Program Synthesis

Krzysztof Krawiec

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

March 11, 2016

Introduction

Outline and objectives

Objective: Provide state-of-the-art perspective on program synthesis, with emphasis on genetic programming.

Outline:

1. Program synthesis: problem definition, paradigms, challenges
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

Detailed table of contents

Course organization

- Too large field to be covered in a short course
- A number of relatively short, focused sections
- Questions and interactions welcome
- Clickable hyperlinks in blue or red

Course organization

- Too large field to be covered in a short course
- A 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
```

Credits

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
- Bartosz Wieloch, Poznan University of Technology
- Tomasz Pawlak, Poznan University of Technology
- Paweł Liskowski, Poznan University of Technology
- Iwo Błądek, Poznan University of Technology

What is program synthesis about?

Program synthesis (PS) task (Programming task)

Given:

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

find a program $p \in P$ such that:

$$Correct(p)$$

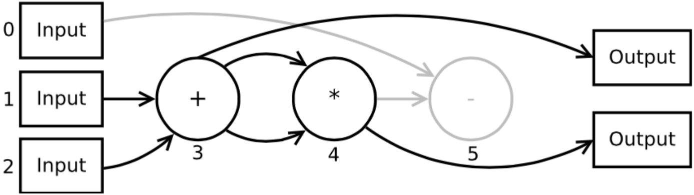
Note:

- Follows [Manna & Waldinger, 1980], yet earlier attempts present in AI
- In this purest form, program synthesis is a *search problem*
- Not to be confused with (an older term of) *automatic programming* (e.g., translating higher-level source code into machine code)
- Essential detail: how to define *Correct*

What is a program?

Several mutually nonexclusive interpretations:

- Source code
- Abstract syntax tree
- Discrete, finite, executable *structure*



[Turner & Miller, Neutral Genetic Drift: An Investigation using Cartesian Genetic Programming, 2016]

What does it mean that a program is correct?

- 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.

What does it mean that a program is correct?

- Programs are not any formal objects: they are *functions* $I \to O$
- We consider a program correct if it *behaves* as expected, i.e., produces the desired output given input.

Possible definitions of *Correct*:

- A program that passes all *tests* (a finite number thereof)
- A program that is *provably* correct.
  - i.e., conforms certain formal *specification*.
- A program mandated as correct by an *oracle* $Correct$.
- User's *intent*.

Practictioner's perspective

S. Gulvani (Microsoft Research), *Dimensions in Program Synthesis* [Gulwani, 2010a]:

1. User intent: logical specifications, natural language, input-output examples (tests), traces, programs
2. Search space: programs, grammars, logics
3. Search technique: brute-force search, version space algebra, machine learning (probabilistic inference, genetic programming), logical reasoning based techniques

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]

Non-orthodox ways of specifying user intent [Gulwani, 2010b]:

- demonstrations,
- natural language,
- partial or inefficient programs [Gulwani, 2010b]

Alternative phrasings of the PS task:

- Program synthesis is the task of discovering an executable program from user intent expressed in the form of some constraints [Gulwani, 2010b].
- Program synthesis is the automatic translation of a specification into a program.

Ways to solve a programming task

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

Ways to solve a programming task

- State of the art: human programmer(s)
  - Slow, imperfect, unreliable, unsafe, ...
  - ... yet getting better and more powerful (?)
  - 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]

Ways to solve a programming task

- State of the art: human programmer(s)
  - Slow, imperfect, unreliable, unsafe, ...
  - ... yet getting better and more powerful (?)
  - 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)

Edsgar Wybe Dijkstra

Edsger Wybe Dijkstra, 1930–2002

On the reliability of programs.

All speakers at the lecture series have received very strict instructions as how to arrange their speach; as a result I expect all speaches to be similar to each other. Mine will not differ, I adhere to the instructions. They told us: first tell what you are going to say, then say it and finally XXXXXXXX summarize what you have said.

My story consists of four points.

1) I shall argue that our programs should be correct

2) I shall argue that debigging is an inadequate means for achieving that goal and that we must prove the correctness of programs

3) I shall argue that we must tailor our programs to the proof requirements

4) I shall argue that programming will become more and more an activity of mathematical nature.

www.cs.utexas.edu/∼EWD/

On importance of correctness

Ariane-5 crash on June 4, 1996.
The culprit: conversion of 64-bit float into a 16-bit int.

Other examples:

- Bug in Intel Pentium processors $\Longrightarrow$ \$475 mln to replace
- Bug in baggage handling system at Denver airport $\Longrightarrow$ nine month delay, \$1.2 per day
- Bug in radiation therapy device $\Longrightarrow$ death of six patients

Related: Formal verification and model checking

Model checking = *an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model* [Baier & Katoen, 2008, p. 11]

Phases:

- Modeling: building a model of a system of consideration, in some language
  - Typically some form of finite-state automaton
- Running: application of *model checker*
  - When checking fails, it produces a *counterexample*
- Analysis: analyze counterexample, refine the model, etc.

Specifying program correctness

- Example of program specification [Manna & Waldinger, 1980]:

$$sqrt(n) \Leftarrow \text{find } z \text{ such that } integer(z) \text{ and } z^2 \leq n \leq (z+1)^2$$
$$\text{where } integer(n) \text{ and } 0 \leq n$$

Specifying program correctness

- Example of program specification [Manna & Waldinger, 1980]:

$$
\begin{array}{c}
\textit{sqrt}(n) \Leftarrow \text{find } z \text{ such that } \textit{integer}(z) \text{ and } z^2 \leq n \leq (z+1)^2 \\
\text{where } \textit{integer}(n) \text{ and } 0 \leq n
\end{array}
$$

More generally:

$$
\begin{array}{c}
f(a) \Leftarrow \text{find } z \text{ such that } R(a,z) \\
\text{where } P(a)
\end{array}
$$

where:

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

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 (isomorphism)

- Haskell Curry (1900-1982), William Alvin Howard (1926-)
- One-to-one correspondence between programs and logic, i.e., programs and proofs, and types and propositions
- In a nutshell:
  - Proofs in logic **are** programs in computer science.
  - Propositions in logic **are** types in computer science.
- A program is a proof of the formula being the type of the program
- The rules of logic are search operators in the space of proofs.
- Prolog 'embodies' the CH correspondence.

Specifying correctness using examples (tests)

```
List(1,2,3,4,5,6,7,8,9); List(2,3,4,5,6,7,8,9)
List(19,-34,0); List(-34,0)
List(100,-200,300,900); List(-200, 300,900)
List(2,2,3); List(2,3)
List(5,4,3); List(4,3)
List(7,-4,-3); List(-4,-3)
List(0,1); List(1)
List(1,0); List(0)
List(12); List()
List(5); List()
List(-17); List()
```

Recap: What is special about program synthesis?

- We are talking about *programs* that generate *programs*.
  - Note: generate, not manipulate (like, e.g., compilers)
  - 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.

Main directions in program synthesis

As outlined in [Manna & Waldinger, 1980]:

- Exact[1] approaches:
  - Deductive program synthesis
  - Inductive programming
  - Transformation of specification (rewriting systems)
- Heuristic approaches (including genetic programming)

[1] Meaning: Either you get a correct program, or you don't get anything.

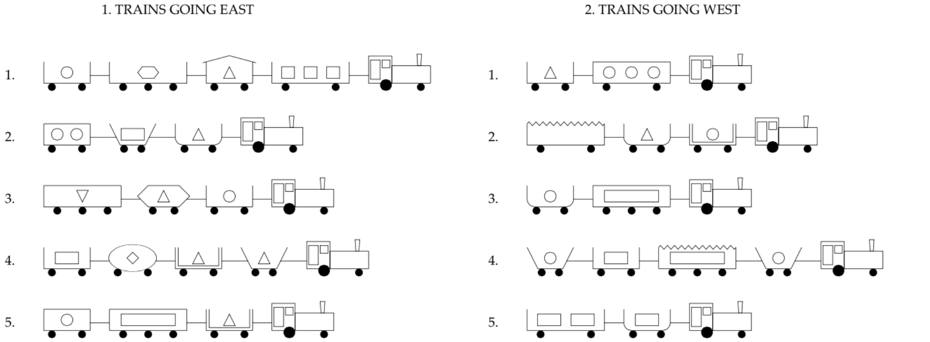
Deductive program synthesis

- 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

Inductive logic programming: An example



Source: [Flach & Lavrac, 2000]

Inductive logic programming: An example

```
east(t1).

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

hasCar(t1,c13).
...
```

```
hasCar(t1,c12).
cshape(c12,rect).
clength(c12,long).
cwall(c12,single).
croof(c12,no).
cwheels(c12,3).
hasLoad(c12,l12).
lshape(l12,hexa).
lnumber(l12,1).

hasCar(t1,c14).
...
```

Exemplary hypothesis:

```
east(T):-hasCar(T,C),clength(C,short),croof(C,no)
```

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[2]
    - 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]

[2] 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[3],
- 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.

[3] 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:

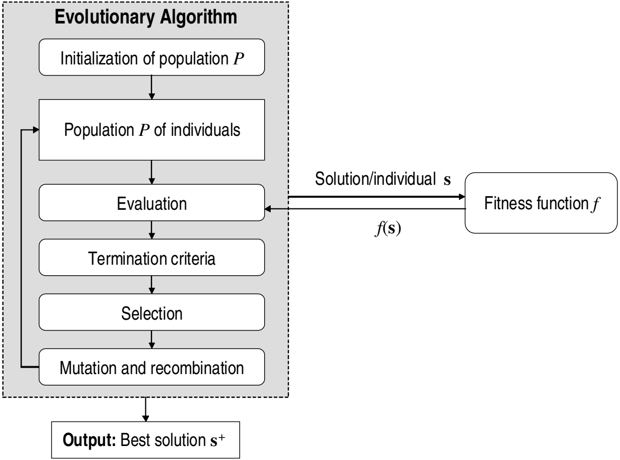
- Operate on populations of candidate solutions
- Candidate solutions are encoded as *genotypes*
- Genotypes get decoded into *phenotypes* when evaluated by the *fitness function* f being optimized.
  - Example: a candidate solution to a traveling salesperson problem is a permutation of cities (genotype), 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!

Generic evolutionary algorithm



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

Features of EC

- '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 Neural Nets being the other ones)

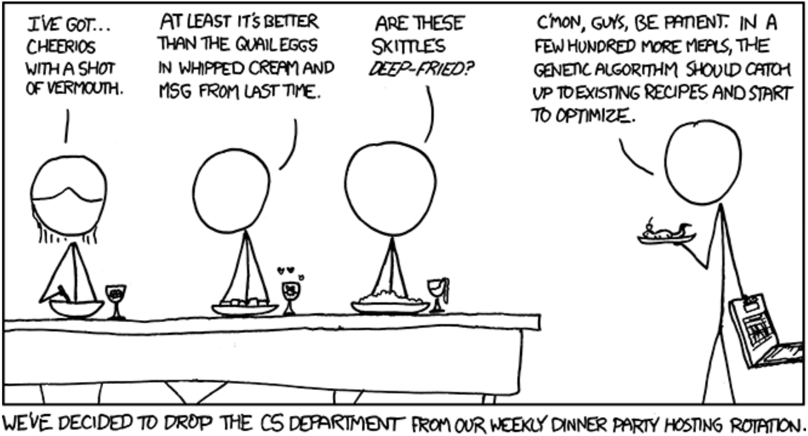
EAs are metaheuristics

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.
  - 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).

- optimization of car chassis (BMW),
- design of analog and digital circuits,
- design of antennae (NASA),
- feature selection in machine learning tasks,
- optimization of wind turbine placement (General Electric),
- 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.

Fitness function

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.

Fitness function: Example

City-block fitness function:

$$f(p) = -\sum_i ||y_i - p(x_i)||, \tag{1}$$

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.

Genetic programming

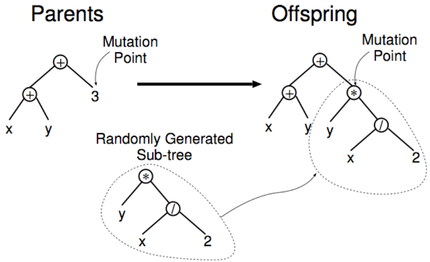
Main evolution loop ('vanilla GP')

procedure GeneticProgramming($f, \mathscr{I}$) ▷ f - fitness function, $\mathscr{I}$ - instruction set
  $\mathscr{P} \leftarrow \{p \leftarrow \text{RandomProgram}(\mathscr{I})\}$ ▷ Initialize population
  repeat ▷ Main loop over generations
    for $p \in \mathscr{P}$ **do** ▷ Evaluation
      $p.f \leftarrow f(p)$ ▷ $p.f$ is a 'field' in program p that stores its fitness
    end for
    $\mathscr{P}' \leftarrow \emptyset$ ▷ Next population
    repeat ▷ Breeding loop
      $p_1 \leftarrow \text{TournamentSelection}(\mathscr{P})$ ▷ First parent
      $p_2 \leftarrow \text{TournamentSelection}(\mathscr{P})$ ▷ Second parent
      $(o_1, o_2) \leftarrow \text{Crossover}(p_1, p_2)$
      $o_1 \leftarrow \text{Mutation}(o_1, \mathscr{I})$
      $o_2 \leftarrow \text{Mutation}(o_2, \mathscr{I})$
      $\mathscr{P}' \leftarrow \mathscr{P}' \cup \{o_1, o_2\}$
    until $|\mathscr{P}'| = |\mathscr{P}|$
    $\mathscr{P} \leftarrow \mathscr{P}'$
  until StoppingCondition($\mathscr{P}$)
  return $\arg\max_{p \in \mathscr{P}} p.f$
end procedure

Search operators: Mutation

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

