Heuristic Approaches to Program Synthesis:
Genetic Programming and Beyond

Krzysztof Krawiec

Laboratory of Intelligent Decision Support Systems
Institute of Computing Science, Poznan University of Technology, Poznań, Poland

PhD Open, University of Warsaw, Jan 15-17 2015
Introduction
Krzysztof Krawiec, PhD, hab., Associate Professor at Poznan University of Technology

Computational Intelligence Group, part of Laboratory of Intelligent Decision Support Systems @ PUT
- The team: three postdocs, 3.5 PhDs
- Main research interests: program synthesis, evolutionary computation, pattern recognition, machine learning, games.

http://www.cs.put.poznan.pl/kkrawiec/
Objective: Provide state-of-the-art perspective on program synthesis, with emphasis on genetic programming.

Outline:
1. Program synthesis: problem definition, paradigms, challenges, why GP?
2. Evolutionary Computation 101
3. Genetic Programming: fundamentals, program representations, search operators, and more
4. Recent developments in GP: semantic and behavioral GP
5. In between: applications, case studies and success stories
Course organization

- No top-down structure – might came out too boring.
- In tune with http://en.wikipedia.org/wiki/Separation_of_concerns: a large number of relatively short, focused sections
- Questions and interactions welcome.
- Clickable hyperlinks in blue or red.

    if( more than 10% of people dozing off in the audience ) then goto Case study
Parts of the work presented here resulted from my cooperation with:

- Alberto Moraglio, University of Exeter
- Jerry Swan, University of Stirling
- Una-May O’Reilly, MIT
- Armando Solar-Lezama, MIT
- Wojciech Jaśkowski, Poznan University of Technology
- Tomasz Pawlak, Poznan University of Technology
- Bartosz Wieloch, Poznan University of Technology
What is program synthesis about?
Given:

- a programming language, i.e., implicitly a set of programs $P$
- a correctness predicate $Correct: P \rightarrow \mathbb{B}$,

find a program $p^*$ such that:

$$p^* = p \in P : Correct(p)$$

Note:

- Formulated roughly as in [Manna & Waldinger, 1980], yet earlier attempts present in AI
- In this purest form, program synthesis is a *search problem*
- Not to be confused with *automatic programming* (e.g., translating higher-level source code into machine code)
- We are primarily interested here in *automated* PS, but for reasons that will become clear later use ’PS’
Ways to solve a programming task

- State of the art: human programmer
  - Imperfect, unreliable, unsafe, ... yet getting better (?)
  - More and more power delegated to computers, entailing growing responsibility.

- Dijkstra’s dream: human programmer, providing proofs of correctness himself or using methods of formal verification
  - programs that are correct by construction [Dijkstra, nd]

- Dijkstra’s nightmare: [automatic] program synthesis
  - Programming cannot be automated, and as such will be always human-driven [Dijkstra, 1988]
  - Indeed: In the beginning, there is always human intent (user’s intent)
  - But: PS reached now further than Dijkstra probably dreamed (or rather bad-dreamed)
How to specify program correctness?

- Programs are not any formal objects: they are *functions* $I \rightarrow O$
- We consider a program correct if it *behaves* as expected, i.e., produces the desired output given input.
- Example of program specification [Manna & Waldinger, 1980]:

\[
\text{sqrt}(n) \Leftarrow \text{find } z \text{ such that } \text{integer}(z) \text{ and } z^2 \leq n \leq (z + 1)^2 \\
\text{where } \text{integer}(n) \text{ and } 0 \leq n
\]
Specifying program correctness

More generally:

\[
f(a) \iff \text{find } z \text{ such that } R(a, z)\]

where \( P(a) \)

where:

- \( a \) – program input
- \( z \) – program output
- \( P(a) \) – input condition (precondition, 'requires')
- \( R(a, z) \) – output condition (postcondition, 'ensures')
Specifying program correctness

Corresponding theorem to prove

\[ \forall a : P(a) \implies \exists z : R(a, z) \]

- \(a\) – program input
- \(z\) – program output
- \(P(a)\) – input condition (precondition, 'requires')
- \(R(a, z)\) – output condition (postcondition, 'ensures')

The proof must be constructive, i.e., must tell how to find \(z\) that satisfies the output condition \(R(a, z)\).
Curry-Howard correspondence

- One-to-one correspondence between CS and logic, i.e. between:
  - programs and proofs
  - types and propositions
- More extreme formulation:
  - Proofs in logic are programs in computer science.
  - Propositions in logic are types in computer science.
- The rules of logic are search operators in the space of proofs.
- Prolog ‘embodies’ the CH correspondence.
Program synthesis is the task of discovering an executable program from user intent expressed in the form of some constraints [Gulwani, 2010].

Program synthesis is the automatic translation of a specification into a program.
Main directions in program synthesis

As outlined in [Manna & Waldinger, 1980]:

- Deductive program synthesis
- Inductive programming
- Transformation of specification
- Heuristic approaches (including genetic programming)
Assumption: specification is complete
Program synthesis = theorem proving
Involves transformation rules, unification, resolution, and mathematical induction (for recursion)
Inductive programming

- Assumption: specification is incomplete
- Primary representative: inductive logic programming (ILP)
  - Synthesis of programs in logic, primarily in Prolog
  - Nowadays considered part of machine learning, mainly preoccupied with learning with relational data, knowledge discovery, data mining
- Involves transformation rules, unification, resolution, and mathematical induction (for recursion)
Inductive logic programming: An example

1. TRAINS GOING EAST

2. TRAINS GOING WEST

Source: [Flach & Lavrac, 2000]
Inductive logic programming: An example

east(t1).

hasCar(t1,c11). hasCar(t1,c12).
cshape(c11,rect). cshape(c12,rect).
clength(c11,short). clength(c12,long).
cwall(c11,single). cwall(c12,single).
croof(c11,no). croof(c12,no).
cwheels(c11,2). cwheels(c12,3).
hasLoad(c11,l11). hasLoad(c12,l12).
lshape(l11,circ). lshape(l12,hexa).
lnumber(l11,1). lnumber(l12,1).

hasCar(t1,c13). hasCar(t1,c14).
...
...

Exemplary hypothesis:
east(T):-hasCar(T,C),clength(C,short),croof(C,no)
We are talking about programs (methods, algorithms) that generate programs.

Note: generate, not manipulate (like, e.g., compilers)

However, this is not metaprogramming – this term is already reserved for a more technical purpose (e.g., Java program composes a shell script which is then executed).

Programs are in a sense not self-contained. Their meaning is externalized, i.e., dwells in the semantics of a given programming language.

Thus, what matters is program ‘behavior’, which can be captured by, e.g.,

- some external formalism (like proof of correctness),
- examples of input-output behavior.
Anticipated benefits of program synthesis

Programs that are:

- **Provably** correct, and thus
  - ‘globally reusable’,
  - certifiable

- Possibly also optimal with respect to non-functional requirements like
  - length, runtime, memory footprint, power consumption, etc.

- Free of malicious insets

- Cheap to produce
Challenges for formal approaches program synthesis

- Size of the proof space
  - Limited effectiveness of theorem provers
  - Consequence: lack of scalability (depending on the paradigm, upper limit of program length in the order of 20’s)

- Limited premises for prioritizing the search
  - Which transformation rule should be applied at a given stage of synthesis/proving process?

- Requirement of formal specification may be problematic.
  - Programmers not always ready/willing to provide such\(^1\)
    - end-users even less so (cf. end-user programming)
  - Describing the desired behaviors by means of examples can be more handy

- May require domain-specific knowledge
  - Each domain 'has its own maths’ that encodes knowledge about that domain;
  
  “we can automate programming only when we can identify a domain with such a well known body of knowledge, that existing implementations are produced (or may be produced) in a routine and obvious fashion” [Faitelson, 2010]

\(^1\)This changing, albeit slowly: see, e.g., design by contract, a methodology of software engineering.
Genetic programming

GP mitigates the challenges by:

- Relying on heuristic search algorithms to search the vast space of programs\(^2\),
- Abandoning (usually) formal specification in favor of examples of correct behavior (thus belongs to inductive programming),
- Naturally embracing domain-specific languages,
- Re-stating the program synthesis task as an *optimization problem*,
  - and thus: relaxing the concept of program correctness (!).
  - A partially incorrect program may be sometimes favored, for instance when advantageous in terms of non-functional properties.

Founded on the metaheuristic of evolutionary algorithms.

\(^2\)Heuristics are being used also in other approaches to program synthesis.
Evolutionary Computation 101
Evolutionary Computation (EC)

A branch of computational intelligence that deals with heuristic bio-inspired global search algorithms with the following properties:

- Operate on populations of candidate solutions
- Candidate solutions are encoded as genotypes
- Genotypes get decoded into phenotypes when evaluated by the fitness function $f$ being optimized.

  - Example: a candidate solution to a traveling salesperson problem is a permutation of cities (genotype), while its phenotype is a specific path of certain length.

- Attempt to find an optimal solution (an ideal) $p^*$:
  $$p^* = \arg\max_{p \in P} f(p)$$

(or conversely ‘arg min’), where $P$ is the considered space (search space) of candidate solutions (solutions for short).

- Note: an optimization, not a search problem!
Historically, one of meta-heuristics, along with tabu search, simulated annealing, etc.
Features of EC

- Generate-and-test approach
- Iterative
  - coarse-grained: generational EA,
  - fine-grained: steady-state EA
- Parallel global search
  - Not equivalent to parallel stochastic local search (SLS), particularly when crossover present
- Importance of crossover: a recombination operator that makes the solutions exchange certain elements (variable values, features)
  - Without crossover, EC boils down parallel stochastic local search
‘Black-box’ optimization ($f$’s dependency on the independent variables does not have to be known or meet any criteria)

Capable of ‘discovering’ both the global and local structure of the search space

- See: big valley hypothesis: good solutions are similar

No guarantees of finding a solution whatsoever

- Finding an optimum cannot be guaranteed, but in practice a well-performing suboptimal solution is often satisfactory.

