Genetic Programming

Krzysztof Krawiec

Laboratory of Intelligent Decision Support Systems
Institute of Computing Science, Poznan University of Technology, Poznań, Poland
http://www.cs.put.poznan.pl/kkrawiec/

June 10, 2016

Introduction

Outline and objectives

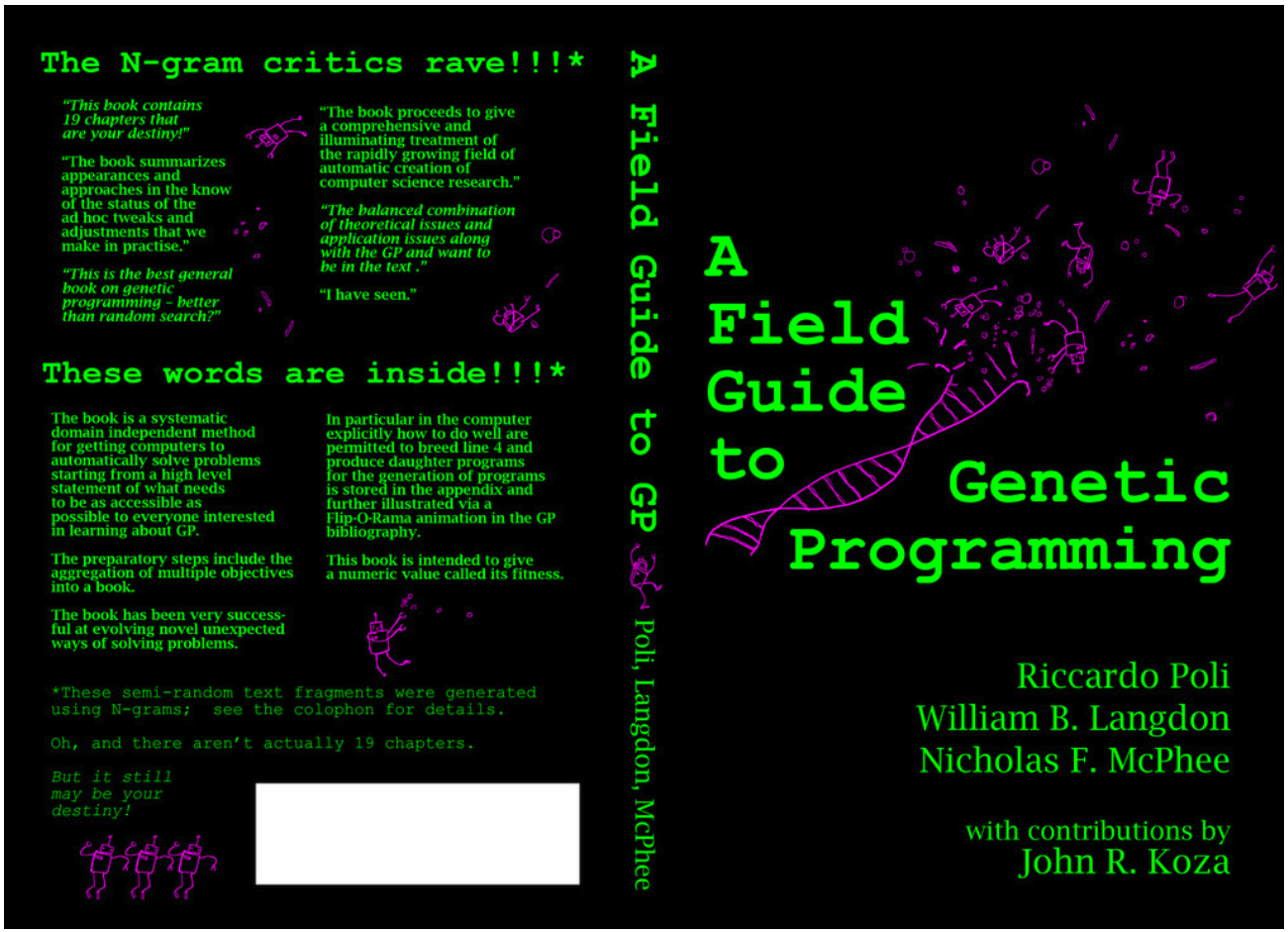
1. Introduction: GP as a variant of EC
2. Specific features of GP
3. Variants of GP
4. Applications
5. Some theory
6. Case studies

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/` [39]



(This presentation uses some figures from the Field Guide)

Background

Evolutionary Computation (EC)

- Heuristic bio-inspired global search algorithms
- 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.

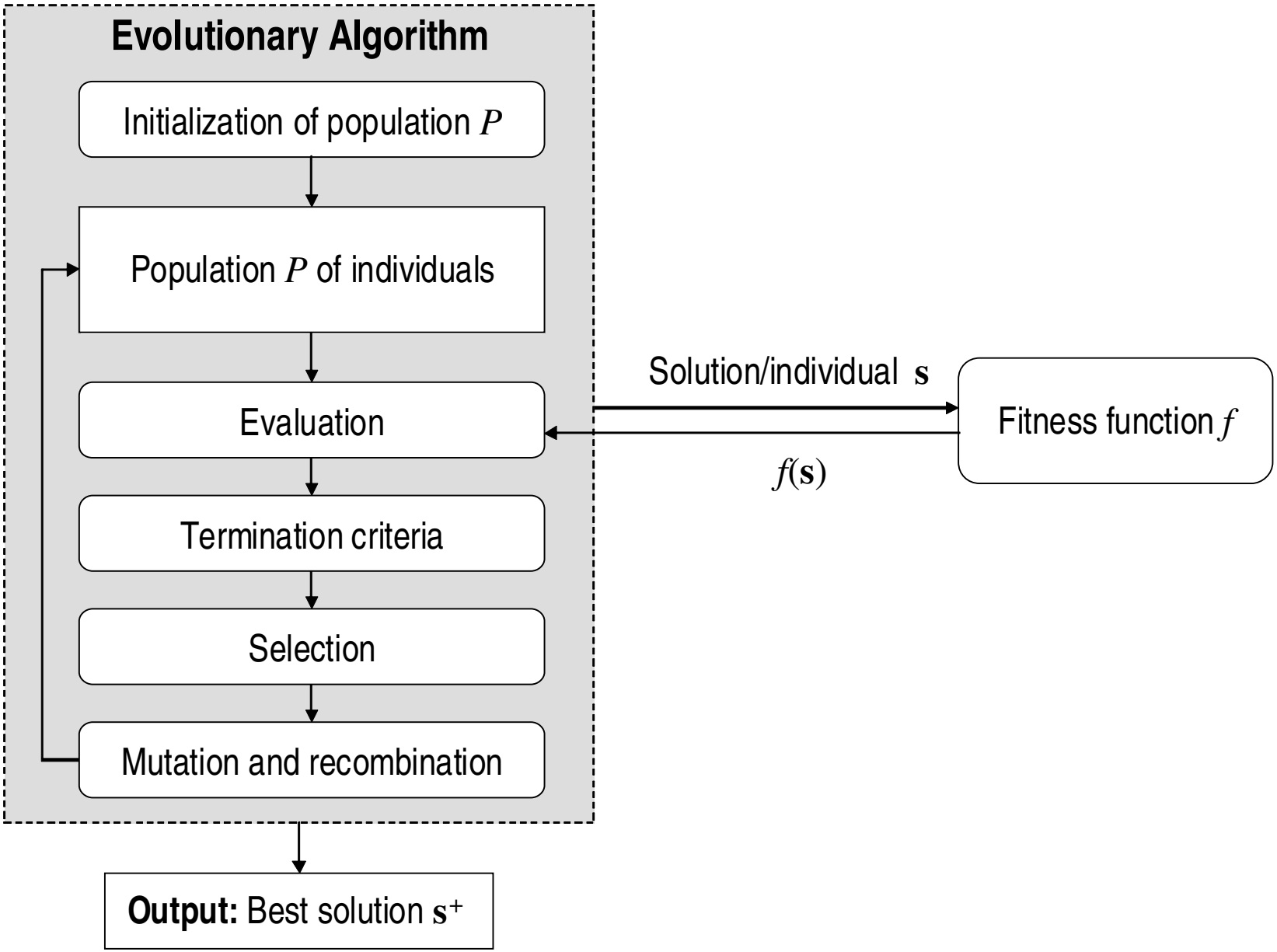
Formulation:

$$p^* = \arg\max_{p \in P} f(p)$$

where

- P is the considered space (*search space*) of *candidate solutions* (*solutions* for short)
- f is a (maximized) fitness function
- p^* is an *optimal solution* (an *ideal*) that maximizes f.