```
function Mutation(p, 𝒥)
    repeat
        s ← Random node in p
        s′ ← RandomProgram(𝒥)
        p′ ← Replace the subtree rooted in s with s′
    until Depth(p′) < d_max                          ▷ d_max is the tree depth limit
    return p′
end function
```

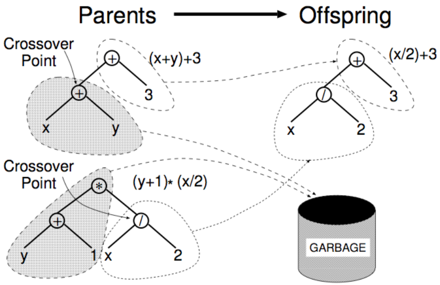


Source: [Poli et al., 2008]

Search operators: Crossover

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

function Crossover(p_1, p_2)
  repeat
    $s_1 \leftarrow$ Random node in p_1
    $s_2 \leftarrow$ Random node in p_2
    $(p'_1, p'_2) \leftarrow$ Swap subtrees rooted in s_1 and s_2
  until Depth$(p'_1) < d_{max} \wedge$ Depth$(p'_2) < d_{max}$ ▷ d_{max} is the tree depth limit
  return (p'_1, p'_2)
end function



Source: [Poli et al., 2008]

Q & A

Q: What is the most likely outcome of application of mutation/crossover to a viable program?

[4] Turns out: In GP, quite many of them can be neutral (*neutral mutations*).

Q & A

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[4]

Yet, GP works. Why?

[4] Turns out: In GP, quite many of them can be neutral (*neutral mutations*).

Q & A

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[4]

Yet, GP works. Why?

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

[4] 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])

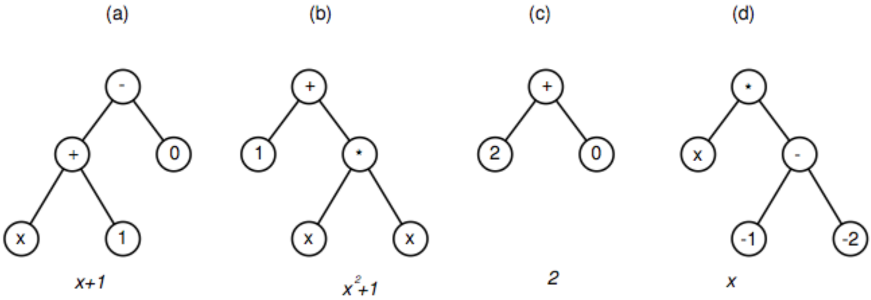
- Objective: Find a program whose output matches $x^2 + x + 1$ over the range $[-1, 1]$.
  - Such tasks can be considered as a form of regression.
  - As solutions are built by manipulating code (symbolic instructions), this is referred to as *symbolic regression*.
- Fitness: sum of absolute errors (City-block distance) for $x \in -1.0, -0.9, ...0.9, 1.0$:

x_i	-1.0	-0.9	...	0	...	0.9	1.0
y_i	1	0.91	...	1	...	2.71	3

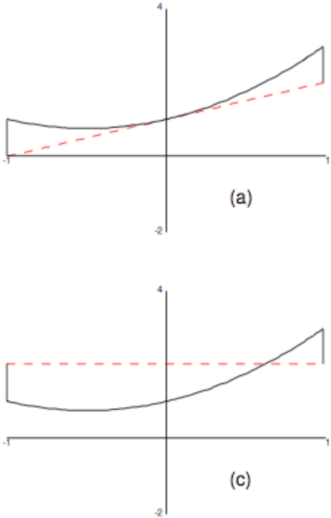
Exemplary run: Setup

- Instruction set:
  - Nonterminal (function) set: +, -, % (protected division), and x; all operating on floats
  - Terminal set: x, 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

Initial population (population 0)



Fitness assignment for population 0



Fitness values: f(a)=7.7, f(b)=11.0, f(c)=17.98, f(d)=28.7

Breeding

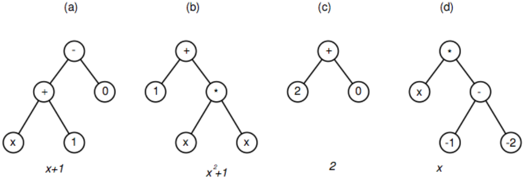
Assume:

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

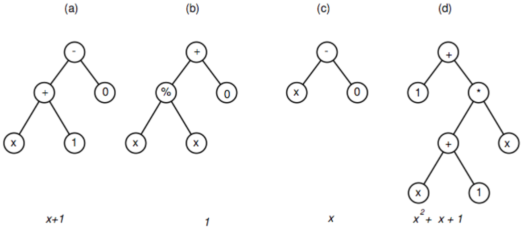
Note:

- 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).

Q: Is GP a ML technique?

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).

Why should GP be considered a viable approach of AI/CI?

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[5] 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.

5

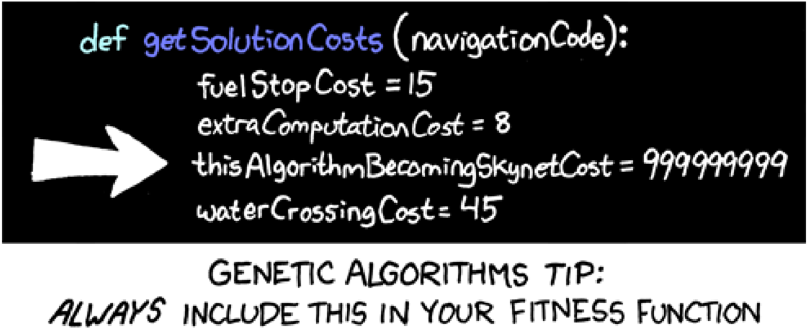
What is GP? – Question revisited

> *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.

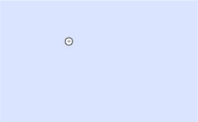
Risks involved?

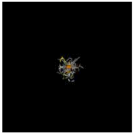
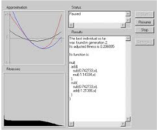
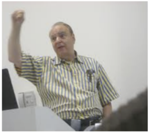


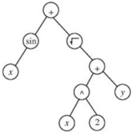
Source: http://xkcd.com/534/

Early work by:

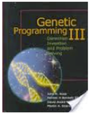
- John R. Koza [Koza, 1989, Koza, 1992b]
- Similar ideas in early works of Schmidhuber [Schmidhuber, 1987]

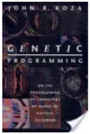









http://www.genetic-programming.com/johnkoza.html

Exemplary GP run using ECJ

Exemplary run of ECJ (EC in Java [Luke, 2010])

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: $+$, $-$, $*$, $/$, 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},
...
{0.6602806044824949,0.1386949839559808 4},
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)) (+ (+
         (sin x) x) (- x x))) (exp (* x (sin x))))
         (sin x)) (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))))
```

FUEL: FUnctional Evolutionary aLgorithms

Compact framework for implementing metaheuristic algorithms written in Scala

- ~2000 LoC
- Convenient on-the-fly manipulation of components
- Single- and multiobjective evolutionary search
- ...
- https://github.com/kkrawiec/fuel

Launching an EA run:

```
object MaxOnes2 extends IApp('numVars -> 500, 'maxGenerations -> 200,
  'printResults -> true) {
  RunExperiment(SimpleEA(
    moves = BitSetMoves(opt('numVars, (_: Int) > 0)),
    eval = (s: BitSet) => s.size,
    optimalValue = 0))
}
```

A more detailed view on GP

There is much beyond the 'vanilla 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:

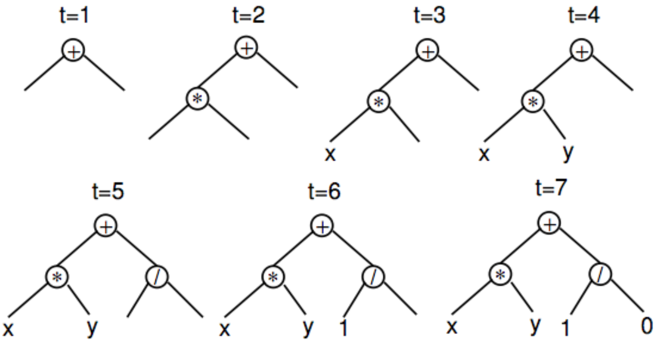
- generative vs. steady-state evolution,
- selection operators (fitness-proportional, tournament, ...)
- extensions: island models, estimation-of-distribution algorithms, multiobjective EAs, ...

Where to get the candidate solutions from?

- 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.