Variables do not have to be explicitly defined
Variants of evolutionary algorithms

Well rooted in EC:
- Genetic algorithms (GA): discrete (binary) encoding
- Evolutionary strategies (ES): real-valued encoding
- Evolutionary programming (EP): not particularly popular nowadays, but historically one of the first approaches to EC
- Genetic Programming (GP)

Newer branches:
- estimation of distribution algorithms (EDA), generative and developmental systems (GDS), differential evolution, learning classifier systems, ...
- not strictly EC: particle swarm optimization (PSO), ant colony optimization (ACO),

Note:
- EC = Evolutionary Computation, the name of the domain
Major events of EC

- Genetic and Evolutionary Computation Conference (GECCO)
- IEEE Congress on Evolutionary Computation (CEC)
- EvoStar (Evo*)
- Parallel Problem Solving from Nature (PPSN)

Some facts:
- ACM SIGEVO group
- IEEE Task Forces
- Several dozens of thousands of publications (GP alone has almost 10,000)
- EC considered one of the three major branches of Computational Intelligence (Fuzzy Systems and Artificial Neural Networks being the other ones)
EAs are metaheuristics

Meta-heuristic = a generic algorithm template that can be adopted to a specific problem class (meta-) and is able to generate solutions of good/acceptable quality with limited computational resources (heuristic-)

Motivations:

- hardness of most nontrivial search and optimization problems,
- practical usefulness of good yet non-optimal solutions,
  - Example: a suboptimal solution (route) to a Traveling Salesperson Problem (TSP) that is only 5% worse than the optimal one may be good enough, given unpredictable factors that may interfere in the execution of that route.
- In other words: straining to achieve further (potentially miniscule) improvements may be technically/economically unjustified.
Convergence to good solutions may take some time …

Source: http://xkcd.com/720/

(Actually, some variants of EC maintain and manipulate infeasible solutions)
EAs is [getting] rigorous

- A growing body of theoretical results: schemata theorems, runtime analysis, first-hitting time proofs, performance bounds, fitness landscapes, ...
- Of course, always conditioned on some assumptions (e.g., unimodality, differentiability, ...) 
- Related milestones:
  - Schemata theorems: solutions’ components that occur in higher-than-average fit individuals tend to dominate population.
  - No-free-lunch (NFL) theorems [Wolpert & Macready, 1997], sharpened NFL theorems [Schumacher et al., 2001]
  - Elementary fitness landscapes [Whitley & Sutton, 2009]
Applications of EAs

Too numerous to cover (see, e.g., the Real-World-Application track of GECCO). A few examples:

- optimization of car chassis,
- design of analog and digital circuits,
- design of antennae,
- feature selection in machine learning tasks,
- optimization of wind turbine placement,
- designing spacecraft trajectories,
- sensor networks,
- and more.

EC’s strength: relative ease of adjusting to a specific problem: defining domain-specific search operators and fitness function is typically sufficient.
What is genetic programming?
Genetic programming

In a nutshell:

- A variant of EA where the genotypes represent *programs*, i.e., entities capable of reading in input data and producing some output data in response to that input.
- The candidate solutions in GP are being assembled from elementary entities called *instructions*.
- Most common program representation: expression trees.
  - Cardinality of search space large or infinite.
  - The number of all expression trees up to given size determined by the Catalan number.
Digression: Catalan numbers: http://oeis.org/A000108

Visit the OEIS Booth at the Joint Math Meetings in San Antonio Jan 10-13!

A000108 Catalan numbers: C(n) = binomial(2n,n)/(n+1) = (2n)!/(n!(n+1)!). Also called Segner numbers.
(Formerly M1459 N0577)

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452, 18367353072152, 69533550916004, 263747951750360, 1002242216651368, 3814986502092304

Offset 0, 3

Comments
The solution to Schroeder's first problem. A very large number of
combinatorial interpretations are known - see references, esp. Stanley,
Enumerative Combinatorics, Volume 2. This is probably the longest entry in
the OEIS, and rightly so.

Number of ways to insert n pairs of parentheses in a word of n+1 letters.
E.g., for n=2 there are 2 ways: ((ab)c) or (a(bc)); for n=3 there are 5
ways: ((ab)(cd)), (((ab)c)d), ((ab)(bc))d), ((bc)d)), (a(b(cd))).

What is genetic programming?
EA solves optimization problems. Program synthesis is a search problem. How to match them?

- Fitness function $f$ measures the similarity of the output produced by the program to the desired output, given as a part of task statement.
- The set of program inputs $I$, even if finite, is usually so large that running each candidate solution on all possible inputs becomes intractable.
- GP algorithms typically evaluate solutions on a sample $I' \subset I$, $|I'| \ll |I|$ of possible inputs, and fitness is only an approximate estimate of solution quality.
- The task is given as a set of fitness cases, i.e., pairs $(x_i, y_i) \in I \times O$, where $x_i$ usually comprises one or more independent variables and $y_i$ is the output variable.
City-block fitness function:

\[ f(p) = -\sum_i \| y_i - p(x_i) \|, \]  

(1)

where

- \( p(x_i) \) is the output produced by program \( p \) for the input data \( x_i \),
- \( \| \cdot \| \) is a metric (a norm) in the output space \( O \),
- \( i \) iterates over all fitness cases.
Main evolution loop (‘vanilla GP’)

1: procedure GeneticProgramming($f$, $\mathcal{I}$)  
   2:   $P \leftarrow \{ p \leftarrow \text{RandomProgram}(\mathcal{I}) \}$  
   3:   repeat  
   4:     for $p \in P$ do  
   5:       $p.f \leftarrow f(p)$  
   6:     end for  
   7:     $P' \leftarrow \emptyset$  
   8:     repeat  
   9:       $p_1 \leftarrow \text{TournamentSelection}(P)$  
  10:       $p_2 \leftarrow \text{TournamentSelection}(P)$  
  11:       $(o_1, o_2) \leftarrow \text{Crossover}(p_1, p_2)$  
  12:       $o_1 \leftarrow \text{Mutation}(o_1, \mathcal{I})$  
  13:       $o_2 \leftarrow \text{Mutation}(o_2, \mathcal{I})$  
  14:       $P' \leftarrow P' \cup \{o_1, o_2\}$  
  15:     until $|P'| = |P|$  
  16:     $P \leftarrow P'$  
  17:     until StoppingCondition($P$)  
  18:     return $\arg\max_{p \in P} p.f$  
  19:   end procedure

$\triangleright f$ - fitness function, $\mathcal{I}$ - instruction set
$\triangleright$ Initialize population
$\triangleright$ Main loop over generations
$\triangleright$ Evaluation
$\triangleright$ $p.f$ is a ‘field’ in program $p$ that stores its fitness
$\triangleright$ Next population
$\triangleright$ Breeding loop
$\triangleright$ First parent
$\triangleright$ Second parent
Search operators: Mutation

Mutation: replace a randomly selected subexpression with a new randomly generated subexpression.

1: function Mutation($p, \mathcal{I}$)
2:   repeat
3:     $s \leftarrow$ Random node in $p$
4:     $s' \leftarrow$ RandomProgram($\mathcal{I}$)
5:     $p' \leftarrow$ Replace the subtree rooted in $s$ with $s'$
6:   until Depth($p'$) $< d_{\text{max}}$  \hspace{1cm} \triangleright d_{\text{max}} \text{ is the tree depth limit}
7:   return $p'$
8: end function

Source: [Poli et al., 2008]
Search operators: Crossover

Crossover: exchange of randomly selected subexpressions (subtree swapping crossover).

1: function Crossover($p_1, p_2$)
2:     repeat
3:         $s_1 \leftarrow$ Random node in $p_1$
4:         $s_2 \leftarrow$ Random node in $p_2$
5:         ($p'_1, p'_2$) $\leftarrow$ Swap subtrees rooted in $s_1$ and $s_2$
6:     until Depth($p'_1$) $< d_{\text{max}} \land$ Depth($p'_2$) $< d_{\text{max}}$
   \hspace{1cm} $\triangleright d_{\text{max}}$ is the tree depth limit
7:         return ($p'_1, p'_2$)
8: end function

Source: [Poli et al., 2008]
Q: What is the most likely outcome of application of mutation/crossover to a viable program?

Hint:

But, however many ways there may be of being alive, it is certain that there are vastly more ways of being dead, or rather not alive. (The Blind Watchmaker [Dawkins, 1996])

A: Most applications of genetic operators are harmful

Yet, GP works. Why?

Mutation is random; natural selection is the very opposite of random (The Blind Watchmaker [Dawkins, 1996])

---

3 Turns out: In GP, quite many of them can be neutral (neutral mutations).
Exemplary run: Setup

A mini-run of GP applied to a symbolic regression problem (from: [Poli et al., 2008])

- **Objective:** Find a program whose output matches $x^2 + x + 1$ over the range $[-1, 1]$.
  - Such tasks can be considered as a form of regression.
  - As solutions are built by manipulating code (symbolic instructions), this is referred to as **symbolic regression**.