Generic evolutionary algorithm



[Unique] characteristic of EC

- Black-box optimization (f's dependency on the independent variables does not have to be known or meet any criteria)
- Variables do not have to be explicitly defined
- Fining an optimum cannot be guaranteed, but in practice a well-performing suboptimal solution is often satisfactory.
- 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

What is genetic programming?

Genetic programming

In a nutshell:

- A variant of EC where the genotypes represent *programs*, i.e., entities capable of reading in input data and producing some output data in response to that input.
- Fitness function f measures the similarity of the output produced by the program to the desired output, given as a part of task statement.
- Standard representation: expression trees.

Important implication: Additional input required by the algorithm (compared to EC):

- Set of instructions (programming language of consideration).
- Data to run the programs on.

Conceptual preliminaries of GP: Space of candidate solutions P

- Candidate solutions $p \in P$ evolving under the selection pressure of the fitness function f are themselves functions of the form $p : I \to O$,
  - I and O are, respectively, the spaces of input data and output data accepted and produced by programs from P.
- Cardinality of $|P|$ is typically large or infinite.

- 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.

Conceptual preliminaries: Fitness function f

- In most cases (and most real-world applications of GP), fitness function f measures the similarity of the output produced by the program to the desired output, given as a part of task statement.
- Then, fitness can be expressed as a monotonous function of the divergence of program's output from the desired one, for instance as:

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

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.

Conceptual preliminaries: Character of candidate solutions

- The candidate solutions in GP are being assembled from elementary entities called *instructions*.
- A part of formulation of a GP task is then also an instruction set $\mathscr{I}$, i.e., a set of symbols used by the search algorithm to compose the programs (candidate solutions).
- Design of $\mathscr{I}$ usually requires some background knowledge;
  - In particular, it should comprise all instructions necessary to find solution to the problem posed (closure).

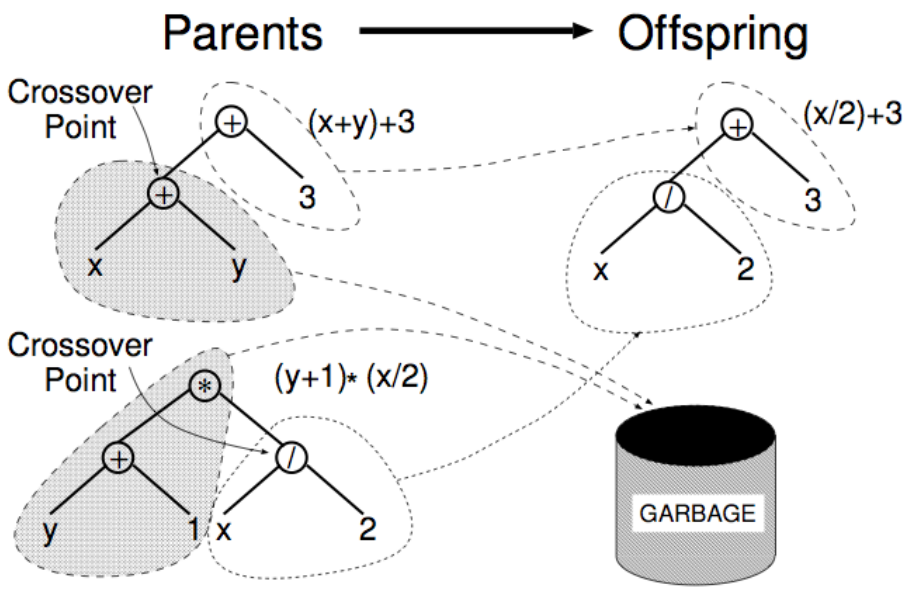
Main evolution loop ('vanilla GP')

```
1:  procedure GeneticProgramming(f, 𝒥)                ▷ f - fitness function, 𝒥 - instruction set
2:      𝒫 ← {p ← RandomProgram(𝒥)}                    ▷ Initialize population
3:      repeat                                        ▷ Main loop over generations
4:          for p ∈ 𝒫 do                              ▷ Evaluation
5:              p.f ← f(p)                            ▷ p.f is a 'field' in program p that stores its fitness
6:          end for
7:          𝒫′ ← ∅                                    ▷ Next population
8:          repeat                                    ▷ Breeding loop
9:              p1 ← TournamentSelection(𝒫)           ▷ First parent
10:             p2 ← TournamentSelection(𝒫)           ▷ Second parent
11:             (o1, o2) ← Crossover(p1, p2)
12:             o1 ← Mutation(o1, 𝒥)
13:             o2 ← Mutation(o2, 𝒥)
14:             𝒫′ ← 𝒫′ ∪ {o1, o2}
15:         until |𝒫′| = |𝒫|
16:         𝒫 ← 𝒫′
17:     until StoppingCondition(𝒫)
18:     return arg max_{p∈𝒫} p.f
19: end procedure
```

Crossover

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

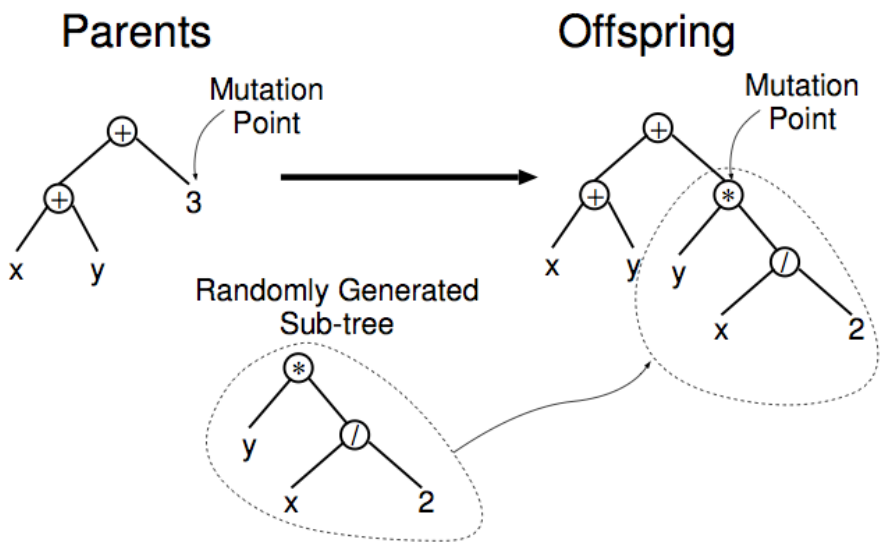
1: **function** Crossover(p_1, p_2)
2: **repeat**
3: $s_1 \leftarrow$ Random node in p_1
4: $s_2 \leftarrow$ Random node in p_2
5: $(p'_1, p'_2) \leftarrow$ Swap subtrees rooted in s_1 and s_2
6: **until** Depth(p'_1) $< d_{max} \wedge$ Depth(p'_2) $< d_{max}$ ▷ d_{max} is the tree depth limit
7: **return**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



Mutation

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

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



Exemplary run: Setup

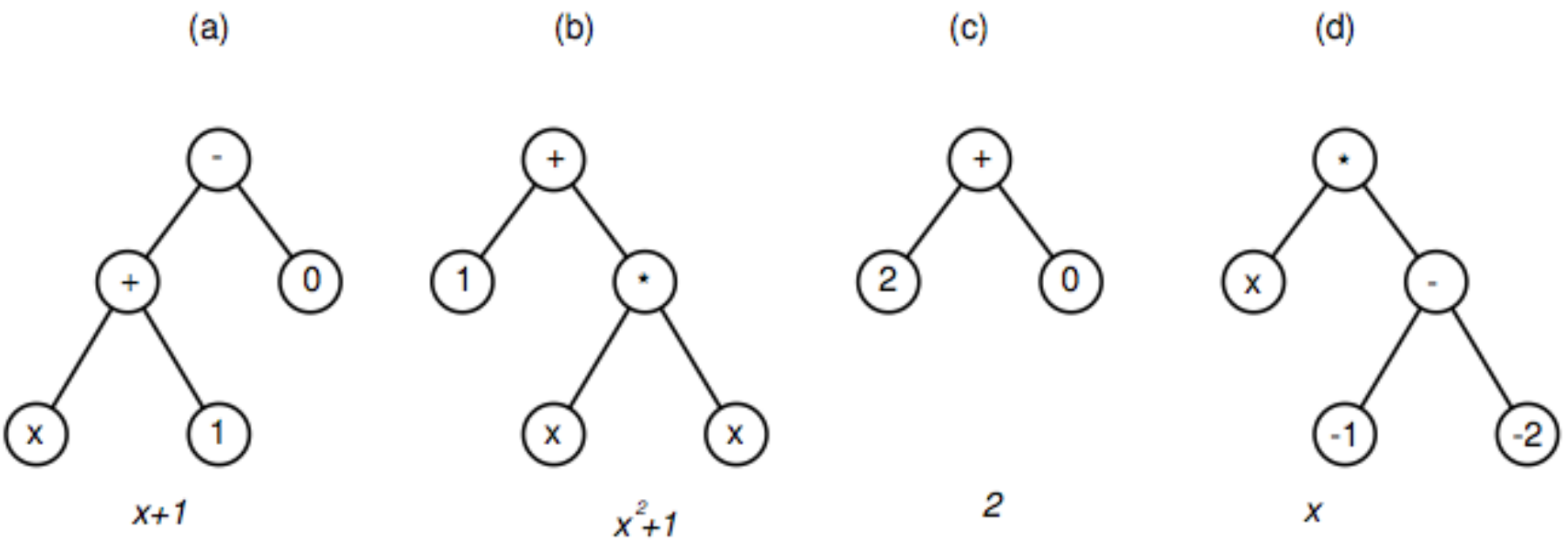
- Objective: Find program whose output matches x^2+x+1 over the range $[-1,1]$.
  - Such tasks can be considered as a form of regression.
  - As solutions are built by manipulating code (instructions), this is referred to as *symbolic regression*.
- Fitness: sum of absolute errors for $x \in -1.0, -0.9,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

