Input: **Number** `num` *(a Java type)*

Symbolic Regression
As an evolutionary example...

Parameters

- Objective: symbolic regression, $x^4 + x^3 + x^2 + x$
- Fitness: sum of errors on 20 random data points in $[-1, 1]$
- Input: **Number** `num` *(a Java type)*

Seeding

- Population initialized using seeding

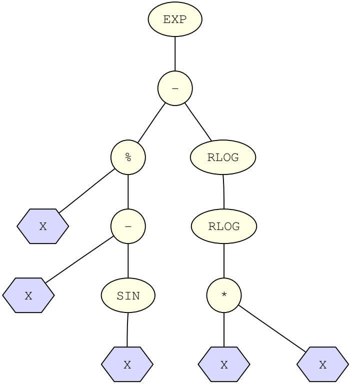
  [Langdon and Nordin, 2000]
- Seed population with clones of Koza's original worst-of-generation-0

  [Koza, 1992]

Symbolic Regression
Seeding with Koza's worst-of-generation-0

Original **Lisp** individual and its **tree** representation:

```
(EXP (- (% X (- X (SIN X))) (RLOG (RLOG (* X X)))))
```



Symbolic Regression
Seeding with Koza's worst-of-generation-0

Translation to **unrestricted Java**

```
class SimpleSymbolicRegression {
  Number simpleRegression(Number num) {
    double x     = num.doubleValue();
    double llsq  = Math.log(Math.log(x*x));
    double dv    = x / (x - Math.sin(x));
    double worst = Math.exp(dv - llsq);
    return Double.valueOf(worst + Math.cos(1));
  }

  /* Rest of class omitted */
}
```

We added a couple of building blocks in the last line

Symbolic Regression
Setup and Statistics

Setup *(similar to Koza's)*

- Population: 500 individuals
- Generations: 51 (or less)
- Probabilities: $p_{\text{cross}} = 0.9$
  (α and β segments are uniform over segment sizes)
- Selection: binary tournament

Symbolic Regression
Setup and Statistics

Setup *(similar to Koza's)*

- Population: 500 individuals
- Generations: 51 (or less)
- Probabilities: $p_{\text{cross}} = 0.9$
  (α and β segments are uniform over segment sizes)
- Selection: binary tournament

Statistics

- Yield: 99% of runs successful (out of 100)
- Runtime: 30–60 s on dual-core 2.6 GHz Opteron
- Memory limits: insignificant w.r.t. runtime

Symbolic Regression
Evolved perfect individuals

A perfect solution easily evolves:

(beware of decompiler quirks!)

```
class SimpleSymbolicRegression_0_7199 {
  Number simpleRegression(Number num) {
    double d = num.doubleValue();
    d = num.doubleValue();
    double d1 = d; d = Double.valueOf(d + d * d *
          num.doubleValue()).doubleValue();
    return Double.valueOf(d +
          (d = num.doubleValue()) * num.doubleValue());
  }

  /* Rest of class unchanged */
}
```

Computes $(x + x \cdot x \cdot x) + (x + x \cdot x \cdot x) \cdot x \;=\; x(1+x)(1+x^2)$

Symbolic Regression
Evolved perfect individuals

Another solution:

```java
class SimpleSymbolicRegression_0_2720 {
  Number simpleRegression(Number num) {
    double d = num.doubleValue();
    d = d; d = d;
    double d1 = Math.exp(d - d);
    return Double.valueOf(num.doubleValue() *
      (num.doubleValue() * (d * d + d) + d) + d);
  }

  /* Rest of class unchanged */
}
```

Computes $x \cdot (x \cdot (x \cdot x + x) + x) + x \;=\; x(1 + x(1 + x(1 + x)))$

Parameters

- Objective: symbolic regression: $x^9 + x^8 + \cdots + x^2 + x$
- Fitness: incremental evaluation, $\sum_{i=1}^{n} x^i$, up to $n = 9$
- Crossover: Gaussian distribution over segment sizes
- Parsimony pressure, growth limit

Initialization

- Worst of generation-0 from simple regression

Java Wilderness
Complex Regression

A perfect solution:

```java
Number simpleRegression(Number num) {
  double d = num.doubleValue();
  return Double.valueOf(d + (d * (d * (d +
      ((d = num.doubleValue()) +
        (((num.doubleValue() * (d = d) + d) *
          d + d) * d + d) * d)
      * d) + d) + d) * d);
}
```

Computes

$$x+(x\cdot(x\cdot(x+(x+(((x\cdot x+x)\cdot x+x)\cdot x+x)\cdot x)\cdot x)+x)+x)\cdot x$$

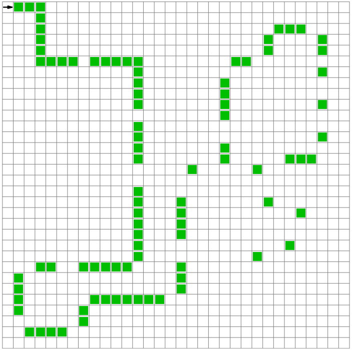
Parameters

- Objective: consume all food pellets on Santa Fe trail
- Fitness: number of food pellets consumed
- Crossover: Gaussian distribution over segment sizes
- Parsimony pressure, growth limit