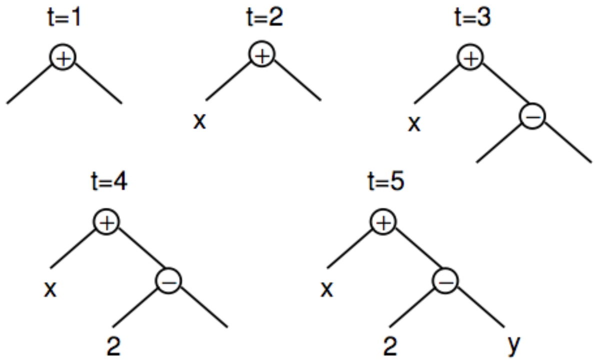
Initialization: *Full* method

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



Initialization: *Grow* method

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



Initialization: Comments

- h_{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

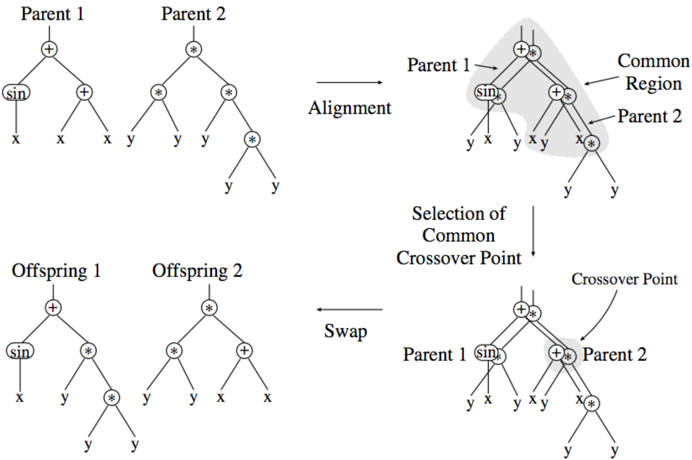
Even though the conventional GP crossover operators care only about program syntax, there are quite many of them. Examples:

- 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.

- 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:

```
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

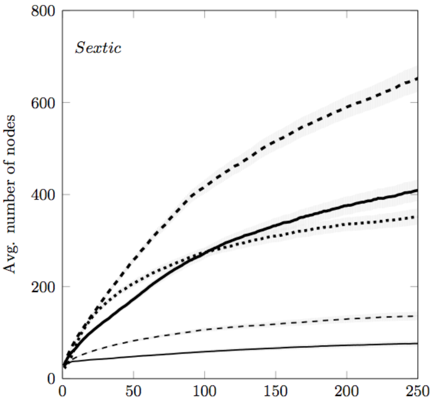
```
pop.subpop.0.species.pipe.num-sources = 2
pop.subpop.0.species.pipe.source.0 = ec.gp.koza.CrossoverPipeline
pop.subpop.0.species.pipe.source.0.prob = 0.9
pop.subpop.0.species.pipe.source.1 = ec.gp.koza.MutationPipeline
pop.subpop.0.species.pipe.source.1.prob = 0.1
```

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)

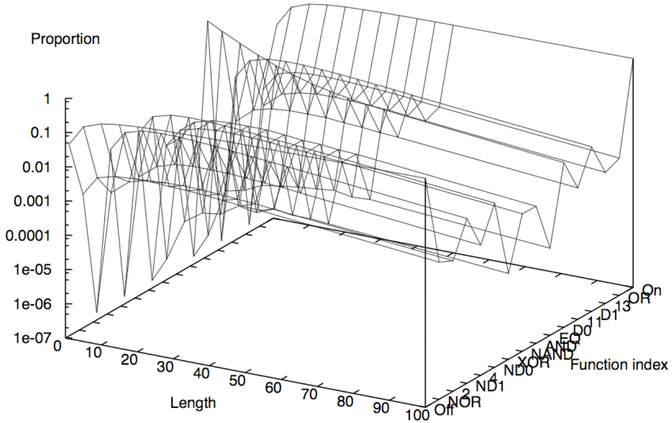


Countermeasures for bloat

- 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]

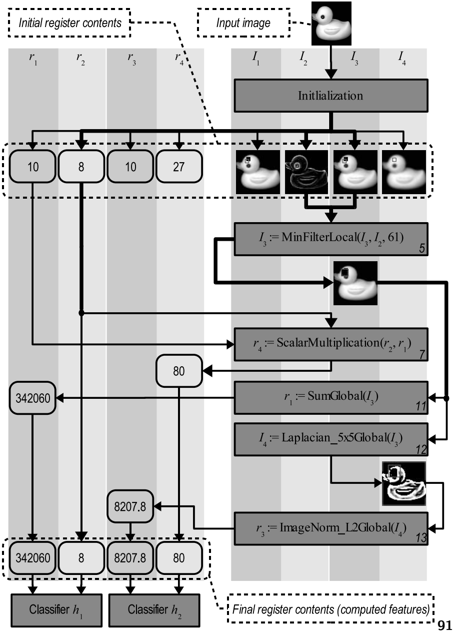
High cost of evaluation

- 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]
- 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[6] types

[6] 'Compatible' = belonging to the same 'set type'

Strongly typed GP: Example

Consider 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

Implementaion of this type system in ECJ:

Types:
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

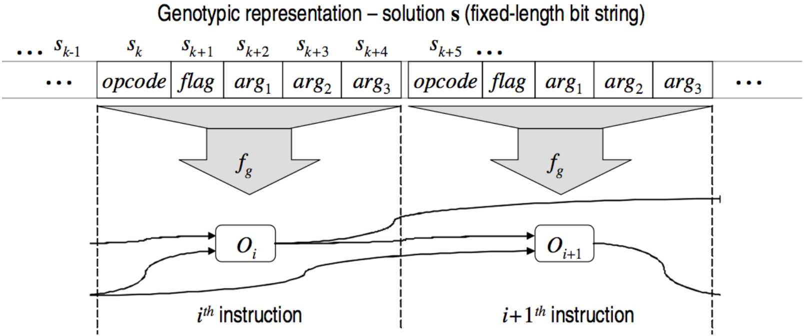
Type constraints for programs:
gp.tc.size = 1
gp.tc.0 = ec.gp.GPTreeConstraints
gp.tc.0.name = tc0
gp.tc.0.fset = f0
gp.tc.0.returns = class

Type constraints for instructions: ('templates' of type constraints)
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

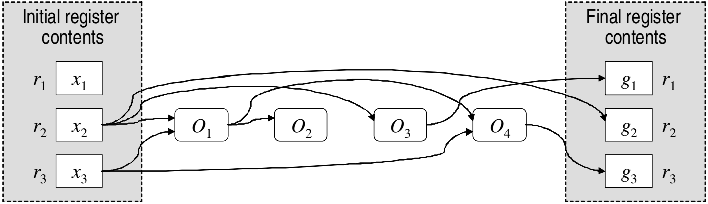
Linear Genetic Programming

- Motivation: Tree-like structures are not natural for contemporary hardware architectures
- Program representation: a <u>sequence</u> of instructions
- Passing data between instructions: via <u>registers</u>
- Often 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:



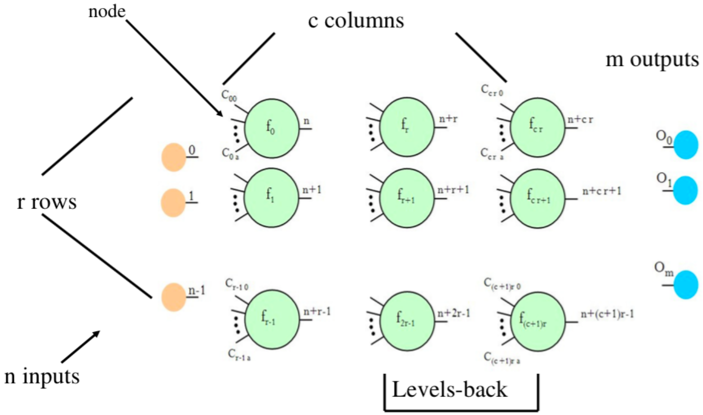
and the corresponding data flow, including the initial and final register contents:



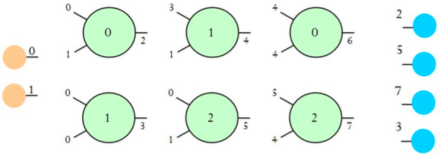
Cartesian GP

Developed from work on the evolution of digital circuits [Miller & Thomson, 1998, Miller & Thomson, 2000].

- Program representation: a graph of instructions
  - However, encoded as a sequence of integers.
- Passing data between instructions: direct
- Applications: evolution of digital and analog circuits.



Example

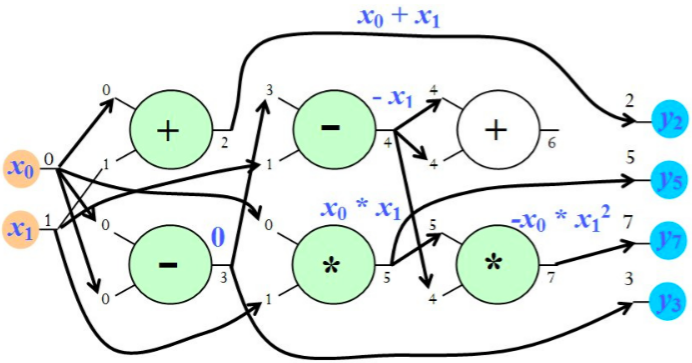


Function look-up table

Function gene (address)	Action
0	Add
1	Subtract
2	Multiply
3	Divide (protected)

Genotype

0 0 1 1 0 0 1 3 1 2 0 1 0 4 4 2 5 4 2 5 7 3



$$y_2 = x_0 + x_1$$
$$y_5 = x_0 * x_1$$
$$y_7 = -x_0 * x_1^2$$
$$y_3 = 0$$

Stack-based GP

- PushGP [Spector et al., 2004]
- Program representation: a nested list of instructions
- Syntax: program ::= instruction | literal | (program*)
- Passing data between instructions: via typed stacks
- Simple cycle of program execution:
  - Pop an instruction from the EXEC stack and execute it.
  - The instruction will usually pop some data from a data stack and push the results on the stack of the appropriate type.
  - Upon termination, the top element of a stack forms program outcome
- Includes certain features that make it Turing-complete (e.g., YANK instruction).
- Natural possibility of implementing autoconstructive programs [Spector, 2010]

Push: Example 1

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

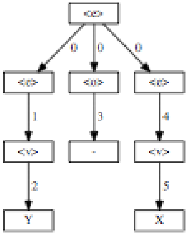
		Fitness case 1		Fitness case 2		Fitness case 3	
Step	EXEC	INT	BOOL	INT	BOOL	INT	BOOL
0	(* + <)	(1 3 4 5)	( )	(2 2 4 2)	( )	(1 2 3 8)	( )
1	(+ <)	(3 4 5)	( )	(4 4 2)	( )	(2 3 8)	( )
2	(<)	(7 5)	( )	(8 2)	( )	(5 8)	( )
3	( )	( )	(F)	( )	(F)	( )	(T)

More details: http://hampshire.edu/lspector/push3-description.html

- 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
```

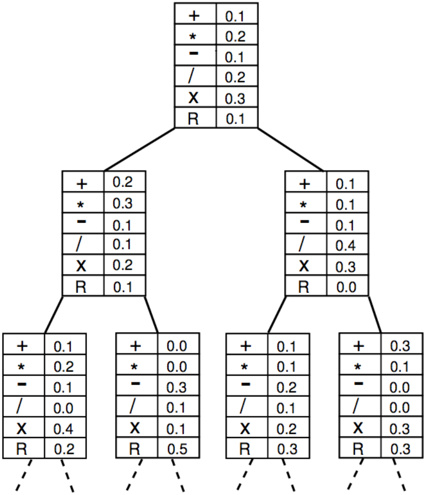


Other variants of GP

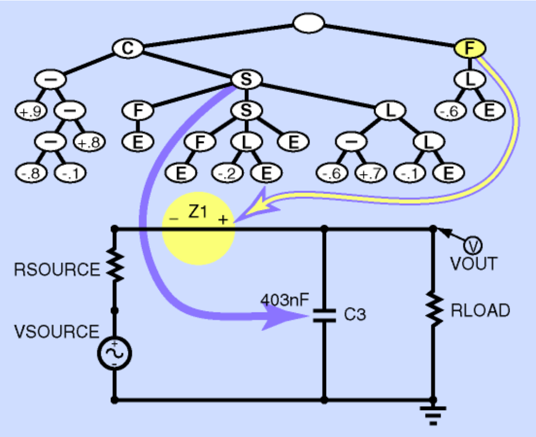
- Multiobjective GP. The extra objectives can:
  - Come with the problem
  - Result from GP's specifics: e.g., use program size as the second (minimized) objective
  - Be associated with different tests (e.g., feature tests [Ross & Zhu, 2004])
- Probabilistic GP (a variant of EDA, Estimation of Distribution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



- Programs generate solutions [Koza et al., 1999].
  - Or modify a 'baseline' solution.
- Intricate mapping between program and the final (evaluated) artifact.