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

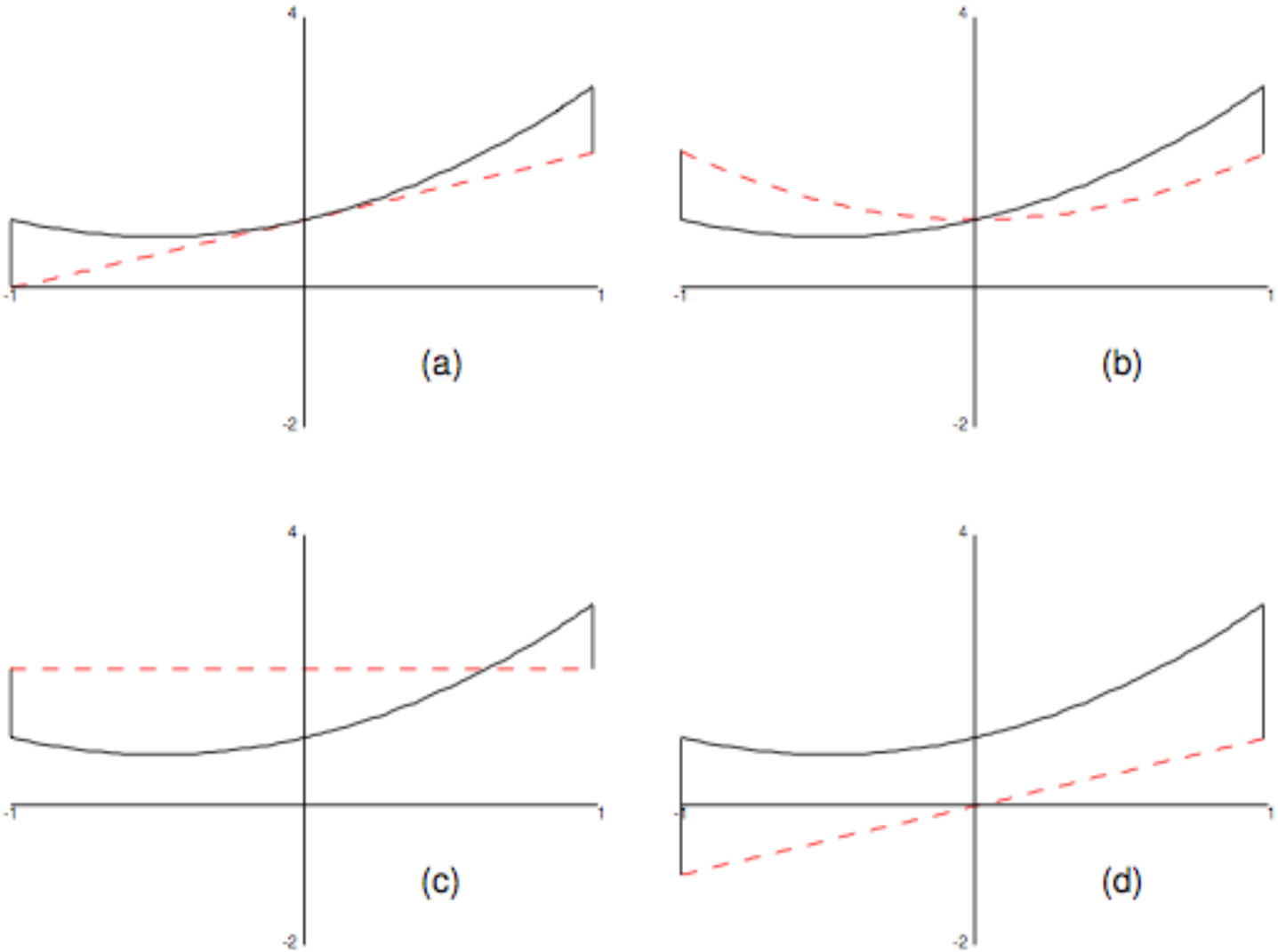
Exemplary run: Setup

- Instruction set:
  - Nonterminal (function) set: +, -, % (protected division), and x; all operating on floats
  - Terminal set: x, and constants chosen randomly between -5 and +5
- Selection: fitness proportionate (roulette wheel) non elitist
- Initial pop: ramped half-and-half (depth 1 to 2. 50% of terminals are constants)
  - (to be explained later)
- 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

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

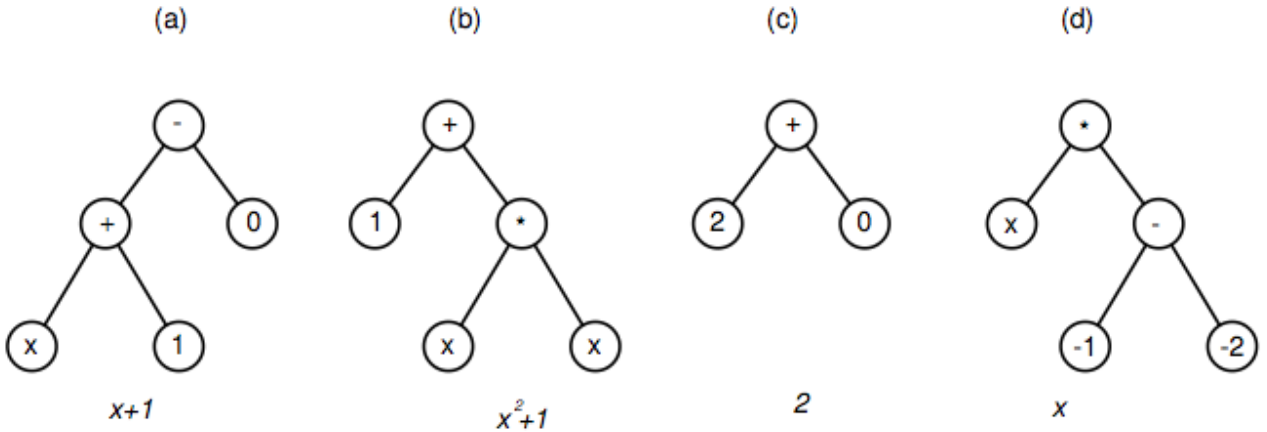
Breeding

Assume:

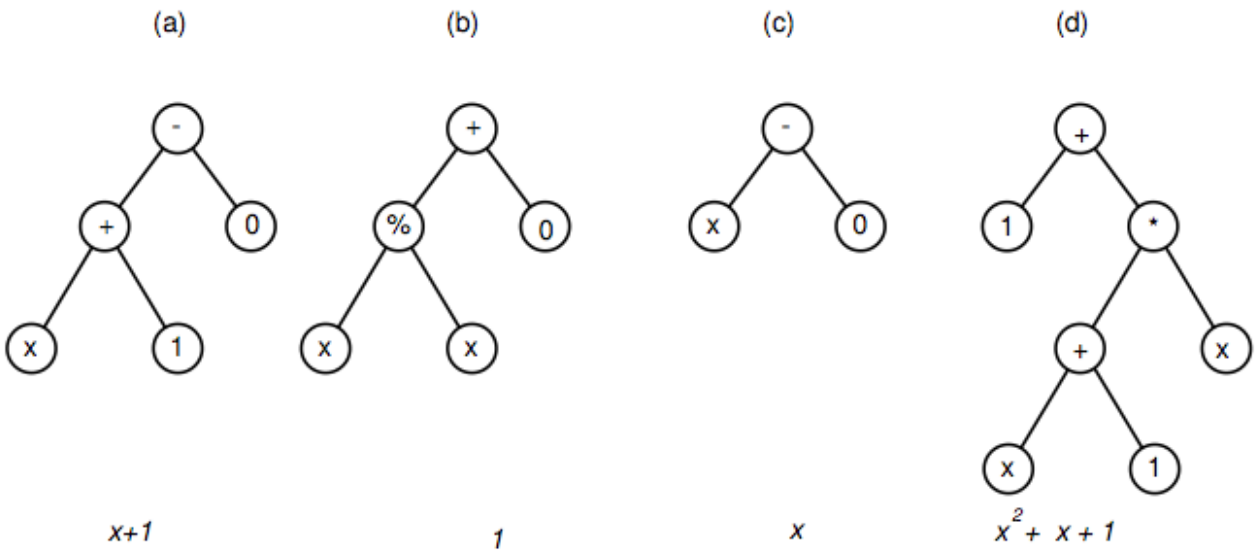
- a gets reproduced
- c gets mutated (at loci 2)
- a and d get crossed-over
- a and b get crossed over

Population 1

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 lengths*.
  - 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).

$\Longrightarrow$ 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 [**?**].

In a broader context

- 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).
- GP is not the only approach to program induction (but probably the best one :)
  - See, e.g., inductive logic programming, ILP
- GP embodies the ultimate goal of AI: to build a system capable of self-programming (adaptation, learning).

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.

Why should GP be considered a viable approach to program synthesis?

GP combines two powerful concepts marked in underline in the above definition:

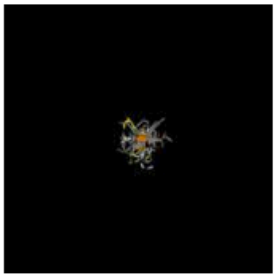
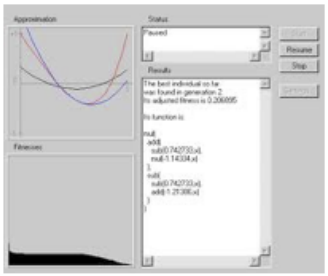
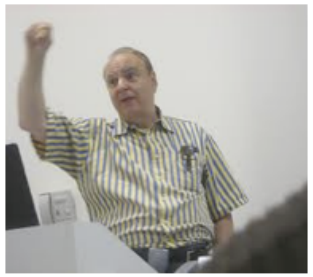
1. **Representing candidate solutions as programs**,
   which in general can conduct any Turing-complete computation (e.g., classification, regression, clustering, reasoning, problem solving, etc.), and thus enable capturing solutions to any type of problems (whether the task is, e.g., learning, optimization,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

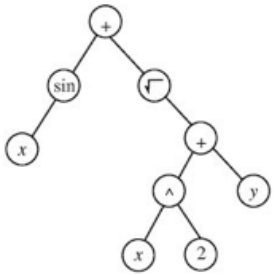
Consequences of the above definition

- Heuristic nature of search.
- Symbolic program representation.
- Input sensitivity and inductive character.
- State-fullness.
- Unconstrained data types.

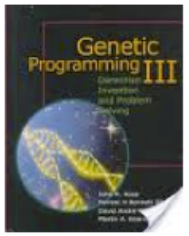
Origins of GP

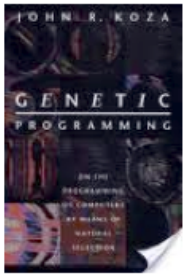








http://www.genetic-programming.com/johnkoza.html

Home assignment: Indentify John Koza's unique visual trait :)

Life demonstration of GP using ECJ

GP on ECJ

- ECJ, Evolutionary Computation in Java, `http://cs.gmu.edu/~eclab/projects/ecj/`
  - by Sean Luke, Liviu Panait, Gabriel Balan, Sean Paus, Zbigniew Skolicki, Elena Popovici, Keith Sullivan, *et al*.
- 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 March 2012: version 20.
- 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-related ECJ features

- 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)

Exemplary run of ECJ

Standard output:

```
java ec.Evolve -file ./ec/app/regression/quinticerc.params
...
Threads:  breed/1 eval/1
Seed: 1427743400
Job: 0
Setting up
Processing GP Types
Processing GP Node Constraints
Processing GP Function Sets
Processing GP Tree Constraints
{-0.13063322286594392,0.016487577414659428},
{0.6533404396941143,0.1402200189629743},
{-0.03750634856569701,0.0014027712093654706},
...
{0.6602806044824949,0.13869498395598084},
Initializing Generation 0
Subpop 0 best fitness of generation: Fitness: Standardized=1.1303205 Adjusted=0.46941292 I
Generation 1
Subpop 0 best fitness of generation: Fitness: Standardized=0.6804932 Adjusted=0.59506345 I
...
```

Exemplary run: The result

The log file produced by the run:

```
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))))
```

GP in R

```
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

GP 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]]];
```

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

GP Benchmarks

Problem	Definition (formula)
Sextic	$x^6 - 2x^4 + x^2$
Septic	$x^7 - 2x^6 + x^5 - x^4 + x^3 - 2x^2 + x$
Nonic	$x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x$
R1	$(x+1)^3/(x^2 - x + 1)$
R2	$(x^5 - 3x^3 + 1)/(x^2 + 1)$
R3	$(x^6 + x^5)/(x^4 + x^3 + x^2 + x + 1)$

Even more GP benchmarks ...

Symbolic Regression

Tower [52] ...

Boolean Functions

N-Multiplexer [17], N-Majority [17], N-Parity [17]

Generalised Boolean Circuits [12, 59]

Digital Adder [55]

Order [9]

Digital Multiplier [55]

Majority [9]

Classification

mRNA Motif Classification [24]

DNA Motif Discovery [25]

Intrusion Detection [11]

Protein Classification [19]

Intertwined Spirals [17]

Predictive Modelling

Mackey-Glass Chaotic Time Series [21]

Financial Trading [5, 4, 8]

Sunspot Prediction [17]

GeneChip Probe Performance [22]

Prime Number Prediction [54]

Drug Bioavailability [43]

Protein Structure Classification [58]

Time Series Forecasting [53]

Path-finding and Planning

Physical Travelling Salesman [28]

Artificial Ant [17]

Lawnmower [18]

Tartarus Problem [6]

Maximum Overhang [36]

Circuit Design [29]

Control Systems

Chaotic Dynamic Systems Control [26]

Pole Balancing [31]

Truck Control [16]

... and more ....

Game-Playing

TORCS Car Racing [50]

Ms PacMan [10]

Othello [27]

Chessboard Evaluation [44]

Backgammon [44]

Mario [49]

NP-Complete Puzzles [14]

Robocode [44]

Rush Hour [44]

Checkers [44]

Freecell [44]

Dynamic Optimisation

Dynamic Symbolic Regression [34, 35, 51]

Dynamic Scheduling [13]

Traditional Programming

Sorting [15, 1]

Case study

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

- 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
- Konferencja Algorytmów Ewolucyjnych i Optymalizacji Globalnej, KAEiOG, September 21, 2011