- **Fitness:** sum of absolute errors (City-block distance) for $x \in -1.0, -0.9, \ldots 0.9, 1.0$:

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>-1.0</th>
<th>-0.9</th>
<th>…</th>
<th>0</th>
<th>…</th>
<th>0.9</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_i$</td>
<td>1</td>
<td>0.91</td>
<td>…</td>
<td>1</td>
<td>…</td>
<td>2.71</td>
<td>3</td>
</tr>
</tbody>
</table>
Exemplary run: Setup

- Instruction set:
  - Nonterminal (function) set: +, -, % (protected division), and \( \times \); all operating on floats
  - Terminal set: \( \times \), and constants chosen randomly between -5 and +5
- Initial population: ramped half-and-half (depth 1 to 2; 50% of terminals are constants)
- Parameters:
  - population size 4,
  - 50% subtree crossover,
  - 25% reproduction,
  - 25% subtree mutation, no tree size limits
- Termination: when an individual with fitness better than 0.1 found
- Selection: fitness proportionate (roulette wheel) non elitist

What is genetic programming?
Initial population (population 0)

What is genetic programming?
Fitness values: $f(a)=7.7$, $f(b)=11.0$, $f(c)=17.98$, $f(d)=28.7$
Assume:

- $a$ gets reproduced
- $c$ gets mutated (at locus 2)
- $a$ and $d$ get crossed-over
- $a$ and $b$ get crossed-over

Note:

- All parents used; this in general does not have to be the case.
Population 0:

Population 1:

Individual $d$ in population 1 has fitness 0.
Summary of our first glimpse at GP
Specific features of GP

- The solutions evolving under the selection pressure of the *fitness function* are themselves *functions* (programs).
- GP operates on symbolic structures of *varying length*.
  - There are no variables for the algorithm to operate on (at least in the common sense).
- The program can be tested only on a limited number of fitness cases (tests).
A: Yes and no.

- In contrast to most EC methods that are typically placed in optimization framework, GP is by nature an inductive learning approach that fits into the domain of machine learning [Mitchell, 1997].

- As opposed to typical ML approaches, GP is very generic
  - Arbitrary programming language, arbitrary input and output representation

- The syntax and semantic of the programming language of consideration serve as means to provide the algorithm with prior knowledge
  - common sense knowledge, background knowledge, domain knowledge
In a broader context

A rather non-human approach to programming

(...) Artificial Intelligence as mimicking the human mind prefers to view itself as at the front line, whereas my explanation relegates it to the rearguard. (The effort of using machines to mimic the human mind has always struck me as rather silly: I’d rather use them to mimic something better.) [Dijkstra, 1988]

This pertains to certain differences between AI and CI:

- AI is (partially) engaged in research aiming at reproducing humans (in particular in research areas closer to cognitive science),
- CI focuses on intelligence as an emergent property (hence the prevailing presence of learning).

Claim (mine):

- GP embodies the ultimate goal of AI: to build a system capable of self-programming (adaptation, learning).
GP combines two powerful concepts marked in underline in the above definition:

1. **Representing candidate solutions as programs,** which in general can conduct any Turing-complete computation (e.g., classification, regression, clustering, reasoning, problem solving, etc.), and thus enable capturing solutions to any type of problems (whether the task is, e.g., learning, optimization, problem solving, game playing, etc.).

2. **Searching the space of candidate solutions using the ‘mechanics’ borrowed from biological evolution,** which is unquestionably a very powerful computing paradigm, given that it resulted in life on Earth and development of intelligent beings.
Why should GP be considered a viable approach to program synthesis?

Argument ‘from practice’:
- Human programmers do not rely (usually) on formal apparatus when programming.
- Neither they perform exhaustive search in the space of programs.
- Yet, they can program really well.

Other arguments:
- numerous ‘success stories’ concerning stochastic techniques in other domains, e.g.,
  - machine learning (bagging, random forests),
  - computer vision (random features)

Stochastic nature of a method does not preclude practical usefulness.
Genetic programming is a branch of computer science studying heuristic algorithms based on neo-Darwinian principles for synthesizing programs, i.e., discrete symbolic compositional structures that process data.

Consequences of the above definition:
- Heuristic nature of search.
- Symbolic program representation.
- Unconstrained data types.
- Unconstrained semantics.
- Input sensitivity and inductive character.
Risks involved?

```python
def getSolutionCosts(navigationCode):
    fuelStopCost = 15
    extraComputationCost = 8
    thisAlgorithmBecomingSkynetCost = 999999999
    waterCrossingCost = 45
```

GENETIC ALGORITHMS TIP:

ALWAYS INCLUDE THIS IN YOUR FITNESS FUNCTION

Origins of GP

Early work by:
- Similar ideas in early works of Schmidhuber [Schmidhuber, 1987]

http://www.genetic-programming.com/johnkoza.html
Exemplary GP run using ECJ
The task: synthesize a program that, given $x \in [-1, 1]$, returns an output equal to $y = x^5 - 2x^3 + x$ (symbolic regression)

Assumptions:

- available instructions: $+$, $-$, $\ast$, $/$, $\sin$, $\cos$, $\exp$, $\log$
- no constants
- no conditional statements nor loops
  - the program space is the space of arithmetic functions.
- set of 20 tests drawn randomly from $x \in [-1, 1]$
Exemplary run: Launch

Standard output:

```
java ec.Evolve -file ./ec/app/regression/quinticerc.params
...
Threads: breed/1 eval/1
Seed: 1427743400
Job: 0
Setting up
Processing GP Types
Processing GP Node Constraints
Processing GP Function Sets
Processing GP Tree Constraints
{-0.13063322286594392,0.016487577414659428},
{0.6533404396941143,0.1402200189629743},
{-0.03750634856569701,0.0014027712093654706},
...
{0.6602806044824949,0.13869498395598084},
Initializing Generation 0
Subpop 0 best fitness of generation: Fitness: Standardized=1.1303205 Adjusted=
Generation 1
Subpop 0 best fitness of generation: Fitness: Standardized=0.6804932 Adjusted=
...
```
Exemplary run: The result

The log file produced by the run (out.stat):

```
Generation: 0
Best Individual:
Subpopulation 0:
Evaluated: true
Fitness: Standardized=1.1303205 Adjusted=0.46941292 Hits=10
Tree 0:
(* (sin (* x x)) (cos (+ x x)))
```
```
Generation: 1
Best Individual:
Subpopulation 0:
Evaluated: true
Fitness: Standardized=0.6804932 Adjusted=0.59506345 Hits=7
Tree 0:
(* (rlog (+ (- x x) (cos x))) (rlog (- (cos (cos (* x x))) (- x x))))
```
```
....
```
Exemplary run

The log file produced by the run:

Best Individual of Run:
Subpopulation 0:
Evaluated: true
Fitness: Standardized=0.08413165 Adjusted=0.92239726 Hits=17

Tree 0:
(* (* (* (- (* (* (* (* x (sin x)) (rlog x)) (+ (+ (sin x) x) (- x x))) (exp (* x
(% (* (- (* (* (* (* x x) (rlog x)) (+ (+
(sinx) x) (- x x))) (exp (* x (sin x))))
(sinx)) (rlog x)) (exp (rlog x)))))) (sin
x)) (rlog x)) x) (cos (cos (* (* (- (* (*
(exp (rlog x)) (+ x (* (* (exp (rlog x))
(rlog x)) x))) (exp (* (* (* (- (exp (rlog
x)) x) (rlog x)) x) (sin (* x x)))) (sin
x)) (* x (% (* (- (* (* (* (* x x) (rlog
x)) (+ (+ x (+ (+ (sin x) x) (- x x))) (-
x x))) (exp (* x (sin x)))) (sin x)) (rlog
x)) (exp (rlog x)))) x))))

Exemplary GP run using ECJ
A more detailed view on GP
Design choices to be made, involving:

- population initialization, generating random programs (and subprograms),
- search operators,
  - many possibilities here, given that no ‘natural’ similarity metrics for program spaces exist,
- program representations (trees prevail in GP, but other representations are used as well)

... and the design choices characteristic for the more general domain of Evolutionary computation:

- generative vs. steady-state evolution,
- selection operators (fitness-proportional, tournament, ...)
- extensions: island models, estimation-of-distribution algorithms, multiobjective EAs, ...
Every stochastic search method needs some underlying sampling algorithm(s).

The distribution of randomly generated solutions is important, as it implies certain *bias* of the algorithm.

Problems:

- We don’t know the ‘ideal’ distribution of GP programs.
- Even if we knew it, it may be difficult to design an algorithm that obeys it.

The simplest initialization methods take care only of the syntax of generated programs.

- The parameter: the maximum depth of produced trees.
Initialization: *Full* method

- Specify the maximum tree height $h_{\text{max}}$.
- The *full* method for initializing trees:
  - Choose nonterminal nodes at random until $h_{\text{max}}$ is reached.
  - Then choose only from terminals.
Initialization: *Grow* method

- Specify the maximum tree height $h_{\text{max}}$.
- The *grow* method for initializing trees:
  - Choose nonterminal or terminal nodes at random until $h_{\text{max}}$ is reached
  - Then choose only from terminals.
Initialization: Comments

- $h_{\text{max}}$ is typically small (e.g., 5), because programs tend to grow with evolution anyway,
- If types are used, the choice of instructions has to be appropriately constrained
  - Typically, every instruction declares the set of accepted types for every input, and the type of output
  - The presence of types may make meeting size constraints difficult.
    - In an extreme case, generation of a syntactically correct program may be impossible!
- More sophisticated techniques exist, e.g., uniform sampling, see review in, e.g., [Poli et al., 2008].
  - An extension: *seeding* the population with candidate solutions that are believed to be good (domain knowledge required).
Alternative crossover operators

Even though the conventional GP crossover operators care only about program syntax, there are quite many of them. Examples:

- homologous crossover (detailed in next slides),
- uniform crossover (detailed in next slides),
- size-fair crossover,
- context-preserving crossover,
- headless chicken crossover (!),
- and more.

Why should crossover be considered important, particularly in GP?