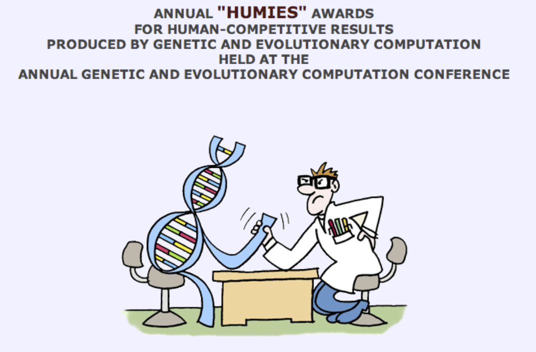


http://www.genetic-programming.com/gpcircuitanimation.gif

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].

(...) *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

Humies

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.

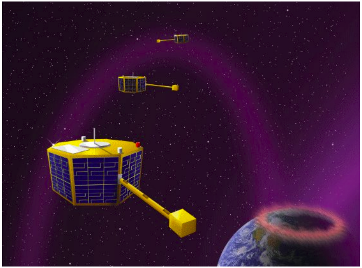
(F) The result is equal to or better than a result that was considered an achievement in its field at the time it was first discovered.

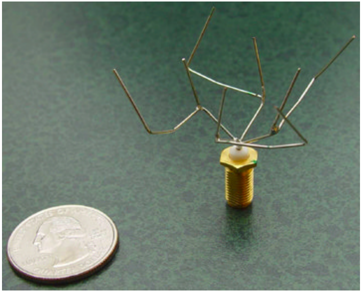
(G) The result solves a problem of indisputable difficulty in its field.

(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).

Selected Gold Humies using GP

- 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





http://idesign.ucsc.edu/papers/hornby_ec11.pdf

- 2009: S. Forrest, C. Le Goues, ThanhVu Nguyen, W. Weimer
  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.

Selected Gold Humies using GP

- 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=95NOYokm6Bk

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

Review

- 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].

Other applications

- Classification problems in machine learning and object recognition [Krawiec, 2001, Krawiec & Bhanu, 2005, Krawiec, 2007, Krawiec & Bhanu, 2007, Olague & Trujillo, 2011],
- 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 ∞)
- 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]

... and more ....

<u>Game-Playing</u>

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]

<u>Dynamic Optimisation</u>

Dynamic Symbolic Regression [O'Neill et al., 2008]

Dynamic Scheduling [Jakobović & Budin, 2006]

<u>Traditional Programming</u>

Sorting [Kinnear, Jr., 1993a]

Semantic GP

The fitness bottleneck problem

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.

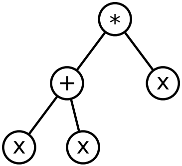
Why do we stick to this design? **There are no principal reasons to maintain the bottleneck.**

(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:



x_i	$p(x_i)$
-0.5	**0.5**
1.0	**2.0**
1.5	**4.5**
2.0	**8.0**

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

Can been used for:

- designing initialization operators,
- diversity maintenance,
- designing search operators.

Key observation for semantics GP

The fitness functions used in GP are usually *metrics*, like:

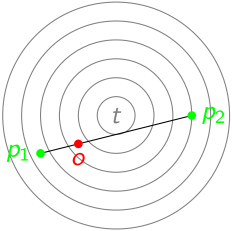
- Hamming distance: $|\{p(x_i) \neq y_i\}|$
- Manhattan distance: $\sum_i |p(x_i) - y_i|$
- Euclidean distance: $\sum_i |p(x_i) - y_i|^2$

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_is above, a.k.a. *target*, t in the next slides).

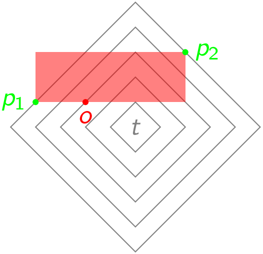
- Thus, the semantic space is a *metric space*, and fitness landscape forms a *unimodal cone*.

Geometric implications of program semantics

Semantic space (t - the target, i.e., vector of desired outputs):



(Euclidean metric)



(City-block 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.

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?

Exact geometric operators: The idea

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

$$\begin{array}{ccccc} T1 & \times & T2 & \overset{GX_{SD}}{\rightarrow} & T3 \\ \downarrow o & & \downarrow o & & \downarrow o \\ O1 & \times & O2 & \overset{GX_{D}}{\rightarrow} & O3 \end{array} \quad (3)$$

- 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 $T1, T2$ and $T3 = GX_{SD}(T1, T2)$ then $O(T3) = GX_D(O(T1), O(T2))$.

For boolean problems

Definition

Given two parent functions $T1, T2 : \{0,1\}^n \rightarrow \{0,1\}$, the recombination SGXB returns the offspring boolean function $T3 = (T1 \wedge TR) \vee (\overline{TR} \wedge T2)$ 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

```
        OR
      /    \
   AND       AND
  /   \     /   \
T1     TR  NOT   T2
            |
            TR
```

```
           AND
T1  =     /   \
        X1     X2

           OR
T2  =     /  \
        X2   X3

          NOT
TR  =      |
          X3
```

```
                        OR
                      /    \
                   AND      X3
T3  =             /   \
               AND     NOT
              /   \     |
            X1     X2   X3
```

- 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 $T1, T2 : \mathbb{R}^n \rightarrow \mathbb{R}$, the recombinations SGXE and SGXM return the real function $T3 = (T1 \cdot TR) + ((1 - TR) \cdot T2)$ 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

Problem	Hits %								Length			
	GP		GPt		SSHC		SGP					
	avg	sd	avg	sd	avg	sd	avg	sd	GP	GPt	SSHC	SGP
Comparator6	80.2	3.8	90.9	3.5	99.8	0.5	99.5	0.7	1.0	2.0	2.9	2.8
Comparator8	80.3	2.8	94.9	2.4	100.0	0.0	99.9	0.2	1.0	2.3	2.9	3.0
Comparator10	82.3	4.3	95.3	0.9	100.0	0.0	100.0	0.1	1.6	2.4	2.7	3.0
Multiplexer6	70.8	3.3	94.7	5.8	99.8	0.5	99.5	0.8	1.1	2.2	2.7	2.9
Multiplexer11	76.4	7.9	88.8	3.4	100.0	0.0	99.9	0.1	2.2	2.4	2.9	2.6
Parity5	52.9	2.4	56.3	4.9	99.7	0.9	98.1	2.1	1.4	1.7	2.9	2.9
Parity6	50.5	0.7	55.4	5.1	99.7	0.6	98.8	1.7	1.0	1.9	3.0	3.0
Parity7	50.1	0.2	51.7	2.8	99.9	0.2	99.5	0.6	1.0	1.7	3.0	3.1
Parity8	50.1	0.2	50.6	0.9	100.0	0.0	99.7	0.3	1.0	1.6	3.4	3.4
Parity9	50.0	0.0	50.2	0.1	100.0	0.0	99.5	0.3	1.0	1.3	3.8	3.8
Parity10	50.0	0.0	50.0	0.0	100.0	0.0	99.4	0.2	0.9	1.2	4.1	4.1
Random5	82.2	6.6	90.9	6.0	99.5	1.2	98.8	2.1	0.9	1.6	2.7	2.8
Random6	83.6	6.6	93.0	4.1	99.9	0.4	99.2	1.3	1.2	1.9	2.9	2.8
Random7	85.1	5.3	92.9	3.8	99.9	0.2	99.8	0.4	1.1	2.0	2.8	2.9
Random8	89.6	5.3	93.7	2.4	100.0	0.1	99.9	0.2	1.4	2.0	3.0	2.9
Random9	93.1	3.7	95.4	2.3	100.0	0.1	100.0	0.1	1.5	1.8	2.9	2.9
Random10	95.3	2.3	96.2	2.0	100.0	0.0	100.0	0.0	1.5	1.8	2.8	3.0
Random11	96.6	1.6	97.3	1.5	100.0	0.0	100.0	0.0	1.6	1.7	2.7	3.1
True5	100.0	0.0	100.0	0.0	99.9	0.6	100.0	0.0	1.1	1.3	2.0	2.4
True6	100.0	0.0	100.0	0.0	99.8	0.6	100.0	0.0	1.2	1.2	2.6	2.5
True7	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0	1.2	1.2	2.9	2.6
True8	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.1	1.2	1.4	3.3	2.9

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

Experimental results: real-valued programs

Problem	Hits %					
	GP		SSHC		SGP	
	avg	sd	avg	sd	avg	sd
Polynomial3	79.9	23.1	100.0	0.0	99.5	1.5
Polynomial4	60.5	27.6	99.9	0.9	99.9	0.9
Polynomial5	40.7	21.6	100.0	0.0	99.5	2.0
Polynomial6	37.5	23.4	100.0	0.0	98.9	3.1
Polynomial7	30.7	18.5	100.0	0.0	99.9	0.9
Polynomial8	34.7	16.0	99.5	2.0	99.7	1.3
Polynomial9	20.7	13.2	100.0	0.0	98.5	4.9
Polynomial10	25.7	16.7	99.4	1.7	99.9	0.9

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.

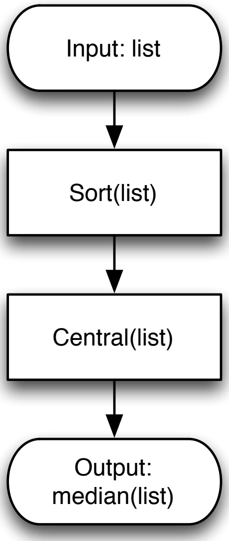
Behavioral GP and search drivers

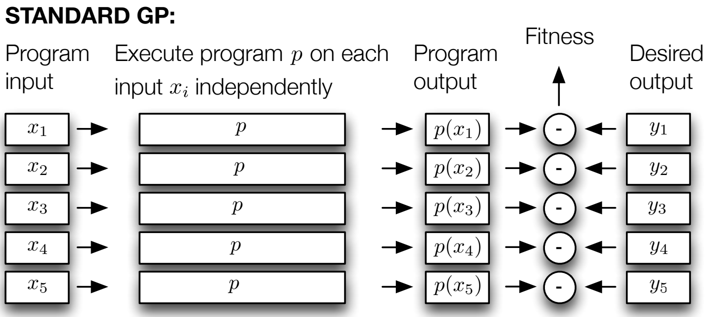
Behavioral GP

- Takes semantic GP even further
- The rationale: The final outcomes of program execution reveal only 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.