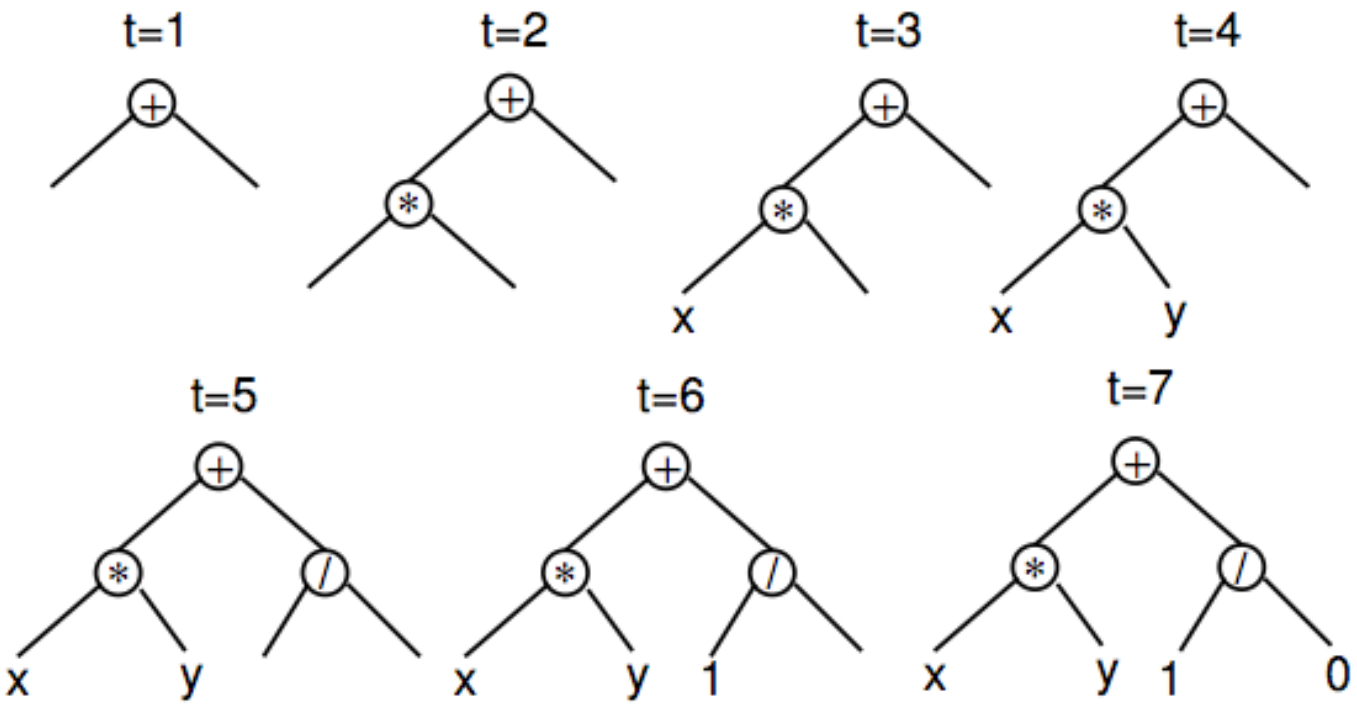
A more detailed view on GP (vanilla GP is not the whole story)

Where to get [random] candidate solutions from?

- Every stochastic search method relies on some 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 most widely used contemporary initialization methods take care only of the syntax of generated programs.
  - Mainly: height constraint.

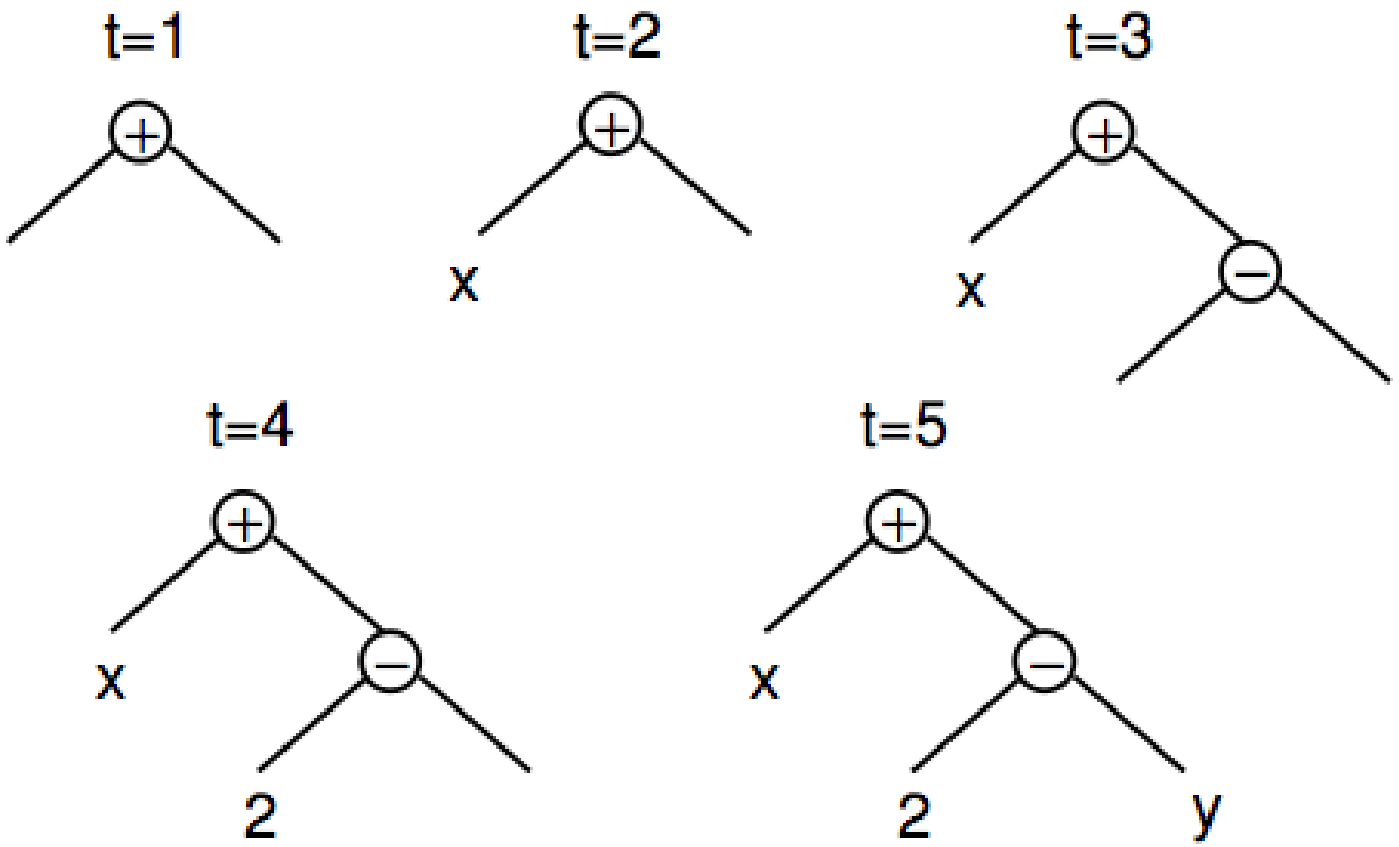
Initialization: *Full* method

- 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.