- Programs are by nature *modular*.
- For instance, in purely functional programming, a piece of code ‘transplanted’ to a different location preserves its semantics (*referential transparency*, a.k.a. *closure* in GP).
- A GP run can be successful by the virtue of gradual accumulation of useful modules.
- Rich literature on modularity in evolution.
Homologous crossover for GP

- Earliest example: one-point crossover [Langdon & Poli, 2002]: identify a common region in the parents and swap the corresponding trees.
- The common region is the ‘intersection’ of parent trees.
Uniform crossover for GP

- Works similarly to uniform crossover in GAs
- The offspring is build by iterating over nodes in the common region and flipping a coin to decide from which parent should an instruction be copied [Poli & Langdon, 1998]
How to employ multiple operators for ‘breeding’?

How should the particular operators coexist in an evolutionary process? In other words:

- How should they be superimposed?
- What should be the ‘piping’ of particular breeding pipelines?
- A topic surprisingly underexplored in GP.

An example: Which is better:

```python
pop.subpop.0.species.pipe = ec.gp.koza.MutationPipeline
pop.subpop.0.species.pipe.num-sources = 1
pop.subpop.0.species.pipe.source.0 = ec.gp.koza.CrossoverPipeline
```

or

```python
pop.subpop.0.species.pipe.num-sources = 2
pop.subpop.0.species.pipe.source.0 = ec.gp.koza.CrossoverPipeline
pop.subpop.0.species.pipe.source.0.prob = 0.9
pop.subpop.0.species.pipe.source.1 = ec.gp.koza.MutationPipeline
pop.subpop.0.species.pipe.source.1.prob = 0.1
```
Challenges for GP
Bloat

- The evolving expressions tend to grow indefinitely in size.
- For tree-based representations, this growth is typically exponential[-ish]
- Evaluation becomes slow, algorithm stalls, memory overrun likely.
- One of the most intensely studied topics in GP: > 250 papers.

Bloat example: Average number of nodes per generation in a typical run of GP solving the Sextic problem \( x^6 - 2x^4 + x^2 \) (GP: dotted line)
Countermeasures for bloat

- Constraining tree height: discard the offspring that violates the upper limit on tree height
  - Surprisingly, theory shows that this can speed up bloat!
- Favoring small programs:
  - Lexicographic parsimony pressure: given two equally fit individuals, prefer (select) the one represented by a smaller tree.
- Bloat-aware operators: size-fair crossover.
Highly non-uniform distribution of program ‘behaviors’

Convergence of binary Boolean random linear functions (composed of AND, NAND, OR, NOR, 8 bits)

Source: [Langdon, 2002]
High cost of evaluation

- Running a program on multiple inputs can be expensive.
- Particularly for some types of data, e.g., images

Solutions:
- Caching of outcomes of subprograms
- Parallel execution of programs on particular fitness cases
- Bloat prevention methods

Right: Example from [Krawiec, 2004]. Synthesis of image analysis algorithms, where evaluation by definition incurs high computational cost.
Variants of GP
Strongly typed GP (STGP)

A way to incorporate prior knowledge and impose a structure on programs [Montana, 1993]

Implementation:

- Provide a set of types
- For each instruction, define the types of its arguments and outcomes
- Make the operators type-aware:
  - Mutation: substitute a random tree of a proper type
  - Crossover: swap trees of compatible\(^4\) types

\(^4\)‘Compatible’ = belonging to the same ‘set type’
For the problem of simple classifiers represented as decision trees:

**Classifier syntax:**
Classifier ::= Class_id  
Classifier ::= if_then_else(Condition, Classifier, Classifier)  
Condition ::= Input_Variable = Constant_Value

**Implementation in ECJ parameter files:**

Classifier syntax:
Classifier ::= Class_id  
Classifier ::= if_then_else(Condition, Classifier, Classifier)  
Condition ::= Input_Variable = Constant_Value

Implementation in ECJ parameter files:
gp.type.a.size = 3  
gp.type.a.0.name = class  
gp.type.a.1.name = var  
gp.type.a.2.name = const  
gp.type.s.size = 0  
gp.tc.size = 1  
gp.tc.0 = ec.gp.GPTreeConstraints  
gp.tc.0.name = tc0  
gp.tc.0.fset = f0  
gp.tc.0.returns = class  
gp.nc.size = 4  
gp.nc.0 = ec.gp.GPNodeConstraints  
gp.nc.0.name = ncSimpleClassifier  
gp.nc.0.returns = class  
gp.nc.0.size = 0  
gp.nc.1 = ec.gp.GPNodeConstraints  
gp.nc.1.name = ncCompoundClassifier  
gp.nc.1.returns = class  
gp.nc.1.size = 4  
gp.nc.1.child.0 = var  
gp.nc.1.child.1 = const  
gp.nc.1.child.2 = class  
gp.nc.1.child.3 = class  
gp.nc.2 = ec.gp.GPNodeConstraints  
gp.nc.2.name = ncVariable  
gp.nc.2.returns = var  
gp.nc.2.size = 0  
gp.nc.3 = ec.gp.GPNodeConstraints  
gp.nc.3.name = ncConstant  
gp.nc.3.returns = const  
gp.nc.3.size = 0
Motivation: Tree-like structures are not natural for contemporary hardware architectures

Program = a sequence of instructions

Data passed via registers

Directly portable to machine code, fast execution.

Natural correspondence to standard (GA-like) crossover operator.

Applications: direct evolution of machine code [Nordin & Banzhaf, 1995].
Example from [Krawiec, 2004]: the process of program interpretation:

and the corresponding data flow, including the initial and final register contents:
The best-known representative: Push and PushGP [Spector et al., 2004]

Very simple syntax: program ::= instruction | literal | ( program* )

No need to specify the number of registers

Natural possibility of implementing autoconstructive programs [Spector, 2010]

Includes certain features that make it Turing-complete (e.g., YANK instruction).

Simple cycle of program execution: pop an instruction from the exec stack and run it. The instruction will usually pop some data from data stack and push the results on the stack of the appropriate type.

The top element of a stack has the natural interpretation of program outcome.
Program:

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

Initial stack states:

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

Stack states after program execution:

BOOLEAN STACK: ( TRUE )
CODE STACK: ( ( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR ) )
FLOAT STACK: ( 9.3 )
INTEGER STACK: ( 6 )
### Push: Example 2

<table>
<thead>
<tr>
<th>Step</th>
<th>EXEC</th>
<th>Fitness case 1</th>
<th>Fitness case 2</th>
<th>Fitness case 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>INT</td>
<td>BOOL</td>
<td>INT</td>
</tr>
<tr>
<td>0</td>
<td>(* + &lt;)</td>
<td>1 3 4 5</td>
<td>( )</td>
<td>2 2 4 2</td>
</tr>
<tr>
<td>1</td>
<td>(+ &lt;)</td>
<td>3 4 5</td>
<td>( )</td>
<td>4 4 2</td>
</tr>
<tr>
<td>2</td>
<td>(&lt;)</td>
<td>7 5</td>
<td>( )</td>
<td>8 2</td>
</tr>
<tr>
<td>3</td>
<td>( )</td>
<td>( )</td>
<td>(F)</td>
<td>( )</td>
</tr>
</tbody>
</table>

More details: [http://hampshire.edu/lspector/push3-description.html](http://hampshire.edu/lspector/push3-description.html)
Grammatical Evolution (GE)

- Grammatical Evolution: The grammar of the programming language of consideration is given as input to the algorithm. [Ryan et al., 1998]
- Individuals encode the choice of productions in the derivation tree (which of available alternative production should be chosen, modulo the number of productions available at given step of derivation).

Grammar:
<e> ::= <e><o><e> | <v>
<o> ::= + | -
<v> ::= X | Y

Chromosome:
12, 3, 7, 15, 9, 36, 14
Other variants of GP

- Graph-based GP
  - Motivation: standard GP cannot reuse subprograms within a single program
  - Example: Cartesian Genetic Programming [Miller, 1999]

- Multiobjective GP. The extra objectives can:
  - Come with the problem
  - Result from GP’s specifics: e.g., use program size as the second (minimized) objective
  - Be associated with different tests (e.g., feature tests [Ross & Zhu, 2004])

- Developmental GP (e.g., using Push)

- Probabilistic GP (a variant of EDA, Estimation of Distribution Algorithms):
  - The algorithm maintains a probability distribution $P$ instead of a population
  - Individuals are generated from $P$ ‘on demand’
  - The results of individuals’ evaluation are used to update $P$
Simple EDA-like GP: PIPE

Probabilistic Incremental Program Evolution [Salustowicz & Schmidhuber, 1997]
Applications of GP
GP produced a number of solutions that are human-competitive, i.e., a GP algorithm automatically solved a problem for which a patent exists [Koza et al., 2003b].

A recent award-winning work has demonstrated the ability of a GP system to automatically find and correct bugs in commercially-released software when provided with test data [Arcuri & Yao, 2008].