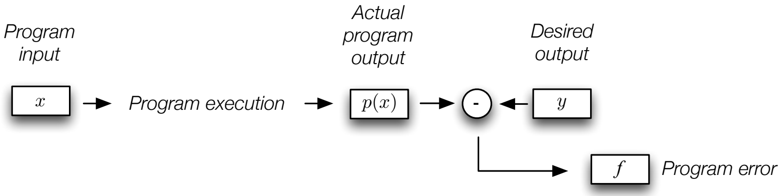
Example: Calculating the median

- 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?

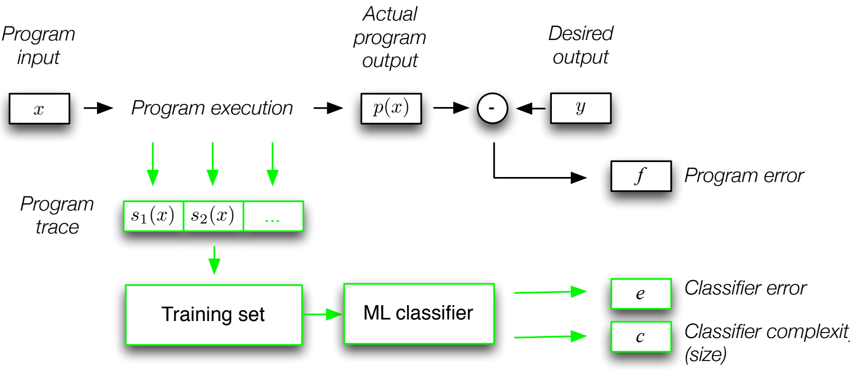



STANDARD GP:
Program input
Execute program p on each input x_i independently
Program output
Fitness
Desired output
x_1
x_2
x_3
x_4
x_5
p
$p(x_1)$
$p(x_2)$
$p(x_3)$
$p(x_4)$
$p(x_5)$
-
y_1
y_2
y_3
y_4
y_5

Standard GP

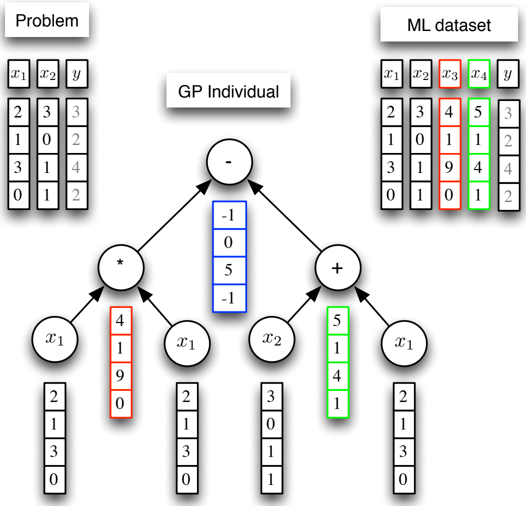

Program
input
Actual
program
output
Desired
output
x
Program execution
p(x)
-
y
f
Program error

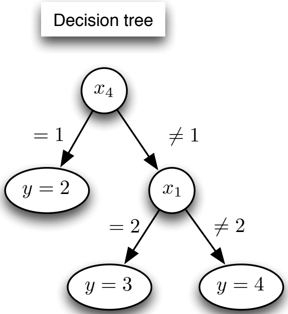
Pattern-guided GP



- **Black:** Conventional GP
- **Green:** PANGEA [Krawiec & Swan, 2013]

Example (nominal domain, tree-based GP)

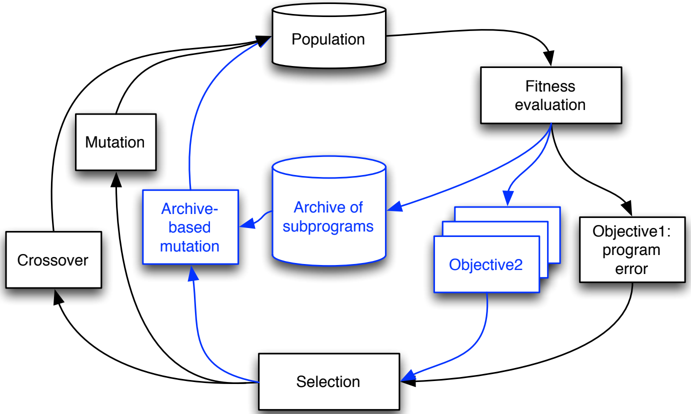




Evaluation:

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

Birds-eye view on program synthesis

"Dimensions in program synthesis"[Gulwani, 2010b], an 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.

Applications: Discovery of *new* algorithms

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]*

Oth

- mutual exclusion algorithms, i.e., algorithms that guarantee mutually exclusive access to critical sections

Applications: Synthesis of program inverses

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 possible 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.

Applications: End user programming

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

Alternative take on the Curry-Howard correspondence

- Motivation: types reveal the underlying semantics [Zoltan and Swan, 2014]
- 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 unknown types a and b, there is in general no function $a \rightarrow b$ (function type $a \rightarrow b$).
    - Only when some assumptions about a and b are made, such a function can be constructed (and thus the associated type $a \rightarrow b$ does exist).

Types reveal a lot about functions

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].*

Types reveal a lot about functions

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:

$$f : \mathsf{List}[T] \to \mathbb{N}$$

Types reveal a lot about functions

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:

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

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:

- $f = reverse$, $a = asciiCode$
- $f = tail$, $a = inc$

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:
Karolina Stanisławska, Krzysztof Krawiec, Zbigniew W. Kundzewicz: *Modeling Global Temperature Changes using Genetic Programming – A Case Study* (2012)

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:
Krzysztof Krawiec, Bartosz Kukawka and Tomasz Maciejewski, *Evolving cascades of voting feature detectors for vehicle detection in satellite imagery*. In IEEE Congress on Evolutionary Computation (CEC 2010). Barcelona, IEEE Press, pages 2392-2399.

Link to slides

Case study 3: Evolution of detectors of anatomical structures

Based on:
Krzysztof Krawiec, *Genetic Programming with Alternative Search Drivers for Detection of Retinal Blood Vessels*. In EvoApps'15, Copenhagen, Denmark, 2015 (to appear).

Link to slides

a_1	0	1	2
0	2	1	2
1	1	0	0
2	0	0	1

$a)$

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

$b)$

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

$c)$

- 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., 2015]

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/
- FUEL + ScaPS (PUT), Scala
  - https://github.com/kkrawiec/fuel

ECJ

- 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

- EpochX (University of Kent, UK), also in Java
  - http://www.epochx.org/
- Ready-to-run examples:
  - http://www.epochx.org/quickstart-guide.php
- Examples, including the Artificial Ant benchmark:
  - http://www.epochx.org/guide-models.php
- 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)

```
uniformDepthCrossoverexpr <- function (expr1, expr2) {
    newexpr1 <- expr1
    indExpr1 = randomIndexingExpressionSym(newexpr1,as.name(quote(newexpr1)))
    indExpr2 = randomIndexingExpressionSym(expr2,as.name(quote(expr2)))
    eval(call("=", indExpr1,indExpr2))
    newexpr1
}
```

http://cran.r-project.org/web/packages/gpr/index.html

Exemplary implementation of GP framework in 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[BooleanFunction[RandomInteger[2^(2^n) - 1], v], {k}];
(* Initial Population of Random Functions *)
fpop = Table[Total[Boole[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 *)
posworst = Position[fpop, fworst][[1, 1]]; (* Position Worst Individual *)
For[i = 0, fbest < (2^n), i++, (* Is current Solution the Optimum? *)
 Print[i, " ", fbest, " ", Length[pop[[posbest]]]];
 p1 = pop[[RandomInteger[{1, k}]]];
 (* select parents uniformly at random in the population *)
 (* Print[p1]; *)
 p2 = pop[[RandomInteger[{1, k}]]];
 (* Print[p2]; *)
 r = BooleanFunction[RandomInteger[2^(2^n) - 1], v];
 (* random recombination mask *)
 (* Print[r]; *)
 o = (p1 && r) || (p2 && !r); (* semantic crossover *)
 (* Print[o]; *)
 d = BooleanMinterms[{Table[RandomInteger[], {n}]}, v];
 (* Perturbing Term *)
 o = If[RandomInteger[] == 0, Or[o, d], And[o, Not[d]]]; (* Semantic Mutation *)
 (* Print[o]; *)
 fo = Total[Boole[BooleanTable[o]]]; (* Fitness of the Offspring *)
 (* Print[fo]; *)
 If[fo > fworst,
  pop[[posworst]] = Simplify[o, TimeConstraint → 0.1];
  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 *)
  , null]; (* Replace Parent if Offspring is better, and Simplify *)
]
Print[pop[[posbest]]]; (* Print the Optimum Solution *)
```

```
(* Equivalent Steady-State Evolutionary Algorithm on Output Vectors *)

n = 8;
k = Round[Sqrt[2^n]];
pop = Table[Table[RandomInteger[], {2^n}], {k}];
fpop = Table[Total[pop[[i]]], {i, k}];
fbest = Max[fpop];
posbest = Position[fpop, fbest][[1, 1]];
fworst = Min[fpop];
posworst = Position[fpop, fworst][[1, 1]];
For[i = 0, fbest < (2^n), i++,
 Print[i, " ", fbest];
 p1 = pop[[RandomInteger[{1, k}]]];
 p2 = pop[[RandomInteger[{1, k}]]];
 r = Table[RandomInteger[], {2^n}];
 o = Table[Mod[(p1[[j]] * r[[j]]) + (p2[[j]] * (1 - r[[j]])), 2], {j, 2^n}];
 o[[RandomInteger[{1, 2^n}]]] = RandomInteger[];
 fo = Total[o];
 If[fo > fworst,
  pop[[posworst]] = o;
  fpop[[posworst]] = fo;
  fbest = Max[fpop];
  posbest = Position[fpop, fbest][[1, 1]];
  fworst = Min[fpop];
  posworst = Position[fpop, fworst][[1, 1]];
  , null];
]
Print[pop[[posbest]]];
```

GP in Scala

- A compact framework for evolutionary computation in Scala
- Composed of two libraries: Scevo and Scaps
- Component assembly via mixins
- Interoperable with
- Links:
    - ScEvo
    - ScaPS

```scala
case class BooleanDomain(override val numVars: Int, instr: Option[Map[Int, List[Any]]] = None)
  extends DomainWithVars[Seq[Boolean], Boolean](numVars, instr) {

  final override def nonInputs = Map(
    1 -> List("!"),
    2 -> List("&", "|", "!&", "!|", "^"))

  override def semantics(input: Seq[Boolean], postOpHook: (Instruction, Boolean) => (Instruction, Boolean)) = {
    require(input.size == numVars)
    new Function1[Instruction, Boolean] {
      def apply(statement: Instruction): Boolean =
        postOpHook(statement, statement match {
          // Need to enforce evaluation of both arguments (for traces to have same length):
          // && - lazy, & - eager
          case Op("&", x, y)  => apply(x) & apply(y)
          case Op("!&", x, y) => !(apply(x) & apply(y))
          case Op("|", x, y)  => apply(x) | apply(y)
          case Op("!|", x, y) => !(apply(x) | apply(y))
          case Op("^", x, y)  => apply(x) ^ apply(y)
          case Op("!", x)     => !apply(x)
          case Op(i: Int)     => input(i)
          case Op(c: Boolean) => c
          case Op(op, _)      => throw new Exception("Invalid opcode: " + op)
        })._2
    }
  }
}