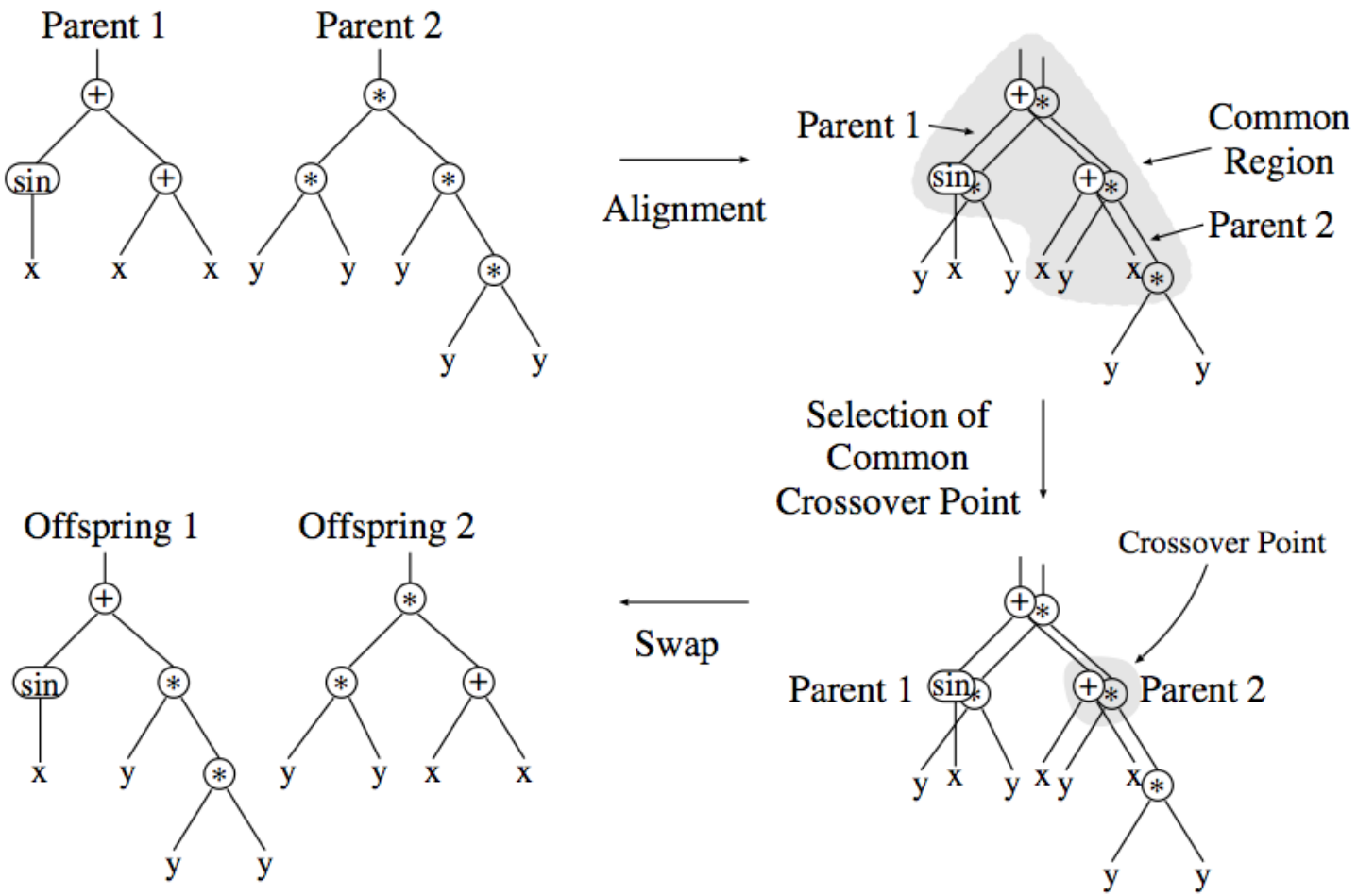
Initialization: *Grow* method

- 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.



Homologous crossover for GP

- Earliest example: one-point crossover [23]: 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 [38]

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 (and in EC probably too).