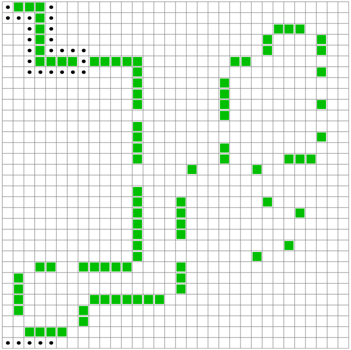
Initialization

- “Avoider” (zero-fitness)

Santa Fe Trail:



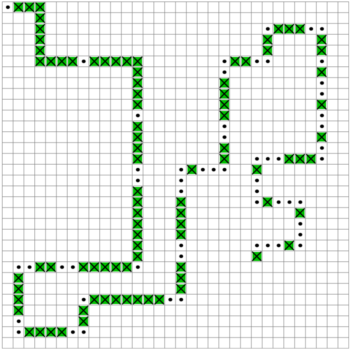
(a) Initial setup



(b) Avoider

A perfect solution:

```
void step() {
  if (foodAhead()) {
    move(); right();
  }
  else {
    right(); right();
    if (foodAhead())
      left();
    else {
      right(); move();
      left();
    }
    left(); left();
  }
}
```



Java Wilderness
Intertwined Spirals

Parameters

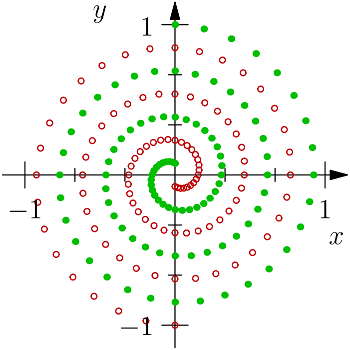
- Objective: two-class classification of intertwined spirals
- Fitness: number of correctly classified points

Initialization

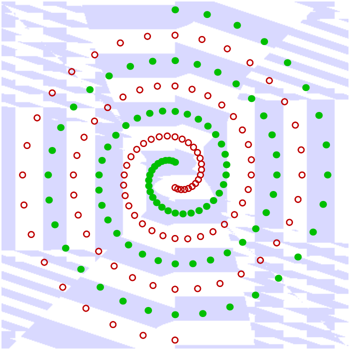
- Arbitrarily organized repository of building blocks: floating-point arithmetics, trigonometric functions, and polar-rectangular coordinates conversion

Java Wilderness
Intertwined Spirals

Intertwined spirals:

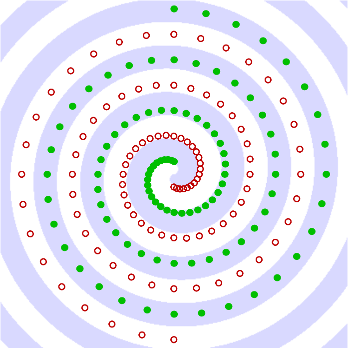


(e) Initial setup



(f) Koza's evolved solution

A perfect solution:



Computes the (approximate) sign of $\sin\left(\frac{9}{4}\pi^2\sqrt{x^2+y^2} - \tan^{-1}\frac{y}{x}\right)$ as the class predictor of (x, y)

Java Wilderness
Intertwined Spirals

Koza's best-of-run:

```
(sin (iflte (iflte (+ Y Y) (+ X Y) (- X Y) (+ Y Y)) (* X X)
(sin (iflte (% Y Y) (% (sin (sin (% Y 0.30400002))) X) (% Y
0.30400002) (iflte (iflte (% (sin (% (% Y (+ X Y))
0.30400002)) (+ X Y)) (% X -0.10399997) (- X Y) (* (+
-0.12499994 -0.15999997) (- X Y))) 0.30400002 (sin (sin
(iflte (% (sin (% (% Y 0.30400002) 0.30400002)) (+ X Y))
(% (sin Y) Y) (sin (sin (sin (% (sin X) (+ -0.12499994
-0.15999997))))) (% (+ (+ X Y) (+ Y Y)) 0.30400002))))
(+ (+ X Y) (+ Y Y))))) (sin (iflte (iflte Y (+ X Y) (- X Y)
(+ Y Y)) (* X X) (sin (iflte (% Y Y) (% (sin (sin (% Y
0.30400002))) X) (% Y 0.30400002) (sin (sin (iflte (iflte
(sin (% (sin X) (+ -0.12499994 -0.15999997))) (% X
-0.10399997) (- X Y) (+ X Y)) (sin (% (sin X) (+
-0.12499994 -0.15999997))) (sin (sin (% (sin X) (+
-0.12499994 -0.15999997)))) (+ (+ X Y) (+ Y Y))))))) (%
Y 0.30400002)))))
```