/* Use case #1: Simple GP run on a benchmark.
 * This class configures the experiment by combining the components via mixin
 */
class BooleanDefault(args: String)
  extends OptionsFromArgs(args) with Rng
  with BooleanBenchmark
  with SearchDrivers.Hamming[Seq[Boolean], Boolean]
  with CorrectnessPredicates.Strict[ScalarEvaluationMin]
  with TreeGPDefault[Seq[Boolean], Boolean, ScalarEvaluationMin] {
  def this(args: Array[String]) = this(args.mkString(" "))
}

class BooleanUseCase1 {
  @org.junit.Test
  def test: Unit = new BooleanDefault("--benchmark mux6 --maxGenerations 100 --instructions withNeg").launch
}
```

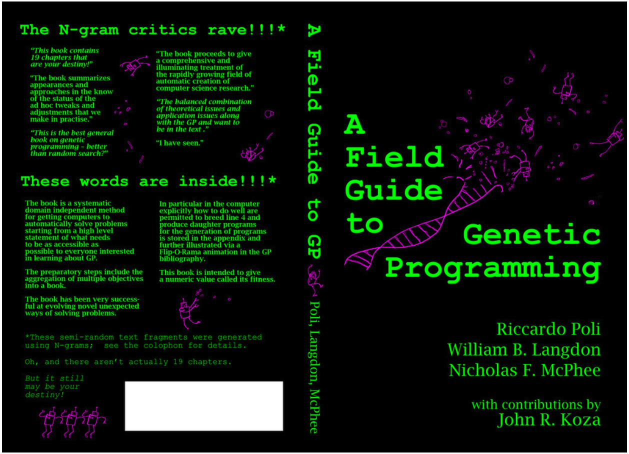
Additional resources

Recommended reading

- Koza, J. R. Genetic Programming: On the Programming of Computers by Means of Natural Selection MIT Press, 1992
- A Field Guide to Genetic Programming (ISBN 978-1-4092-0073-4) http://www.gp-field-guide.org.uk/
- Langdon, W. B. Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming! Kluwer, 1998
- Langdon, W. B. & Poli, R. Foundations of Genetic Programming Springer-Verlag, 2002
- Riolo, R. L.; Soule, T. & Worzel, B. (ed.) Genetic Programming Theory and Practice V Springer, 2007
- Riolo, R.; McConaghy, T. & Vladislavleva, E. (ed.) Genetic Programming Theory and Practice VIII Springer, 2010
- See: http://www.cs.bham.ac.uk/~wbl/biblio/

Recommended reading

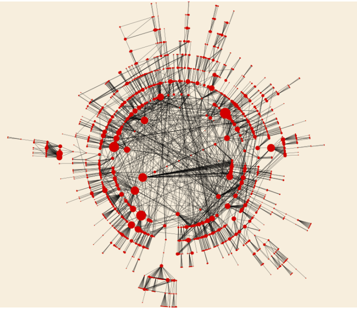
- A Field Guide to Genetic Programming
  http://www.gp-field-guide.org.uk/ [Poli et al., 2008]



(This presentation uses some figures from the Field Guide)

GP Bibliography and GP homepage

- The online GP bilbiography www.cs.bham.ac.uk/~wbl/biblio/



- The genetic programming 'home page'
  http://www.genetic-programming.com/

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.

Exercise 1: Mona Lisa (non-GP)

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:

```
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')

Exercise 4: Evolving agent's controller

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

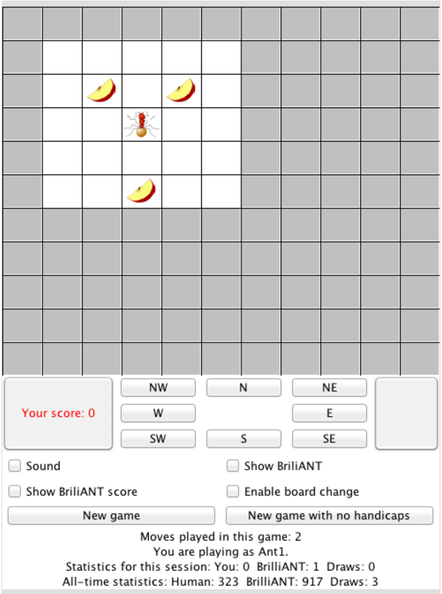
```
java -cp ../../../jar/ecj.22.jar ec.Evolve -file progn4.params
```

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

Demos

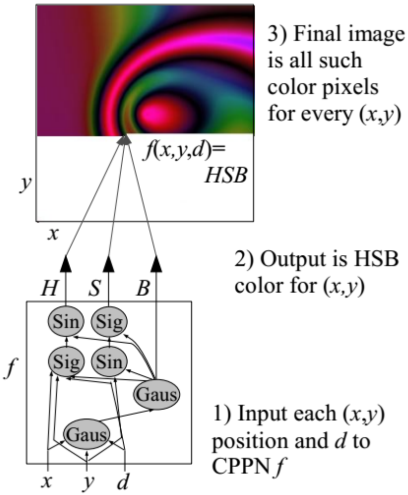
Ant Wars

- A two-person, zero-sum, partially observable, turn-based game used as a bencchmark 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 briliant online at http://www.cs.put.poznan.pl/kkrawiec/antwars/



PicBreeder

- 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/



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

Synthesizing fully-fledged programs

- Recursive sorting algorithms of $n \log n$ complexity using object-oriented GP [Kinnear, Jr., 1993b, Ryan & Nicolau, 2003, Ciesielski & Li, 2004, Spector et al., 2005, Agapitos & Lucas, 2006]
- 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, fibonaccci, 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

Assignment

Assignment: Instructions

I. Read **one** of the papers from the following list, focusing on the following issues:

- What is the question addressed in the paper?
- What data or evidence was collected by the author(s) to address the question?
- What did the data or evidence show?

II. Prepare a report (in English (preferably) or Polish) containing:

1. Your first and last name
2. Authors and the title of the paper
3. A few sentences about the strong (most interesting, intriguing) elements of the proposed approach
4. A few sentences about the weak points
5. Your individual thoughts/observations concerning the paper.
6. How could this be employed to solve some problems in your research area.

Email the report (plain text, no attachments!) to krawiec at cs.put.poznan.pl with "[SD]" tag in the email subject **by April 30th**.

Assignment: Instructions

There are two groups of papers to pick from:

1. Papers concerning program synthesis, in particular GP
2. Papers related to GP

You may choose a paper from either of these groups.

Assignment

1. Reading in program synthesis

Wadler, P. (1989). Theorems for free!
In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89 (pp. 347–359). New York, NY, USA: ACM

Abstract: From the type of a polymorphic function we can derive a theorem that it satisfies. Every function of the same type satisfies the same theorem. This provides a free source of useful theorems, courtesy of Reynolds' abstraction theorem for the polymorphic lambda calculus.

http://www.mpi-sws.org/~dreyer/tor/papers/wadler.pdf
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.9875

Schmidt, M. & Lipson, H. (2009). Distilling free-form natural laws from experimental data.
Science, 324(5923), 81–85

Abstract: For centuries, scientists have attempted to identify and document analytical laws that underlie physical phenomena in nature. Despite the prevalence of computing power, the process of finding natural laws and their corresponding equations has resisted automation. A key challenge to finding analytic relations automatically is defining algorithmically what makes a correlation in observed data important and insightful. We propose a principle for the identification of nontriviality. We demonstrated this approach by automatically searching motion-tracking data captured from various physical systems, ranging from simple harmonic oscillators to chaotic double-pendula. Without any prior knowledge about physics, kinematics, or geometry, the algorithm discovered Hamiltonians, Lagrangians, and other laws of geometric and momentum conservation. The discovery rate accelerated as laws found for simpler systems were used to bootstrap explanations for more complex systems, gradually uncovering the "alphabet" used to describe those systems.

http://www.sciencemag.org/content/324/5923/81.short
(plus accompanying material)

Paper #3: GP for fixing software bugs

Weimer, W., Forrest, S., Le Goues, C., & Nguyen, T. (2010). Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5), 109–116

Abstract: There are many methods for detecting and mitigating software errors but few generic methods for automatically repairing errors once they are discovered. This paper highlights recent work combining program analysis methods with evolutionary computation to automatically repair bugs in off-the-shelf legacy C programs. The method takes as input the buggy C source code, a failed test case that demonstrates the bug, and a small number of other test cases that encode the required functionality of the program. The repair procedure does not rely on formal specifications, making it applicable to a wide range of extant software for which formal specifications rarely exist.

http://dl.acm.org/ft_gateway.cfm?id=1735249&type=html

Paper #4: Behavioral GP

Krawiec, K. & O'Reilly, U.-M. (2014). Behavioral programming: a broader and more detailed take on semantic GP.
In C. Igel, D. V. Arnold, C. Gagne, E. Popovici, A. Auger, J. Bacardit, D. Brockhoff, S. Cagnoni, K. Deb, B. Doerr, J. Foster, T. Glasmachers, E. Hart, M. I. Heywood, H. Iba, C. Jacob, T. Jansen, Y. Jin, M. Kessentini, J. D. Knowles, W. B. Langdon, P. Larranaga, S. Luke, G. Luque, J. A. W. McCall, M. A. Montes de Oca, A. Motsinger-Reif, Y. S. Ong, M. Palmer, K. E. Parsopoulos, G. Raidl, S. Risi, G. Ruhe, T. Schaul, T. Schmickl, B. Sendhoff, K. O. Stanley, T. Stuetzle, D. Thierens, J. Togelius, C. Witt, & C. Zarges (Eds.), *GECCO '14: Proceedings of the 2014 conference on Genetic and evolutionary computation* (pp. 935–942). Vancouver, BC, Canada: ACM

Abstract: In evolutionary computation, the fitness of a candidate solution conveys sparse feedback. Yet in many cases, candidate solutions can potentially yield more information. In genetic programming (GP), one can easily examine program behavior on particular fitness cases or at intermediate execution states. However, how to exploit it to effectively guide the search remains unclear. In this study we apply machine learning algorithms to features describing the intermediate behavior of the executed program. We then drive the standard evolutionary search with additional objectives reflecting this intermediate behavior. The machine learning functions independent of task-specific knowledge and discovers potentially useful components of solutions (subprograms), which we preserve in an archive and use as building blocks when composing new candidate solutions. In an experimental assessment on a suite of benchmarks, the proposed approach proves more capable of finding optimal and/or well-performing solutions than control methods.

http://dl.acm.org/citation.cfm?id=2598288

Spector, L. (2001). Autoconstructive evolution: Push, pushGP, and pushpop. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, & E. Burke (Eds.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)* (pp. 137–146). San Francisco, California, USA: Morgan Kaufmann

Abstract: This paper is a preliminary report on autoconstructive evolution, a framework for evolutionary computation in which the machinery of reproduction and diversification (and thereby the machinery of evolution) evolves within the individuals of an evolving population of problem solvers. Autoconstructive evolution is illustrated with Pushpop, an evolving population of programs expressed in the Push programming language. The Push programming language can also be used in a more traditional genetic programming framework and may have unique benefits when so employed; the PushGP system, which uses traditional genetic programming techniques to evolve Push programs, is also described.