GP is one of leading methodologies that can be used to ‘automate’ science, helping the researchers to find the hidden complex patterns in the observed phenomena [Schmidt & Lipson, 2009].
(...) Entries were solicited for cash awards for human-competitive results that were produced by any form of genetic and evolutionary computation and that were published

http://www.genetic-programming.org/combined.php
The conditions to qualify:
(A) The result was patented as an invention in the past, is an improvement over a patented invention, or would qualify today as a patentable new invention.
(B) The result is equal to or better than a result that was accepted as a new scientific result at the time when it was published in a peer-reviewed scientific journal.
(C) The result is equal to or better than a result that was placed into a database or archive of results maintained by an internationally recognized panel of scientific experts.
(D) The result is publishable in its own right as a new scientific result — independent of the fact that the result was mechanically created.
(E) The result is equal to or better than the most recent human-created solution to a long-standing problem for which there has been a succession of increasingly better human-created solutions.
(F) The result is equal to or better than a result that was considered an achievement in its field at the time it was first discovered.
(G) The result solves a problem of indisputable difficulty in its field.
(H) The result holds its own or wins a regulated competition involving human contestants (in the form of either live human players or human-written computer programs).
2004: Jason D. Lohn Gregory S. Hornby Derek S. Linden, NASA Ames Research Center, 
*An Evolved Antenna for Deployment on NASA’s Space Technology 5 Mission*

Automatically finding patches using genetic programming: A Genetic Programming Approach to Automated Software Repair

```c
void zunebug(int days) {
    int year = 1980;
    while (days > 365) {
        if (isLeapYear(year)) {
            if (days > 366) {
                days -= 366;
                year += 1;
            }
        } else {
        }
    } else {
        days -= 365;
        year += 1;
    }
    printf("current year is %d\n", year);
}
```

Successfully fixes a 'New Year’s bug' in Microsoft’s MP3 player Zune.
2008: Lee Spector, David M. Clark, Ian Lindsay, Bradford Barr, Jon Klein
   *Genetic Programming for Finite Algebras*

2010: Natalio Krasnogor Paweł Widera Jonathan Garibaldi
   *Evolutionary design of the energy function for protein structure prediction*

2011: Achiya Elyasaf Ami Hauptmann Moshe Sipper
   *GA-FreeCell: Evolving Solvers for the Game of FreeCell*
Application: Bug fixing

GenProg [Le Goues et al., 2012]:

- Maintains a population candidate *repairs* as sequences of *edits* to software source code.
- Each candidate is applied to the original program to produce a new program, which is evaluated using test suites.
- Fitness = number of tests passed.
- Termination = a candidate repair is found that retains all required functionality *and* fixes the bug.
- Does not require special code annotations or formal specifications, and applies to unmodified legacy software.
- Won IFIP TC2 Manfred Paul Award (2009), and Humies (twice)
Application: Bug fixing

Economic aspects: https://www.youtube.com/watch?v=Z3itydu_rjo

For embedded devices: https://www.youtube.com/watch?v=95N0Yokm6Bk

Follow-ups/related:

- reduction of the power consumption of software
- assembly and binary repairs of embedded systems.
- automated repair of exploits in binary code of a network router
  - exploits allowing unauthenticated users to change administrative options and completely disable authentication across reboots
  - https://github.com/eschulte/netgear-repair
Other applications

- Learning game strategies [Jaskowski et al., 2008].
- See [Poli et al., 2008] for an extensive review of GP applications.
Assessment of GP techniques
Criteria for assessing the quality of GP-evolved solutions

Criteria for assessing GP algorithms:
- success rate (percentage of evolutionary runs ended with success)
- time-to-success (can be $\infty$)
- error of the best-of-run individual

Criteria for assessing programs obtained with GP:
- error rate (percentage of tests passed)
- program size (number of instructions)
- execution time
- transparency (readability)
GP Benchmarks

A community-wide initiative to set assessment standards in GP.

http://gpbenchmarks.org/

Symbolic Regression
Tower [Vladislavleva et al., 2009] ...

Boolean Functions
N-Multiplexer, N-Majority, N-Parity [Koza, 1992b]
Generalised Boolean Circuits [Harding et al., 2010, Yu, 2001]
Digital Adder [Walker et al., 2009]
Order [Durrett et al., 2011]
Digital Multiplier [Walker et al., 2009]
Majority [Durrett et al., 2011]

Classification
mRNA Motif Classification [Langdon et al., 2009]
DNA Motif Discovery [Langdon et al., 2010]
Intrusion Detection [Hansen et al., 2007]
Protein Classification [Langdon & Banzhaf, 2008]
Intertwined Spirals [Koza, 1992b]
... and more ....

Predictive Modelling
Mackey-Glass Chaotic Time Series [Langdon & Banzhaf, 2005]
Financial Trading [Dempsey et al., 2006]
Sunspot Prediction [Koza, 1992b]
GeneChip Probe Performance [Langdon & Harrison, 2008]
Prime Number Prediction [Walker & Miller, 2007]
Drug Bioavailability [Silva & Vanneschi, 2010]
Protein Structure Classification [Widera et al., 2010]
Time Series Forecasting [Wagner et al., 2007]

Path-finding and Planning
Physical Travelling Salesman [Lucas, 2012b]
Artificial Ant [Koza, 1992b]
Lawnmower [Koza, 1994]
Tartarus Problem [Cuccu & Gomez, 2011]
Maximum Overhang [Paterson et al., 2008]
Circuit Design [McConaghy, 2011]

Control Systems
Chaotic Dynamic Systems Control [Lones et al., 2010]
Pole Balancing [Nicolau et al., 2010]
Truck Control [Koza, 1992a]
Game-Playing
TORCS Car Racing [torcs, 2012]
Ms PacMan [Galván-López et al., 2010]
Othello [Lucas, 2012a]
Chessboard Evaluation [Sipper, 2011]
Backgammon [Sipper, 2011]
Mario [Togelius et al., 2009]
NP-Complete Puzzles [Kendall et al., 2008]
Robocode [Sipper, 2011]
Rush Hour [Sipper, 2011]
Checkers [Sipper, 2011]
Freecell [Sipper, 2011]

Dynamic Optimisation
Dynamic Symbolic Regression [O’Neill et al., 2008]
Dynamic Scheduling [Jakobović & Budin, 2006]

Traditional Programming
Sorting [Kinnear, Jr., 1993a]
Semantic GP
Fitness bottleneck problem:
The complex effects\(^{(1)}\) of program execution on multiple examples\(^{(2)}\) are combined into one scalar value (fitness).

Consequences:
- Loss of information.
- Compensation of performance on particular tests (examples).
- Search algorithm cannot reverse-engineer the compressed information.

Why do we stick to this design? **There are no principal reasons to maintain the bottleneck.**

\(^{(2)}\) motivates **semantic GP**
\(^{(1)}\) motivates **behavioral evaluation**
Program semantics in GP

Program semantics = the vector of outputs produced by a program for the training examples (a.k.a. sampling semantics).

Program $p$:  

\[
\begin{array}{c}
\times \\
+ \\
\times \\
\times \\
\times
\end{array}
\]

Semantics ($p$) = [0.5, 2.0, 4.5, 8.0]

Can been used for:
- designing initialization operators,
- diversity maintenance,
- designing search operators.
The fitness functions used in GP are usually metrics, like:

- **Hamming distance**: \(|\{p(x_i) \neq y_i\}|\)
- **Manhattan distance**: \(\sum_i |p(x_i) - y_i|\)
- **Euclidean distance**: \(\sum_i |p(x_i) - y_i|^2\)

Given \(n\) fitness cases, such a fitness function measures, in the \(n\)-dimensional semantic space, the distance of program semantics from the point that defines the desired output of program (\(y_i\)s above, a.k.a. target, \(t\) in the next slides).

Thus, the semantic space is a *metric space*, and fitness landscape forms a *unimodal cone*. 
Semantic space ($t$ - the target, i.e., vector of desired outputs):

- (Euclidean metric)
  - The (often difficult) program synthesis task becomes trivial in semantic space (unimodal and convex fitness landscape).
  - Search operators with attractive guarantees can be designed.

- (City-block metric)
**Geometric crossover**

- A *geometric offspring* $o$:

$$||o, p_1|| + ||o, p_2|| = ||p_1, p_2|| \quad (2)$$

- Crossover operator that produces geometric offspring is *geometric crossover* (a.k.a. topological crossover).

- Produce offspring that inherit some aspects of *behavior* from the parents.
  - Offspring’s semantics is ‘in between’ the parents in the semantic space.

- The segment connecting the parents embraces all semantics (and, indirectly, programs) that are (semantically) as similar as possible to both parents.

- The **big question**: can we design efficient search operators that are geometric?
For some domains, exactly geometric effect can be attained by purely syntactic manipulations [Moraglio et al., 2012].

- A general method to derive exact semantic geometric crossovers and mutations for different problem domains that search directly the semantic space

\[
\begin{align*}
T_1 \times T_2 & \xrightarrow{GX_{SD}} T_3 \\
O(T_1) \times O(T_2) & \xrightarrow{GX_D} O(T_3)
\end{align*}
\]

- Top: semantic geometric crossover $GX_{SD}$ on genotypes (e.g., trees),
- Bottom: Geometric crossover ($GX_D$) operating on the phenotypes (i.e., output vectors) induced by the genotype-phenotype mapping $O$.
- It holds that for any $T_1$, $T_2$ and $T_3 = GX_{SD}(T_1, T_2)$ then $O(T_3) = GX_D(O(T_1), O(T_2))$. 
For boolean problems

**Definition**

Given two parent functions $T_1, T_2 : \{0, 1\}^n \rightarrow \{0, 1\}$, the recombination SGXB returns the offspring boolean function $T_3 = (T_1 \land TR) \lor (\overline{TR} \land T_2)$ where $TR$ is a randomly generated boolean function.

**Theorem**

$SGXB$ is a semantic geometric crossover for the space of boolean functions with fitness function based on Hamming distance, for any training set and any boolean problem.
Example

- Left: Semantic Crossover scheme for Boolean Functions;
- Centre: Example of parents (T1 and T2) and random mask (TR);
- Right: Offspring (T3) obtained by substituting T1, T2 and TR in the crossover scheme and simplifying.
For real-valued programs

**Definition**

Given two parent functions $T_1, T_2 : \mathbb{R}^n \rightarrow \mathbb{R}$, the recombinations SGXE and SGXM return the real function $T_3 = (T_1 \cdot TR) + ((1 - TR) \cdot T_2)$ where $TR$ is a random real constant in $[0, 1]$ (SGXE), or a random real function with codomain $[0, 1]$ (SGMX).