Our best-of-run:

```
boolean isFirst(double x, double y) {
  double a, b, c, e;
  a = Math.hypot(x, y);  e = y;
  c = Math.atan2(y, b = x) +
    -(b = Math.atan2(a, -a))
    * (c = a + a) * (b + (c = b));
  e = -b * Math.sin(c);
  if (e < -0.0056126487018762772) {
    b = Math.atan2(a, -a);
    b = Math.atan2(a * c + b, x);  b = x;
    return false;
  }
  else
    return true;
}
```

Parameters

- Objective: summation of numbers in an input array
- Fitness: differences from actual sums on test inputs
- **Time limit**: 5000 backward branches

Code instrumentation

- Bytecode is instrumented with calls to time-limit check
- Before each backward branch and method invocation
- Robust and portable technique

Initialization

- “Weird” program that does **not** compute the sum

Array sum: array loop solution

```
public int sumlist(int list[]) {
  int sum  = 0;
  int size = list.length;
  for (int tmp = 0; tmp < list.length; tmp++) {
    size = tmp;
    sum = sum - (0 - list[tmp]);
  }
  return sum;
}
```

Array sum: List loop solution

```
int sumlist(List list) {
  int sum = 0;
  int size = list.size();
  for (Iterator iterator = list.iterator();
                         iterator.hasNext(); ) {
    int tmp = ((Integer) iterator.next())
                         .intValue();
    tmp = tmp + sum;
    if (tmp == list.size() + sum)
      sum = tmp;
    sum = tmp;
  }
  return sum;
}
```

Array sum: List-recursive solution

```
int sumlistrec(List list) {
  int sum = 0;
  if (list.isEmpty())
    sum = sum;
  else
    sum += ((Integer)list.get(0)).intValue() +
         sumlistrec(list.subList(1, list.size()));
  return sum;

}
```

Java Wilderness
The Tale of Alta Del

Parameters

- Objective: learn to play Tic-Tac-Toe
- Fitness: rounds won in single-elimination tournament

Initialization

- Negamax algorithm with α-β pruning and one of four (plausibly) insidious imperfections

Performance

- All imperfections are easily swept away (with interesting quirks!)

The Tic-Tac-Toe seed:

```
int negamaxAB(TicTacToeBoard board,
      int alpha, int beta, boolean save) {
  Position[] free = getFreeCells(board);
  // utility is derived from the number of free cells left
  if (board.getWinner() != null)
    alpha = utility(board, free);
  else if (free.length == 0)
    alpha = 0 save = false ;
  else for (Position move: free) {
    TicTacToeBoard copy = board.clone();
    copy.play(move.row(), move.col(),
                    copy.getTurn());
    int utility = - (removed) negamaxAB(copy,
                    -beta, -alpha, false save );
    if (utility > alpha) {
      alpha = utility;
      if (save)
        // save the move into a class instance field
        chosenMove = move;
      if ( alpha >= beta beta >= alpha )
        break;
    }
  }
  return alpha;
}
```

Conclusions

Completely **unrestricted** Java programs can be **evolved**
(via bytecode)

Loops and recursion are not a problem!

Extant (bad) Java programs can be **improved**

Future Work

- Actively searching for consistent bytecode segments during compatibility checks
- Class-level evolution: cross-method crossover, introduction of new methods
- Development of mutation operators
- Applying FINCH to additional hard problems
- Designing an IDE plugin to leverage FINCH for **software engineers**
- Applying FINCH to meta-evolution
- Automatic improvement of existing applications: the realm of **extant software**

“I believe that in about fifty years’ time it will be possible, to programme computers [. . .] to make them play the imitation game so well that an average interrogator will not have more than 70 per cent. chance of making the right identification after five minutes of questioning.”

A. M. Turing, “Computing machinery and intelligence,” Mind, 59(236), 433-460, Oct. 1950

"... despite its current widespread use, there was, within living memory, equal skepticism about whether compiled code could be trusted. If a similar change of attitude to evolved code occurs over time ..."

M. Harman, "Automated patching techniques: The fix is in," Communications of the ACM, 53(5), May 2010

We believe that in about fifty years' time it will be possible, to program computers. . . by means of evolution.

We believe that in about fifty years' time it will be possible, to program computers. . . by means of evolution.

Not merely *possible* but indeed *prevalent*.

We believe that in about fifty years' time it will be possible, to program computers. . . by means of evolution.

Not merely *possible* but indeed *prevalent*.

Turing was wrong—will we be?

We believe that in about fifty years' time it will be possible, to program computers. . . by means of evolution.

Not merely *possible* but indeed *prevalent*.

Turing was wrong—will we be?

To find out, please register for GPTP 2060.

References

M. Orlov and M. Sipper. Genetic programming in the wild: Evolving unrestricted bytecode. In G. Raidl et al., editors, *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, July 8–12, 2009, Montréal Québec, Canada*, pages 1043–1050, New York, NY, USA, July 2009. ACM Press. ISBN 978-1-60558-325-9. doi:10.1145/1569901.1570042.

M. Orlov and M. Sipper. Flight of the FINCH through the Java wilderness. *IEEE Transactions on Evolutionary Computation*, 2010. Conditionally accepted.


THE
END
IS
AT HAND
(1. PETER. 4. 7.)