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.9569

Moraglio, A., Krawiec, K., & Johnson, C. G. (2012). Geometric semantic genetic programming.
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

Abstract: Traditional Genetic Programming (GP) searches the space of functions/programs by using search operators that manipulate their syntactic representation, regardless of their actual semantics/behaviour. Recently, semantically aware search operators have been shown to outperform purely syntactic operators. In this work, using a formal geometric view on search operators and representations, we bring the semantic approach to its extreme consequences and introduce a novel form of GP — Geometric Semantic GP (GSGP) — that searches directly the space of the underlying semantics of the programs. This perspective provides new insights on the relation between program syntax and semantics, search operators and fitness landscape, and allows for principled formal design of semantic search operators for different classes of problems. We derive specific forms of GSGP for a number of classic GP domains and experimentally demonstrate their superiority to conventional operators.

http://dl.acm.org/citation.cfm?id=2415038

Paper #7: Deductive approach to program synthesis

Manna, Z. & Waldinger, R. (1980). A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1), 90–121

https://pdfs.semanticscholar.org/ceb3/163c56465fda5fef591d0ff0a6c7f434a04d.pdf

Assignment

2. Reading related to general evolutionary computation

Stanley, K. O. (2007). Compositional pattern producing networks: A novel abstraction of development.
Genetic Programming and Evolvable Machines, 8(2), 131–162.
Special issue on developmental systems

Abstract: Natural DNA can encode complexity on an enormous scale. Researchers are attempting to achieve the same representational efficiency in computers by implementing developmental encodings, i.e. encodings that map the genotype to the phenotype through a process of growth from a small starting point to a mature form. A major challenge in in this effort is to find the right level of abstraction of biological development to capture its essential properties without introducing unnecessary inefficiencies. In this paper, a novel abstraction of natural development, called Compositional Pattern Producing Networks (CPPNs), is proposed. Unlike currently accepted abstractions such as iterative rewrite systems and cellular growth simulations, CPPNs map to the phenotype without local interaction, that is, each individual component of the phenotype is determined independently of every other component. Results produced with CPPNs through interactive evolution of two-dimensional images show that such an encoding can nevertheless produce structural motifs often attributed to more conventional developmental abstractions, suggesting that local interaction may not be essential to the desirable properties of natural encoding in the way that is usually assumed.

http://link.springer.com/article/10.1007%2Fs10710-007-9028-8

Paper #9: EC for modeling modularity in biological networks

Kashtan, N. & Alon, U. (2005). Spontaneous evolution of modularity and network motifs.
Proceedings of the National Academy of Sciences, 102(39), 13773–13778

Abstract: Biological networks have an inherent simplicity: they are modular with a design that can be separated into units that perform almost independently. Furthermore, they show reuse of recurring patterns termed network motifs. Little is known about the evolutionary origin of these properties. Current models of biological evolution typically produce networks that are highly nonmodular and lack understandable motifs. Here, we suggest a possible explanation for the origin of modularity and network motifs in biology. We use standard evolutionary algorithms to evolve networks. A key feature in this study is evolution under an environment (evolutionary goal) that changes in a modular fashion. That is, we repeatedly switch between several goals, each made of a different combination of subgoals. We find that such modularly varying goals lead to the spontaneous evolution of modular network structure and network motifs. The resulting networks rapidly evolve to satisfy each of the different goals. Such switching between related goals may represent biological evolution in a changing environment that requires different combinations of a set of basic biological functions. The present study may shed light on the evolutionary forces that promote structural simplicity in biological networks and offers ways to improve the evolutionary design of engineered systems.

http://www.pnas.org/content/102/39/13773.abstract

Bibliography

Agapitos, A. & Lucas, S. M. (2006).
Evolving efficient recursive sorting algorithms.
In G. G. Yen, L. Wang, P. Bonissone, & S. M. Lucas (Eds.), *Proceedings of the 2006 IEEE Congress on Evolutionary Computation* (pp. 9227–9234). Vancouver: IEEE Press.

Arcuri, A. & Yao, X. (2008).
A novel co-evolutionary approach to automatic software bug fixing.
In J. Wang (Ed.), *2008 IEEE World Congress on Computational Intelligence* (pp. 162–168). Hong Kong: IEEE Computational Intelligence Society IEEE Press.

Baier, C. & Katoen, J.-P. (2008).
Principles of Model Checking (Representation and Mind Series).
The MIT Press.

Cardamone, L., Mocci, A., & Ghezzi, C. (2011).
Dynamic synthesis of program invariants using genetic programming.
In A. E. Smith (Ed.), *Proceedings of the 2011 IEEE Congress on Evolutionary Computation* (pp. 617–624). New Orleans, USA: IEEE Computational Intelligence Society IEEE Press.

Castle, T. & Johnson, C. G. (2012).
Evolving high-level imperative program trees with strongly formed genetic programming.
In A. Moraglio, S. Silva, K. Krawiec, P. Machado, & C. Cotta (Eds.), *Proceedings of the 15th European Conference on Genetic Programming, EuroGP 2012*, volume 7244 of *LNCS* (pp. 1–12). Malaga, Spain: Springer Verlag.

Ciesielski, V. & Li, X. (2004).
Experiments with explicit for-loops in genetic programming.
In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation* (pp. 494–501). Portland, Oregon: IEEE Press.

Cuccu, G. & Gomez, F. (2011).
When novelty is not enough.
In *Proc. EvoApplications*.

Dawkins, R. (1996).
The Blind Watchmaker: Why the Evidence of Evolution Reveals a Universe Without Design.
Norton.

Dempsey, I., O'Neill, M., & Brabazon, A. (2006).
Adaptive Trading With Grammatical Evolution.
In *Proc. CEC*.

Dijkstra, E. W. (1988).
On the cruelty of really teaching computing science.
circulated privately.

Dijkstra, E. W. (n.d.).
On the reliability of programs.
circulated privately.

Durrett, G., Neumann, F., & O'Reilly, U.-M. (2011).
Computational Complexity Analysis of Simple Genetic Programming On Two Problems Modeling Isolated Program Semantics.
In *Proc. FOGA*.

Faitelson, D. (2010).
Program Synthesis from Domain Specific Object Models.
VDM Publishing.

Flach, P. A. & Lavrac, N. (2000).
The role of feature construction in inductive rule learning.

Galván-López, E., Swafford, J., O'Neill, M., & Brabazon, A. (2010).
Evolving a Ms. PacMan Controller Using Grammatical Evolution.
In *Applications of Evolutionary Computation*. Springer.

Gulwani, S. (2010a).
Dimensions in program synthesis.
In R. Bloem & N. Sharygina (Eds.), *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23* (pp.1̃).: IEEE.

Gulwani, S. (2010b).
Dimensions in program synthesis.
In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming* (pp. 13–24). Hagenberg, Austria: ACM.
Invited talk.

Gulwani, S., Harris, W. R., & Singh, R. (2012).
Spreadsheet data manipulation using examples.
Communications of the ACM, 55(8), 97–105.

Hansen, J. V., Lowry, P. B., Meservy, R. D., & McDonald, D. M. (2007).
Genetic Programming for Prevention of Cyberterrorism through Dynamic and Evolving Intrusion Detection.
Decision Support Systems, 43, 1362–1374.

Harding, S., Miller, J. F., & Banzhaf, W. (2010).
Developments in Cartesian Genetic Programming: self-modifying CGP.
GPEM, 11, 397–439.

Harris, W. R. & Gulwani, S. (2011).
Spreadsheet table transformations from examples.
In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11 (pp. 317–328). New York, NY, USA: ACM.

Jakobović, D. & Budin, L. (2006).
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.
In M. O'Neill, L. Vanneschi, S. Gustafson, A. I. Esparcia Alcazar, I. De Falco, A. Della Cioppa, & E. Tarantino (Eds.), *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science* (pp. 13–24). Naples: Springer.

Kashtan, N. & Alon, U. (2005).
Spontaneous evolution of modularity and network motifs.
Proceedings of the National Academy of Sciences, 102(39), 13773–13778.

Kendall, G., Parkes, A., & Spoerer, K. (2008).
A Survey of NP-Complete Puzzles.
International Computer Games Association Journal, 31(1), 13–34.

Kinnear, Jr., K. E. (1993a).
Evolving a Sort: Lessons in Genetic Programming.
In *Proc. of the International Conference on Neural Networks*.

Kinnear, Jr., K. E. (1993b).
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.

Koza, J. (1992a).
A Genetic Approach to the Truck Backer Upper Problem and the Inter-twined Spiral Problem.
In *Proc. International Joint Conference on Neural Networks*.

Koza, J. R. (1989).
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.

Koza, J. R. (1992b).
Genetic Programming: On the Programming of Computers by Means of Natural Selection.
MIT Press.

Koza, J. R. (1994).
Genetic Programming II: Automatic Discovery of Reusable Programs.
MIT Press.

Koza, J. R., Andre, D., Bennett III, F. H., & Keane, M. (1999).
Genetic Programming III: Darwinian Invention and Problem Solving.
Morgan Kaufman.

Koza, J. R., Bennett III, F. H., Andre, D., & Keane, M. A. (2003a).
Genetic programming problem solver with automatically defined stores loops and recursions.
United States Patent 6532453.

Koza, J. R., Keane, M. A., Streeter, M. J., Mydlowec, W., Yu, J., & Lanza, G. (2003b).
Genetic Programming IV: Routine Human-Competitive Machine Intelligence.
Kluwer Academic Publishers.

Krawiec, K. (2001).
Evolutionary computation framework for learning from visual examples.
Image Processing and Communications, 7(3-4), 85–96.

Krawiec, K. (2004).
Evolutionary Feature Programming: Cooperative learning for knowledge discovery and computer vision.
Number 385 in . Poznan University of Technology, Poznan, Poland: Wydawnictwo Politechniki Poznanskiej.

Krawiec, K. (2007).
Generative learning of visual concepts using multiobjective genetic programming.
Pattern Recognition Letters, 28(16), 2385–2400.

Krawiec, K. & Bhanu, B. (2005).
Visual learning by coevolutionary feature synthesis.
IEEE Transactions on System, Man, and Cybernetics – Part B, 35(3), 409–425.

Krawiec, K. & Bhanu, B. (2007).
Visual learning by evolutionary and coevolutionary feature synthesis.
IEEE Transactions on Evolutionary Computation, 11(5), 635–650.

Krawiec, K. & O'Reilly, U.-M. (2014).
Behavioral programming: a broader and more detailed take on semantic GP.
In C. Igel, D. V. Arnold, C. Gagne, E. Popovici, A. Auger, J. Bacardit, D. Brockhoff, S. Cagnoni, K. Deb, B. Doerr, J. Foster, T. Glasmachers, E. Hart, M. I. Heywood, H. Iba, C. Jacob, T. Jansen, Y. Jin, M. Kessentini, J. D. Knowles, W. B. Langdon, P. Larranaga, S. Luke, G. Luque, J. A. W. McCall, M. A. Montes de Oca, A. Motsinger-Reif, Y. S. Ong, M. Palmer, K. E. Parsopoulos, G. Raidl, S. Risi, G. Ruhe, T. Schaul, T. Schmickl, B. Sendhoff, K. O. Stanley, T. Stuetzle, D. Thierens, J. Togelius, C. Witt, & C. Zarges (Eds.), *GECCO '14: Proceedings of the 2014 conference on Genetic and evolutionary computation* (pp. 935–942). Vancouver, BC, Canada: ACM.