**Theorem**

SGXE and SGXM are semantic geometric crossovers for the space of real functions with fitness function based on Euclidean and Manhattan distances, respectively, for any training set and any real problem.
### Experimental results: Boolean problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>Hits %</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparator6</td>
<td>GP: 80.2 ± 3.8</td>
<td>Shanghai: 3.5</td>
</tr>
<tr>
<td>Comparator8</td>
<td>GP: 80.3 ± 2.8</td>
<td>Shanghai: 4.9</td>
</tr>
<tr>
<td>Comparator10</td>
<td>GP: 82.3 ± 4.3</td>
<td>Shanghai: 95.3</td>
</tr>
<tr>
<td>Multiplexer6</td>
<td>GP: 70.8 ± 3.3</td>
<td>Shanghai: 94.7</td>
</tr>
<tr>
<td>Multiplexer11</td>
<td>GP: 76.4 ± 7.9</td>
<td>Shanghai: 88.8</td>
</tr>
<tr>
<td>Parity5</td>
<td>GP: 52.9 ± 2.4</td>
<td>Shanghai: 56.3</td>
</tr>
<tr>
<td>Parity6</td>
<td>GP: 50.5 ± 0.7</td>
<td>Shanghai: 55.4</td>
</tr>
<tr>
<td>Parity7</td>
<td>GP: 50.1 ± 0.2</td>
<td>Shanghai: 51.7</td>
</tr>
<tr>
<td>Parity8</td>
<td>GP: 50.1 ± 0.2</td>
<td>Shanghai: 50.6</td>
</tr>
<tr>
<td>Parity9</td>
<td>GP: 50.0 ± 0.0</td>
<td>Shanghai: 50.2</td>
</tr>
<tr>
<td>Parity10</td>
<td>GP: 50.0 ± 0.0</td>
<td>Shanghai: 50.0</td>
</tr>
<tr>
<td>Random5</td>
<td>GP: 82.2 ± 6.6</td>
<td>Shanghai: 90.9</td>
</tr>
<tr>
<td>Random6</td>
<td>GP: 83.6 ± 6.6</td>
<td>Shanghai: 93.0</td>
</tr>
<tr>
<td>Random7</td>
<td>GP: 85.1 ± 5.3</td>
<td>Shanghai: 92.9</td>
</tr>
<tr>
<td>Random8</td>
<td>GP: 89.6 ± 5.3</td>
<td>Shanghai: 93.7</td>
</tr>
<tr>
<td>Random9</td>
<td>GP: 93.1 ± 3.7</td>
<td>Shanghai: 95.4</td>
</tr>
<tr>
<td>Random10</td>
<td>GP: 95.3 ± 2.3</td>
<td>Shanghai: 96.2</td>
</tr>
<tr>
<td>Random11</td>
<td>GP: 96.6 ± 1.6</td>
<td>Shanghai: 97.3</td>
</tr>
<tr>
<td>True5</td>
<td>GP: 100.0</td>
<td>Shanghai: 100.0</td>
</tr>
<tr>
<td>True6</td>
<td>GP: 100.0</td>
<td>Shanghai: 100.0</td>
</tr>
<tr>
<td>True7</td>
<td>GP: 100.0</td>
<td>Shanghai: 100.0</td>
</tr>
<tr>
<td>True8</td>
<td>GP: 100.0</td>
<td>Shanghai: 100.0</td>
</tr>
</tbody>
</table>

**GP:** conventional GP, **SSH:** semantic stochastic hill climber, **SGP:** semantic geometric GP
Experimental results: real-valued programs

<table>
<thead>
<tr>
<th>Problem</th>
<th>GP</th>
<th>SCP:</th>
<th>SSHC</th>
<th>SGP:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>avg</td>
<td>sd</td>
<td>avg</td>
<td>sd</td>
</tr>
<tr>
<td>Polynomial3</td>
<td>79.9</td>
<td>23.1</td>
<td>100.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Polynomial4</td>
<td>60.5</td>
<td>27.6</td>
<td>99.9</td>
<td>0.9</td>
</tr>
<tr>
<td>Polynomial5</td>
<td>40.7</td>
<td>21.6</td>
<td>100.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Polynomial6</td>
<td>37.5</td>
<td>23.4</td>
<td>100.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Polynomial7</td>
<td>30.7</td>
<td>18.5</td>
<td>100.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Polynomial8</td>
<td>34.7</td>
<td>16.0</td>
<td>99.5</td>
<td>2.0</td>
</tr>
<tr>
<td>Polynomial9</td>
<td>20.7</td>
<td>13.2</td>
<td>100.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Polynomial10</td>
<td>25.7</td>
<td>16.7</td>
<td>99.4</td>
<td>1.7</td>
</tr>
</tbody>
</table>

GP: conventional GP, SSHC: semantic stochastic hill climber, SGP: semantic geometric GP
Conclusions:

- Semantic of a GP program is a means for getting better insight into its properties.
- ‘Semantic setting’ implies certain properties of the fitness landscape (convexity, unimodality).
- Search operators (approximate or exact) can be designed that exploit such properties.
- Semantic GP an be seen as ‘multiobjectivization’ of a problem.
- The challenge: offspring size.

New results

- runtime analysis for GSGP,
- bounds on fitness improvement/deterioration in GSGP (in review)

Work in progress:

- Exploitation of semantic properties for problem decomposition (module detection).
- Other semantic properties worth considering, e.g., equidistance.
Behavioral GP and search drivers
- Takes semantic GP even further
- The rationale: The final outcomes of program execution reveal only a fraction of the actual program’s activity.
- More detailed information can be obtained by tracing the entire program execution.
- This allows detecting and reuse of potentially useful program components.
Two stages required:
- Sort the array
- Locate the central element.

Most nontrivial tasks require such stage-wise problem decomposition.

The sorted list is a desired intermediate computation state.

Human programmers can define such states a priori.

Can we determine such states in advance?

Can we help evolution in detecting and promoting the desired intermediate computation states?
**STANDARD GP:**

Program input | Execute program $p$ on each input $x_i$ independently | Program output | Fitness | Desired output
---|---|---|---|---
$x_1$ | $p$ | $p(x_1)$ | - | $y_1$
$x_2$ | $p$ | $p(x_2)$ | - | $y_2$
$x_3$ | $p$ | $p(x_3)$ | - | $y_3$
$x_4$ | $p$ | $p(x_4)$ | - | $y_4$
$x_5$ | $p$ | $p(x_5)$ | - | $y_5$
Standard GP

Program input

$x$  $\rightarrow$  Program execution  $\rightarrow$  Actual program output  $\rightarrow$  Desired output  $\rightarrow$  Program error

Program error

Behavioral GP and search drivers
Pattern-guided GP

Program input

\[ x \]

Program execution

\[ \rightarrow \]

Actual program output

\[ p(x) \]

- \[ y \]

Desired output

\[ f \]

Program error

\[ e \]

Classifier error

\[ c \]

Classifier complexity (size)

Training set

ML classifier

\[ s_1(x) \]

\[ s_2(x) \]

\[ \ldots \]

Program trace

\[ \textbf{Black:} \] Conventional GP

\[ \textbf{Green:} \] PANGEA [Krawiec & Swan, 2013]
Example (nominal domain, tree-based GP)

**Problem**

<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

**GP Individual**

```
           -
        /   \
  *       +   
   \
 x1      x2
  4     5
  1  1
  9  4
```

**ML dataset**

<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

**Decision tree**

- x4
  - = 1
  - ≠ 1
  - y = 2
  - = 2
  - ≠ 2
  - y = 3
  - y = 4

**Evaluation:**

- Program error: 10
- Classifier error: 0 examples
- Classifier complexity: 5 nodes
Behavioral GP [Krawiec & O’Reilly, 2014]

Key ingredients:
- Multiobjective evaluation and selection
- Archiving of promising subprograms
- Mutation operator supplied by subprograms from the archive
- Immense improvements of performance [Krawiec & O’Reilly, 2014]
Birds-eye view on program synthesis
“Dimensions in program synthesis” [Gulwani, 2010], a rather complete overview of
- applications,
- problems,
- solution spaces, and
- approaches to program synthesis (as a whole, not only GP).

In particular, identifies new application areas of potential interest also for GP.
In particular:

- *Bitvector* algorithms

These algorithms

(...*) typically describe some plausible yet unusual operation on integers or bit strings that could easily be programmed using either a longish fixed sequence of machine instructions or a loop, but the same thing can be done much more cleverly using just four or three or two carefully chosen instructions whose interactions are not at all obvious until explained or fathomed” Hackers Delight[Warren, 2002]

- mutual exclusion algorithms, i.e., algorithms that guarantee mutually exclusive access to critical sections
Problem formulation: given a program $p : I \rightarrow O$ that implements an injection, synthesize a program $p' : O \rightarrow I$.

Common design pattern in software engineering:

- compression/decompression,
- encryption/decryption,
- serialization/deserialization,
- insert/delete operations on data structures,
- transactional memory rollback,

What is doable here?

- The approach by [Srivastava et al., 2010] can synthesize inverses for compressors (e.g., LZ77), packers (e.g., UUEncode), and arithmetic transformers (e.g., image rotations).
- Length of inverse programs: 5 .. 20 lines of code, synthesized within a minute.
Applications: Program understanding

Examples:
- explaining a complicated program written in a low-level language in terms of a high-level language
- malware deobfuscation
- maintenance of poorly documented software.
Many end-users need some form of 'programmatic automation' of certain tasks, like commodity traders, graphic designers, chemists, human resource managers, finance pros, ...

- These users typically lack the technical skills to program from scratch.

General Purpose Programming Assistance

- Synthesis can be used to find tricky/mundane implementation details after human insight has been expressed in the form of a partial program [65]

- Automated Debugging

See also: *Flash fill* [Gulwani et al., 2012]
The role of types
Motivation: types reveal the underlying semantics [Zoltan and Swan, 2014]

Engages also type systems: other formulation: to prove a theorem, a type must be constructed, and and a value of that type has to be found.

An interesting related observation: For many types, there are no values.