Krawiec, K. & Swan, J. (2013).
Pattern-guided genetic programming.
In C. Blum, E. Alba, A. Auger, J. Bacardit, J. Bongard, J. Branke, N. Bredeche, D. Brockhoff, F. Chicano, A. Dorin, R. Doursat, A. Ekart, T. Friedrich, M. Giacobini, M. Harman, H. Iba, C. Igel, T. Jansen, T. Kovacs, T. Kowaliw, M. Lopez-Ibanez, J. A. Lozano, G. Luque, J. McCall, A. Moraglio, A. Motsinger-Reif, F. Neumann, G. Ochoa, G. Olague, Y.-S. Ong, M. E. Palmer, G. L. Pappa, K. E. Parsopoulos, T. Schmickl, S. L. Smith, C. Solnon, T. Stuetzle, E.-G. Talbi, D. Tauritz, & L. Vanneschi (Eds.), *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference* (pp. 949–956). Amsterdam, The Netherlands: ACM.

Langdon, W. & Banzhaf, W. (2008).
Repeated Patterns in Genetic Programming.
Natural Computing, 7, 589–613.

Langdon, W. B. (2002).
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.

Langdon, W. B. & Banzhaf, W. (2005).
Repeated Sequences in Linear Genetic Programming Genomes.
Complex Systems, 15(4), 285–306.

Langdon, W. B. & Harrison, A. P. (2008).
Evolving Regular Expressions for GeneChip Probe Performance Prediction.
In *Proc. PPSN* (pp. 1061–1070).

Langdon, W. B. & Poli, R. (2002).
Foundations of Genetic Programming.
Springer-Verlag.

Langdon, W. B., Rowsell, J., & Harrison, A. P. (2009).
Creating Regular Expressions as mRNA Motifs with GP to Predict Human Exon Splitting.
In *Proc. GECCO*.

Langdon, W. B., Sanchez Graillet, O., & Harrison, A. P. (2010).
Automated DNA Motif Discovery.
arXiv.org.

Le Goues, C., Dewey-Vogt, M., Forrest, S., & Weimer, W. (2012).
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.

Lones, M., Tyrrell, A., Stepney, S., & Caves, L. (2010).
Controlling Complex Dynamics with Artificial Biochemical Networks.
In *Proc. EuroGP* (pp. 159–170).

Lucas, S. (2012a).
Othello Competition.
http:/\protect\kern-.1667em\relax/algoval.essex.ac.uk:8080/othello/html/Othello.html.

[Online; accessed 27-Jan-2012].

Lucas, S. (2012b).
The Physical Travelling Salesperson Problem.
http:/\protect\kern-.1667em\relax/algoval.essex.ac.uk/ptsp/ptsp.html.
[Online: accessed 27–Jan-2012].

Luke, S. (2010).
The ECJ Owner's Manual – A User Manual for the ECJ Evolutionary Computation Library, zeroth edition, online version 0.2 edition.

Manna, Z. & Waldinger, R. (1980).
A deductive approach to program synthesis.
ACM Trans. Program. Lang. Syst., 2(1), 90–121.

McConaghy, T. (2011).
FFX: Fast, Scalable, Deterministic Symbolic Regression Technology.
In *Proc. GPTP*.

Miller, J. F. & Thomson, P. (1998).
Evolving digital electronic circuits for real-valued function generation using a genetic algorithm.
In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, & R. Riolo (Eds.), *Genetic Programming 1998: Proceedings of the Third Annual Conference* (pp. 863–868). University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann.

Miller, J. F. & Thomson, P. (2000).
Cartesian genetic programming.
In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, & T. C. Fogarty (Eds.), *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS* (pp. 121–132). Edinburgh: Springer-Verlag.

Mitchell, T. M. (1997).
Machine Learning.
McGraw-Hill.

Montana, D. J. (1993).
Strongly Typed Genetic Programming.
BBN Technical Report #7866, Bolt Beranek and Newman, Inc., 10 Moulton Street, Cambridge, MA 02138, USA.

Moraglio, A., Krawiec, K., & Johnson, C. G. (2012).
Geometric semantic genetic programming.
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. (2015).
Automatic programming via iterated local search for dynamic job shop scheduling.
IEEE Transactions on Cybernetics, 45(1), 1–14.

Nicolau, M., Schoenauer, M., & Banzhaf, W. (2010).
Evolving Genes to Balance a Pole.
In *Proc. EuroGP*.

Nordin, P. & Banzhaf, W. (1995).
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.

Olague, G. & Trujillo, L. (2011).
Evolutionary-computer-assisted design of image operators that detect interest points using genetic programming.
Image and Vision Computing, 29(7), 484–498.

Olsson, R. (1998).
Population management for automatic design of algorithms through evolution.
In *Proceedings of the 1998 IEEE World Congress on Computational Intelligence* (pp. 592–597). Anchorage, Alaska, USA: IEEE Press.

O'Neill, M., Brabazon, A., & Hemberg, E. (2008).
Subtree Deactivation Control with Grammatical Genetic Programming in Dynamic Environments.
In *Proc. CEC*.

Paterson, M., Peres, Y., Thorup, M., Winkler, P., & Zwick, U. (2008).
Maximum Overhang.
In *Proc. 19th Annual ACM-SIAM Symposium on Discrete Algorithms*.

Poli, R. (2001).
Exact schema theory for genetic programming and variable-length genetic algorithms with one-point crossover.
Genetic Programming and Evolvable Machines, 2(2), 123–163.

Poli, R. & Langdon, W. B. (1998).
On the search properties of different crossover operators in genetic programming.
In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, & R. Riolo (Eds.), *Genetic Programming 1998: Proceedings of the Third Annual Conference* (pp. 293–301). University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann.

Poli, R., Langdon, W. B., & McPhee, N. F. (2008).
A field guide to genetic programming.
Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).

Ross, B. J. & Zhu, H. (2004).
Procedural texture evolution using multiobjective optimization.
New Generation Computing, 22(3), 271–293.

Ryan, C., Collins, J. J., & O'Neill, M. (1998).
Grammatical evolution: Evolving programs for an arbitrary language.
In W. Banzhaf, R. Poli, M. Schoenauer, & T. C. Fogarty (Eds.), *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS* (pp. 83–96). Paris: Springer-Verlag.

Ryan, C. & Nicolau, M. (2003).
Doing genetic algorithms the genetic programming way.
In R. L. Riolo & B. Worzel (Eds.), *Genetic Programming Theory and Practice* chapter 12, (pp. 189–204). Kluwer.

Salustowicz, R. P. & Schmidhuber, J. (1997).
Probabilistic incremental program evolution.
Evolutionary Computation, 5(2), 123–141.

Schmidhuber, J. (1987).
Evolutionary principles in self-referential learning. on learning now to learn: The meta-meta-meta...-hook.
Diploma thesis, Technische Universitat Munchen, Germany.

Schmidt, M. & Lipson, H. (2009).
Distilling free-form natural laws from experimental data.
Science, 324(5923), 81–85.

Schumacher, C., Vose, M. D., & Whitley, L. D. (2001).
The no free lunch and problem description length.
In L. Spector & E. D. Goodman (Eds.), *GECCO 2001: Proc. of the Genetic and Evolutionary Computation Conf.* (pp. 565–570). San Francisco: Morgan Kaufmann.

Silva, S. & Vanneschi, L. (2010).
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*.

Sipper, M. (2011).
Let the Games Evolve!
In *Proc. GPTP*.

Spector, L. (2001).
Autoconstructive evolution: Push, pushGP, and pushpop.
In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, & E. Burke (Eds.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)* (pp. 137–146). San Francisco, California, USA: Morgan Kaufmann.

Spector, L. (2010).
Towards practical autoconstructive evolution: Self-evolution of problem-solving genetic programming systems.
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.
In M. Keijzer, G. Antoniol, C. B. Congdon, K. Deb, B. Doerr, N. Hansen, J. H. Holmes, G. S. Hornby, D. Howard, J. Kennedy, S. Kumar, F. G. Lobo, J. F. Miller, J. Moore, F. Neumann, M. Pelikan, J. Pollack, K. Sastry, K. Stanley, A. Stoica, E.-G. Talbi, & I. Wegener (Eds.), *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation* (pp. 1291–1298). Atlanta, GA, USA: ACM.

Spector, L., Klein, J., & Keijzer, M. (2005).
The push3 execution stack and the evolution of control.
In H.-G. Beyer, U.-M. O'Reilly, D. V. Arnold, W. Banzhaf, C. Blum, E. W. Bonabeau, E. Cantu-Paz, D. Dasgupta, K. Deb, J. A. Foster, E. D. de Jong, H. Lipson, X. Llora, S. Mancoridis, M. Pelikan, G. R. Raidl, T. Soule, A. M. Tyrrell, J.-P. Watson, & E. Zitzler (Eds.), *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2 (pp. 1689–1696). Washington DC, USA: ACM Press.

Spector, L., Perry, C., & Klein, J. (2004).
Push 2.0 Programming Language Description.
Technical report, School of Cognitive Science, Hampshire College.

Srivastava, S., Gulwani, S., Chaudhuri, S., & Foster, J. (2010).
Program Inversion Revisited.
Technical Report MSR-TR-2010-34, Microsoft Research.

Stanley, K. O. (2007).
Compositional pattern producing networks: A novel abstraction of development.
Genetic Programming and Evolvable Machines, 8(2), 131–162.
Special issue on developmental systems.

Togelius, J., Karakovskiy, S., Koutnik, J., & Schmidhuber, J. (2009).
Super Mario Evolution.
In *Proc. IEEE Computational Intelligence and Games*.

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

Vladislavleva, E., Smits, G., & Den Hertog, D. (2009).
Order of Nonlinearity as a Complexity Measure for Models Generated by Symbolic Regression via Pareto Genetic Programming.
IEEE Trans EC, 13(2), 333–349.

Wadler, P. (1989).
Theorems for free!
In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89 (pp. 347–359). New York, NY, USA: ACM.

Wadler, P. (2014).
Propositions as types.

Wagner, N., Michalewicz, Z., Khouja, M., & McGregor, R. (2007).
Time Series Forecasting for Dynamic Environments: The DyFor Genetic Program Model.
IEEE Trans EC.

Walker, J. & Miller, J. (2007).
Predicting Prime Numbers Using Cartesian Genetic Programming.
In *Proc. EuroGP*.

Walker, J. A., Völk, K., Smith, S. L., & Miller, J. F. (2009).
Parallel Evolution using Multi-chromosome Cartesian Genetic Programming.
GPEM, 10, 417–445.

Warren, H. S. (2002).
Hacker's Delight.
Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Weimer, W., Forrest, S., Le Goues, C., & Nguyen, T. (2010).
Automatic program repair with evolutionary computation.
Communications of the ACM, 53(5), 109–116.

Whitley, L. D. & Sutton, A. M. (2009).
Elementary landscape analysis.
In *GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference* (pp. 3227–3236). New York, NY, USA: ACM.

Widera, P., Garibaldi, J., & Krasnogor, N. (2010).
GP challenge: Evolving energy function for protein structure prediction.
GPEM, 11, 61–88.

Wolpert, D. H. & Macready, W. G. (1997).
No free lunch theorems for optimization.
IEEE Trans. on Evolutionary Computation, 1(1), 67–82.

Yu, T. (2001).
Hierarchical Processing for Evolving Recursive and Modular Programs Using Higher-Order Functions and Lambda Abstraction.
GPEM, 2, 345–380.