- Example: given two types $a$ and $b$, there is in general no function $a \to b$.
- Only when some assumptions about $a$ and $b$ are made, such a function can be constructed (and thus the associated type $a \to b$ does exist).
Wadler, 1989:

*Write down the definition of a polymorphic function on a piece of paper. Tell me its type, but be careful not to let me see the function’s definition. I will tell you a theorem that the function satisfies* [Wadler, 1989].

Example:

\[ f : \text{List}[T] \rightarrow \mathbb{N} \]

implies that \( f \) **has to be** a function of list length.

See: *Theorems for free* [Wadler, 1989]
Another example

\[ f : \text{List}[T] \rightarrow \text{List}[T] \]

**From this follows**, that for all types \( T \) and \( T' \) and every total function \( a : T \rightarrow T' \),

\[ a^* \circ f_T = f_{T'} \circ a^* \]

where \( a^* \) is a 'map \( a' \), and \( f_T \) is an instance of \( f \) for type \( T \).

In other words, it is irrelevant whether we

- first apply \( a \) to every element of the list and then apply \( f_T \) to the resulting list,
- or the reverse: first apply \( f_T \) to the list and then apply \( a \) to every element of the resulting list.

Examples:

- \( f = \text{reverse}, a = \text{asciiCode} \)
- \( f = \text{tail}, a = \text{inc} \)
Related results (selected)

- The Coq proof assistant
  - Computer-checked proof of the four-color theorem
- Formal verification of some commercial software (Coq)
  - Certified programs
- For more, see: [Wadler, 2014]
Case studies
Case study 1: Evolution of temperature models

Based on:

Joint work with:
- Institute of Computing Science, Poznan University of Technology, Poznan, Poland
- Institute for Agricultural and Forest Environment, Polish Academy of Sciences, Poznan, Poland and Potsdam Institute for Climate Impact Research, Potsdam, Germany

Link to slides
Case study 2: Evolution of features for object detection in aerial imagery

Based on:

[Link to slides]
Case study 3: Evolution of detectors of anatomical structures

Based on:

Link to slides
Case study 4: Evolution of algebraic terms

<table>
<thead>
<tr>
<th>$a_1$</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

$t^A(x, y, z) = \begin{cases} 
  x & \text{if } x \neq y \\
  z & \text{if } x = y 
\end{cases}$

$m(x, x, y) = m(y, x, x) = y$

- Ternary domain: inputs and outputs from \{0, 1, 2\}.
- Only one binary instruction, defining the underlying algebra (a).
- The discriminator term task(s): synthesize an expression that accepts three inputs $x, y, z$ and is semantically equivalent to the one shown in (b).
  - $3^3 = 27$ fitness cases (tests).
- The Malcev term tasks(s): evolve a ternary term that satisfies (c)
  - Specifies program output only for some combinations of inputs: the desired value for $m(x, y, z)$, where $x, y,$ and $z$ are all distinct, is not determined.
  - Only 15 fitness cases (tests)
- [Spector et al., 2008] evolved the smallest terms to date, previously unknown to mathematicians.
Case study 5: Evolution of job acceptance conditions

Overall idea:

- Take an exact search algorithm (e.g., branch-and-bound, B&B)
- The actual efficiency of B&B depends on how it prioritizes the search, i.e., which search directions/nodes are visited first.
- Use GP to evolve a heuristic function that captures the properties of the specific problem instance and prefers the states that are likely to end up in
- Successfully applied in job shop scheduling [Nguyen et al., ]
Software packages
Software packages

- Evolutionary Computation in Java (George Mason University, DC)
  - Generic software framework for EA, well-prepared to work with GP
  - cs.gmu.edu/~eclab/projects/ecj/
- EpochX (University of Kent, UK), also in Java
  - http://www.epochx.org/
- Disciplus™ (RML Technologies)
  - http://www.rmltech.com/
- FlexGP (CSAIL, MIT), Java
  - http://flexgp.github.io/gp-learners/
ECJ, Evolutionary Computation in Java, 
http://cs.gmu.edu/~eclab/projects/ecj/

Probably the most popular freely available framework for EC, with a strong support for GP

Licensed under Academic Free License, version 3.0

As of Jan 2015: version 22.

Many other libraries integrate with ECJ.
Selected ECJ features

- GUI with charting
- Platform-independent checkpointing and logging
- Hierarchical parameter files
- Multithreading
- Mersenne Twister Random Number Generators (compare to: http://www.alife.co.uk/nonrandom/)
- Abstractions for implementing a variety of EC forms.
- Prepared to work in a distributed environment (including so-called island model)

- GP Tree Representations
- Set-based Strongly-Typed Genetic Programming
- Ephemeral Random Constants
- Automatically-Defined Functions and Automatically Defined Macros
- Multiple tree forests
- Six tree-creation algorithms
- Extensive set of GP breeding operators
- Grammatical Encoding
- Eight pre-done GP application problem domains (ant, regression, multiplexer, lawnmower, parity, two-box, edge, serengeti)
• EpochX (University of Kent, UK), also in Java
• Ready-to-run examples:
• Examples, including the Artificial Ant benchmark:
• Has been used to evolve programs with loops [Castle & Johnson, 2012]
GP in R

- A package in R (The R Project for Statistical Computing) that facilitates symbolic regression and more.
- Relies on the ‘natural reflection’ in R (R is an interpreted language)

http://cran.r-project.org/web/packages/gpr/index.html
Exemplary implementation of GP framework in **Mathematica**

```mathematica
(* Steady-State Evolutionary Algorithm  
   with Semantic Operators on Boolean Functions *)

n = 8; (* Number of Variables *)
k = Round[Sqrt[2^"n"]]; (* Population Size *)
v = Table[Symbol["x" <> ToString[i]], {i, n}]; (* Vector of Variables *)
pop = Table[RandomFunction[RandomInteger[2, 2^n - 1], v], {k}]; (* Initial Population of Random Functions *)
fpop = Table[Total[BooleanTable[pop[[i]]]], {i, k}]; (* Fitness of Initial Population *)
fbest = Max[fpop]; (* Fitness Best Individual *)
posbest = Position[fpop, fbest][[1, 1]]; (* Position Best Individual *)
fworst = Min[fpop]; (* Fitness Worst Individual *)
fworst = Position[fpop, fworst][[1, 1]]; (* Position Worst Individual *)
For[i = 0, fbest < 2^n, i++, (* Is current Solution the Optimum? *)
Print[i, " ", fbest, ", ", Length[fpop[[posbest]]]];
pl = pop[[RandomInteger[{1, k}]]]; (* select parents uniformly at random in the population *)
  (* Print[pl]; *)
p2 = pop[[RandomInteger[{1, k}]]]; (* Print[p2]; *)
r = RandomInteger[2**(2^n - 1), v]; (* random recombination mask *)
  (* Print[r]; *)
o = (pl && r) || (p2 && !r); (* semantic crossover *)
  (* Print[o]; *)
d = BooleanInterval[Table[RandomInteger[], {n}]]; (* Perturbing Term *)
  (* Print[d]; *)
o = If[RandomInteger[] < 0.05, o, And[o, Not[d]]]; (* Semantic Mutation *)
  (* Print[o]; *)
fo = Total[BooleanTable[o]]; (* Fitness of the Offspring *)
  (* Print[fo]; *)
If[fo > fworst, posworst = Position[fpop, fworst][[1, 1]]; (* Position Worst Individual *)
  (* Replace Parent if Offspring is better, and Simplify *)
  fpop[[posworst]] = fo;
  fbest = Max[fpop]; (* Fitness Best Individual *)
  posbest = Position[fpop, fbest][[1, 1]]; (* Position Best Individual *)
  fworst = Min[fpop]; (* Fitness Worst Individual *)
  posworst = Position[fpop, fworst][[1, 1]]; (* Position Worst Individual *)
  Print[pop[[posbest]]]; (* Print the Optimum Solution *)
]
A compact framework for evolutionary computation in Scala

Composed of two libraries: Scevo and Scaps

Component assembly via mixins

Interoperable with

Links:

- ScEvo
- Scaps
Additional resources
Recommended reading

- See: http://www.cs.bham.ac.uk/~wbl/biblio/
Recommended reading

- A Field Guide to Genetic Programming
  http://www.gp-field-guide.org.uk/ [Poli et al., 2008]

(This presentation uses some figures from the Field Guide)
The online GP bibliography [www.cs.bham.ac.uk/~wbl/biblio/](http://www.cs.bham.ac.uk/~wbl/biblio/)

The genetic programming ‘home page’
Classes/exercises
Prerequisites

- Java VM (JRE), ECJ, command line

Instructions:
- Download ecj.zip from cs.gmu.edu/~eclab/projects/ecj/
- Unzip it
- Open terminal
- Applications are available in the directory/package: ecj/ec/app/
- Warning: Some functionalities (e.g., GUI with charting) may require additional libraries. See documentation.
The task:

Could you paint a replica of the Mona Lisa using only 50 semi transparent polygons? (source link)

Note: Contrary to page content, this is not GP, just EA: solutions are vectors of coordinates and colors of polygons (inspect the *param file)

Configuration file:

ec/app/mona/mona.params

Launching:

java -cp ../..../jar/ecj.22.jar ec.Evolve -file mona.params
Exercise 2: Synthesis of Boolean functions

- Synthesis of Boolean functions
- Running on the multiplexer problem:

  \[
  \text{java -cp ..//..//..//jar/ecj.22.jar ec.Evolve -file 6.params}
  \]

- Have a look at out.stat
- See the impact of initial population: seed.0 = <integer>
- Other problems: parity
Exercise 3: Symbolic regression

- Symbolic regression

  java -cp ../../../jar/ecj.22.jar ec.Evolve -file noerc.params

- See the effect of:
  - increasing population size,
  - increasing the number of generations,
  - using multiple threads for evaluation (parameter ‘evalthreads’)

Classes/exercises 160
Exercise 4: Evolving agent’s controller

- Artificial ant: An agent (ant) operates in a discrete environment, collecting food pellets.
- See exemplary board

```
java -cp ../../jar/ecj.22.jar ec.Evolve -file progn4.params
```

- Note:
  - delayed rewards,
  - agent can be assessed only via taking part in entire episodes,
  - relations to reinforcement learning.
Demos
Ant Wars

- A two-person, zero-sum, partially observable, turn-based game used as a benchmark in GP.
- Our GP-evolved player, BriliAnt, won the AntWars contest [Jaskowski et al., 2008].
- BriliAnt exhibits a surprisingly rich repertoire of evolved behaviors: efficient diagonal board exploration, counting. Can even commit suicide when that pays off!
- Play with brilliant online at http://www.cs.put.poznan.pl/kkrawiec/antwars/
Interactive evolution of GP-generated patterns

Involves CPPN, Compositional Pattern Producing Network, a kind of GP program that capable of generating complex patterns in arbitrarily dimensional spaces.

CPPN used also in NeuroEvolution of Augmented Topologies (NEAT), an algorithm evolution of neural networks with indirect encoding.

See http://picbreeder.org/ and http://endlessforms.com/

PicBreeder
Recent developments in program synthesis
Recent developments in program synthesis

- Growing importance of domain-specific languages
  - Moving to higher-level concepts shrinks the search space and improves scalability

- Programming by example
  - Flash fill in MS Excel [Harris & Gulwani, 2011] (users SAT solvers to solve synthesis tasks)
  - https://www.youtube.com/watch?v=qHkgJFJR5cM
  - https://www.youtube.com/watch?v=_mkh5LrkcRI

- End user programming
  - New ways of specifying user’s intent
  - Interactive programming

- Programming using natural language
- Test-driven development
- Feedback generation
Importance of user intent

If a user is not capable of producing formal specification, how should we elicit if from him?

- Or: “How to program when you cannot” – The motto of software engineering according to E. Dijkstra :) [Dijkstra, 1988]

Alternative ways of specifying user intent (apart from input-output examples) [Gulwani, 2010]:

- demonstrations,
- natural language,
- partial or inefficient programs [Gulwani, 2010]
Recursive sorting algorithms of $n \log n$ complexity using object-oriented GP

Solutions to: list reversal, cartesian product, intersecting two lists, string comparison, sorting a list, locating a substring, binary multiplication, simplifying a polynomial, transposing a matrix, permutation generation, path finding, binary addition, and more [Olsson, 1998]

Loops: John Koza’s patent: [Koza et al., 2003a]

Synthesizing loop invariants [Cardamone et al., 2011]

Recursive programs (factorial, fibonacci, etc.)
Topics not covered in this course

- Schemata theorem for GP
  - Exact formula for the expected number of individuals sampling a schema a the next generation [Poli, 2001]
  - Plus later work for other types of crossover.
- Theory on bloat
- Theory on semantic GP


Experiments with explicit for-loops in genetic programming.

When novelty is not enough.
In *Proc. EvoApplications*.

*The Blind Watchmaker: Why the Evidence of Evolution Reveals a Universe Without Design*.
Norton.

Adaptive Trading With Grammatical Evolution.
In *Proc. CEC*.

On the cruelty of really teaching computing science.
circulated privately.

On the reliability of programs.
circulated privately.


Genetic Programming for Prevention of Cyberterrorism through Dynamic and Evolving Intrusion Detection.

Developments in Cartesian Genetic Programming: self-modifying CGP.
*GPEM, 11*, 397–439.

Spreadsheet table transformations from examples.

Dynamic Scheduling with Genetic Programming.
In *Proc. EuroGP*.

Jaskowski, W., Krawiec, K., & Wieloch, B. (2008).
Winning ant wars: Evolving a human-competitive game strategy using fitnessless selection.
A Survey of NP-Complete Puzzles.

Evolving a Sort: Lessons in Genetic Programming.
In *Proc. of the International Conference on Neural Networks*.

Generality and difficulty in genetic programming: Evolving a sort.
In S. Forrest (Ed.), *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93* (pp. 287–294). University of Illinois at Urbana-Champaign: Morgan Kaufmann.

A Genetic Approach to the Truck Backer Upper Problem and the Inter-twined Spiral Problem.
In *Proc. International Joint Conference on Neural Networks*.

Hierarchical genetic algorithms operating on populations of computer programs.
In N. S. Sridharan (Ed.), *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1 (pp. 768–774). Detroit, MI, USA: Morgan Kaufmann.


*Evolutionary Feature Programming: Cooperative learning for knowledge discovery and computer vision.*
Number 385 in . Poznan University of Technology, Poznan, Poland: Wydawnictwo Politechniki Poznanskiej.

Generative learning of visual concepts using multiobjective genetic programming.
*Pattern Recognition Letters, 28*(16), 2385–2400.

Visual learning by coevolutionary feature synthesis.

Visual learning by evolutionary and coevolutionary feature synthesis.
**Krawiec, K. & O’Reilly, U.-M. (2014).**

Behavioral programming: a broader and more detailed take on semantic GP.


Best paper.

---

**Krawiec, K. & Swan, J. (2013).**

Pattern-guided genetic programming.

Repeated Patterns in Genetic Programming.

Random search is parsimonious.
In E. Cantú-Paz (Ed.), *Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO-2002)* (pp. 308–315). New York, NY: AAAI.

Repeated Sequences in Linear Genetic Programming Genomes.

Evolving Regular Expressions for GeneChip Probe Performance Prediction.
In *Proc. PPSN* (pp. 1061–1070).

*Foundations of Genetic Programming*.
Springer-Verlag.

Creating Regular Expressions as mRNA Motifs with GP to Predict Human Exon Splitting.
In *Proc. GECCO*. 
Automated DNA Motif Discovery.
arXiv.org.

A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each.
In M. Glinz (Ed.), 34th International Conference on Software Engineering (ICSE 2012) (pp. 3–13). Zurich.

Controlling Complex Dynamics with Artificial Biochemical Networks.
In Proc. EuroGP (pp. 159–170).

Othello Competition.
http://\protect\kern-.1667em\relax/algova1.essex.ac.uk:8080/othello/html/Othello.html.
[Online; accessed 27-Jan-2012].

The Physical Travelling Salesperson Problem.
http://\protect\kern-.1667em\relax/algova1.essex.ac.uk/ptsp/ptsp.html.
[Online: accessed 27–Jan-2012].

A deductive approach to program synthesis.

FFX: Fast, Scalable, Deterministic Symbolic Regression Technology.
In *Proc. GPTP*.

An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach.

*Machine Learning*.
McGraw-Hill.
BBN Technical Report #7866, Bolt Beranek and Newman, Inc., 10 Moulton Street, Cambridge, MA 02138, USA.

In C. A. Coello Coello, V. Cutello, K. Deb, S. Forrest, G. Nicosia, & M. Pavone (Eds.), *Parallel Problem Solving from Nature, PPSN XII (part 1),* volume 7491 of *Lecture Notes in Computer Science* (pp. 21–31). Taormina, Italy: Springer.

Nguyen, S., Zhang, M., Johnston, M., & Tan, K. C.
Automatic programming via iterated local search for dynamic job shop scheduling.
*IEEE Transactions on Cybernetics.*
Forthcoming.

In *Proc. EuroGP.*

Genetic programming controlling a miniature robot.
In E. V. Siegel & J. R. Koza (Eds.), *Working Notes for the AAAI Symposium on Genetic Programming* (pp. 61–67). MIT, Cambridge, MA, USA: AAAI.
Evolutionary-computer-assisted design of image operators that detect interest points using genetic programming.

Population management for automatic design of algorithms through evolution.

Subtree Deactivation Control with Grammatical Genetic Programming in Dynamic Environments.
In *Proc. CEC*.

Maximum Overhang.

Exact schema theory for genetic programming and variable-length genetic algorithms with one-point crossover.
*Genetic Programming and Evolvable Machines, 2*(2), 123–163.
On the search properties of different crossover operators in genetic programming.

A field guide to genetic programming.
(With contributions by J. R. Koza).

Procedural texture evolution using multiobjective optimization.

Grammatical evolution: Evolving programs for an arbitrary language.


State-of-the-Art Genetic Programming for Predicting Human Oral Bioavailability of Drugs.
In Proc. 4th International Workshop on Practical Applications of Computational Biology and Bioinformatics.

Let the Games Evolve!
In Proc. GPTP.

In R. Riolo, T. McConaghy, & E. Vladislavleva (Eds.), Genetic Programming Theory and Practice VIII, volume 8 of Genetic and Evolutionary Computation chapter 2, (pp. 17–33). Ann Arbor, USA: Springer.

Spector, L., Clark, D. M., Lindsay, I., Barr, B., & Klein, J. (2008).
Genetic programming for finite algebras.
The push3 execution stack and the evolution of control.
Cantu-Paz, D. Dasgupta, K. Deb, J. A. Foster, E. D. de Jong, H. Lipson, X. Llora, S.
(Eds.), GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary
computation, volume 2 (pp. 1689–1696). Washington DC, USA: ACM Press.

Push 2.0 Programming Language Description.

Program Inversion Revisited.
Technical Report MSR-TR-2010-34, Microsoft Research.

Super Mario Evolution.
In Proc. IEEE Computational Intelligence and Games.

torcs (2012).
TORCS: The Open Car Racing Simulator.
http://torcs.sourceforge.net/.


*Hacker’s Delight.*
Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Elementary landscape analysis.

GP challenge: Evolving energy function for protein structure prediction.

No free lunch theorems for optimization.

Hierarchical Processing for Evolving Recursive and Modular Programs Using Higher-Order Functions and Lambda Abstraction.
*GPEM*, 2, 345–380.