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
```

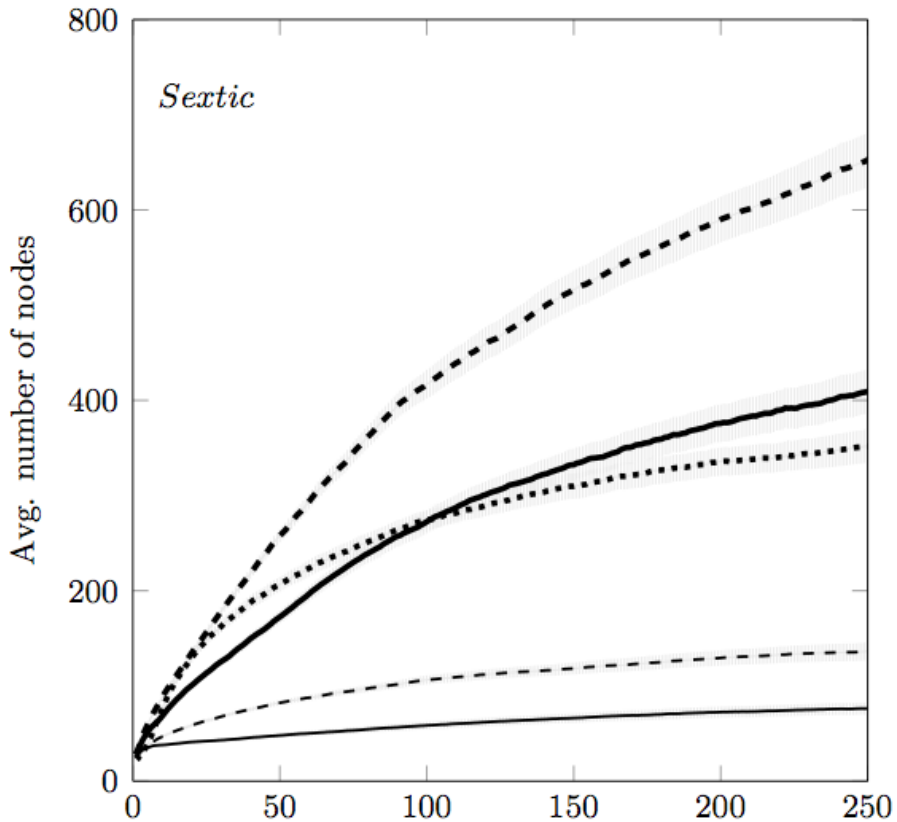
The Challenges for GP

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: 240+ papers as of March, 2012.

Bloat example

Average number of nodes per generation in a typical run of GP solving the *Sextic* problem $x^6 - 2x^4 + x^2$.



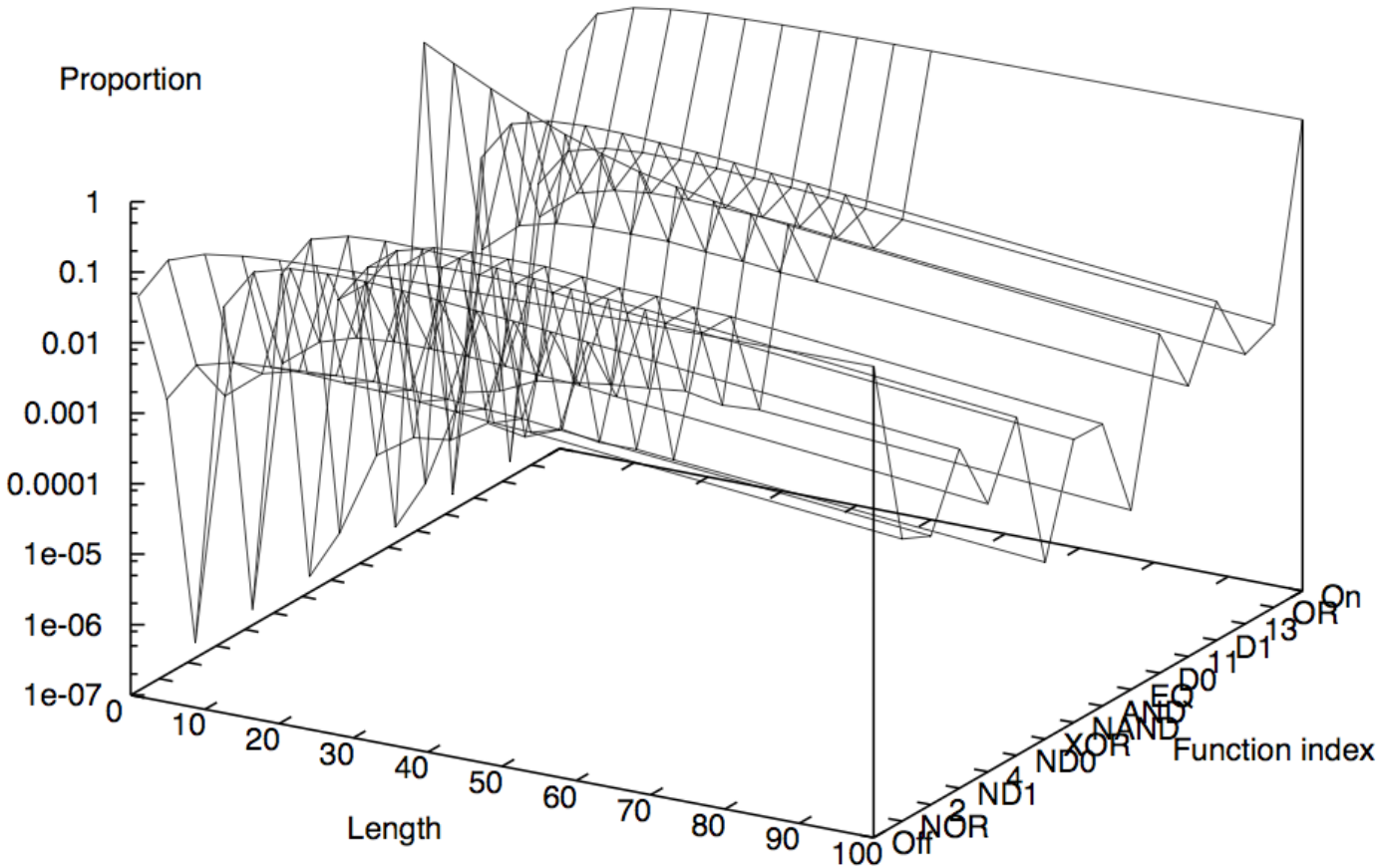
(GP: dotted line)

Countermeasures for bloat

- Constraining tree height
  - Surprisingly, 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)



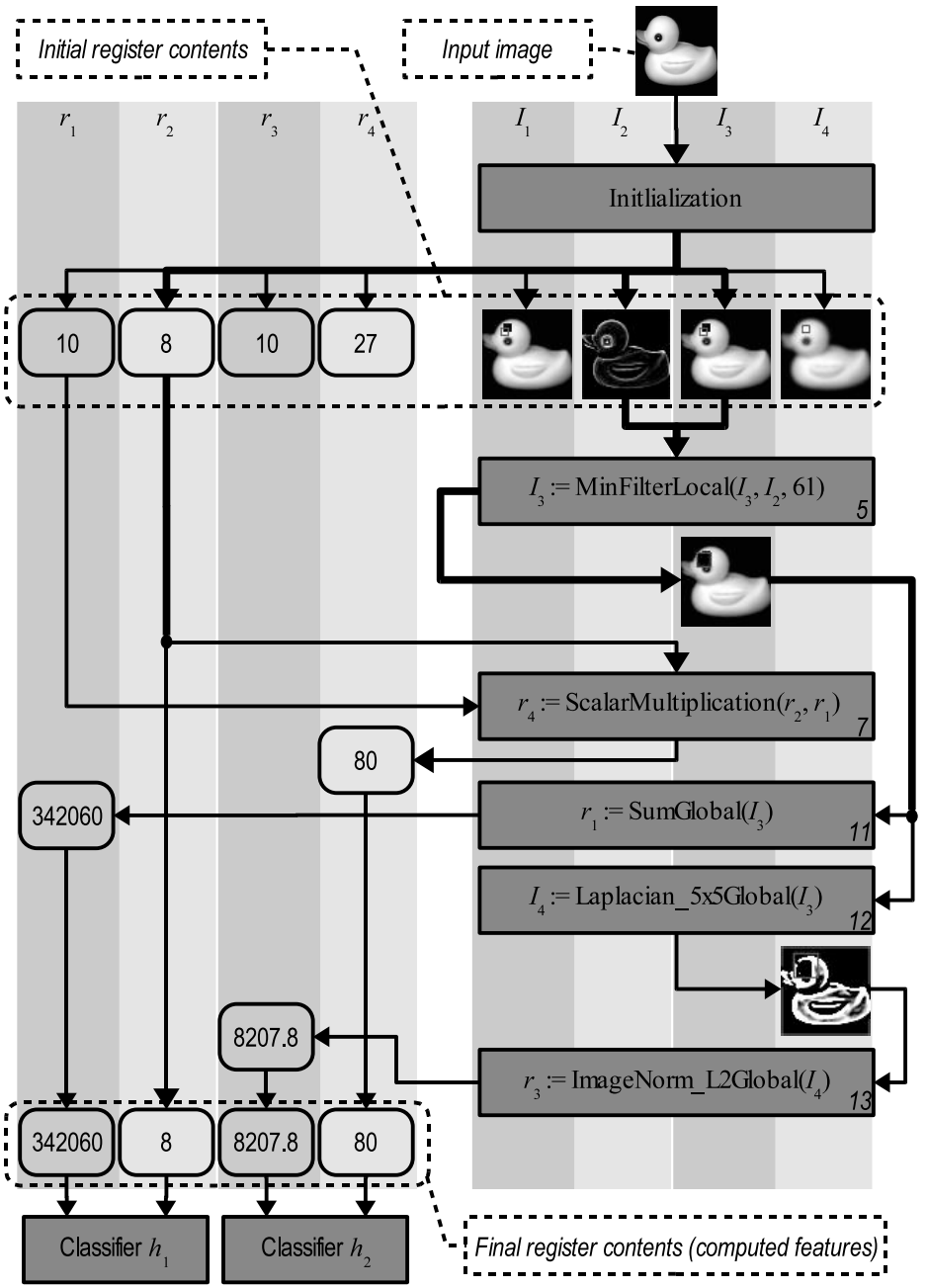
From: [20] Langdon, W. B. Cantú-Paz, E. (ed.) Random Search is Parsimonious Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO-2002), AAAI, 2002, 308-315

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



Variants of GP

Strongly typed GP (STGP)

- A way to incorporate prior knowledge and impose a structure on programs [30]
- Implementation:
  - Provide a set of types
  - For each instruction, define the types of its arguments and outcomes
  - Make the operators type-aware:
    - Mutation: substitute a random tree of a proper type
    - Crossover: swap trees of compatible[1] types

[1] Compatible: belonging to the same 'set type'

Strongly typed GP in ECJ

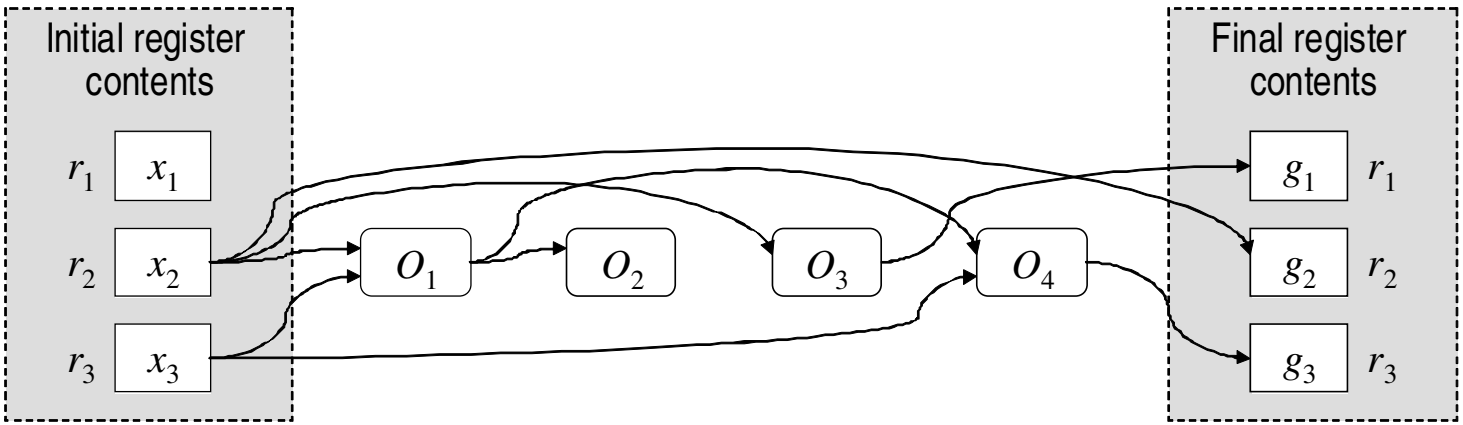
For the problem of simple classifiers represented as decision trees:

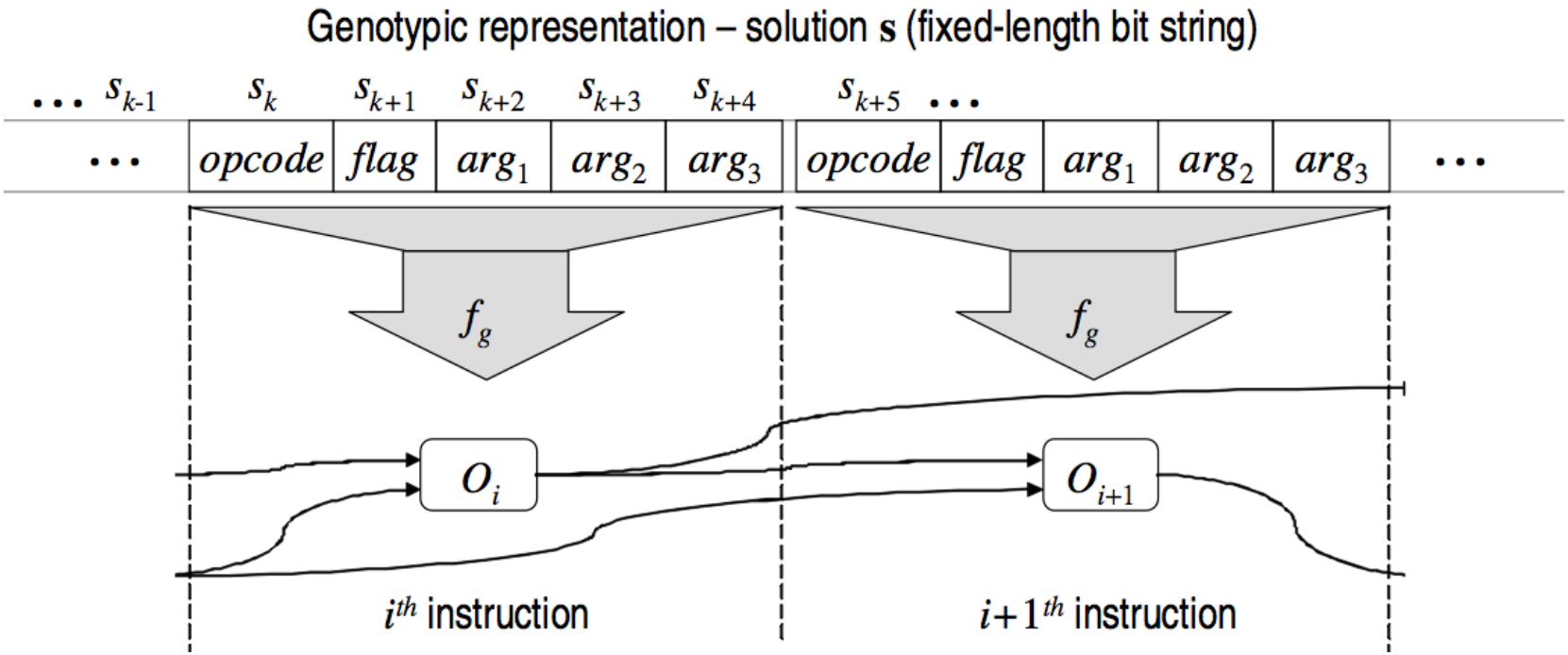
Classifier syntax:
Classifier ::= Class_id
Classifier ::= if_then_else(Condition, Classifier, Classifier)
Condition ::= Input_Variable = Constant_Value

Implementation in ECJ parameter files:

```
gp.type.a.size = 3
gp.type.a.0.name = class
gp.type.a.1.name = var
gp.type.a.2.name = const
gp.type.s.size = 0
gp.tc.size = 1
gp.tc.0 = ec.gp.GPTreeConstraints
gp.tc.0.name = tc0
gp.tc.0.fset = f0
gp.tc.0.returns = class
gp.nc.size = 4
gp.nc.0 = ec.gp.GPNodeConstraints
gp.nc.0.name = ncSimpleClassifier
gp.nc.0.returns = class
gp.nc.0.size = 0
gp.nc.1 = ec.gp.GPNodeConstraints
gp.nc.1.name = ncCompoundClassifier
gp.nc.1.returns = class
gp.nc.1.size = 4
gp.nc.1.child.0 = var
gp.nc.1.child.1 = const
gp.nc.1.child.2 = class
gp.nc.1.child.3 = class
gp.nc.2 = ec.gp.GPNodeConstraints
gp.nc.2.name = ncVariable
gp.nc.2.returns = var
gp.nc.2.size = 0
gp.nc.3 = ec.gp.GPNodeConstraints
gp.nc.3.name = ncConstant
gp.nc.3.returns = const
gp.nc.3.size = 0
```

- Motivation:
  - Tree-like structures are not natural for contemporary hardware architectures
- Program = a sequence of instructions
- Data passed via registers
- Pros:
  - Directly portable to machine code, fast execution.
  - Natural correspondence to standard (GA-like) crossover operator.
- Applications: direct evolution of machine code [32].




Genotypic representation – solution s (fixed-length bit string)
s_{k-1} s_k s_{k+1} s_{k+2} s_{k+3} s_{k+4} s_{k+5}
opcode flag arg_1 arg_2 arg_3 opcode flag arg_1 arg_2 arg_3
f_g
f_g
O_i
O_{i+1}
i^{th} instruction
$i+1^{th}$ instruction

Stack-based GP

- The best-known representative: Push and PushGP
  `hampshire.edu/lspector/push.html` [48]
- Pros:
  - Very simple syntax: program ::= instruction | literal | (program*)
  - No need to specify the number of registers
  - The top element of a stack has the natural interpretation of program outcome
  - Natural possibility of implementing autoconstructive programs [47]
  - Includes certain features that make it Turing-complete (e.g., YANK instruction).

Push: Example

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 )
```

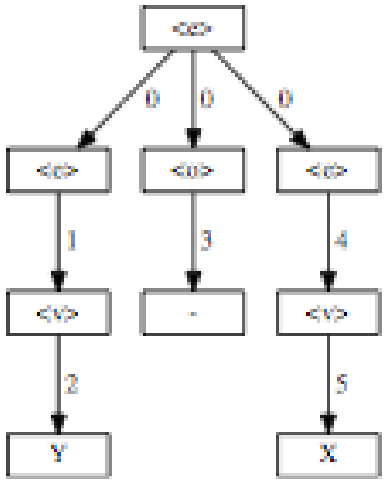
http://hampshire.edu/lspector/push3-description.html

Other variants of GP

- Grammatical Evolution: The grammar of the programming language of consideration is given as input to the algorithm. 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
```



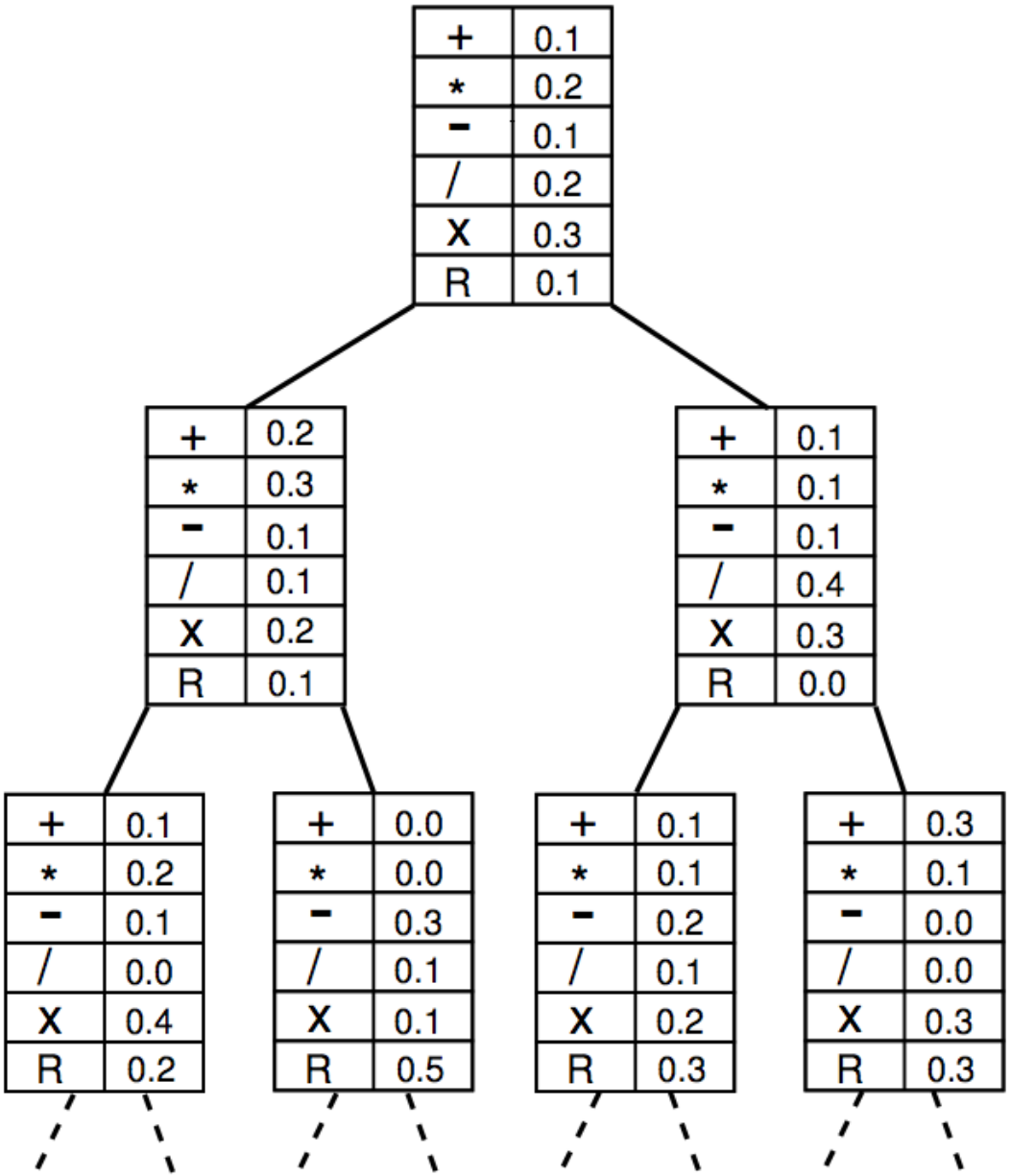
- Graph-based GP
  - Motivation: standard GP cannot reuse subprograms (within a single program)
  - Example: Cartesian Genetic Programming

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 [40])
- Developmental GP (e.g., using Push)
- Probabilistic GP (a variant of EDA, Estimation of Distribution Algorithms):
  - The algorithm maintains a probability distribution P instead of a population
  - Individuals are generated from P 'on demand'
  - The results of individuals' evaluation are used to update P

Simple EDA-like GP: PIPE

Probabilistic Incremental Program Evolution [41]

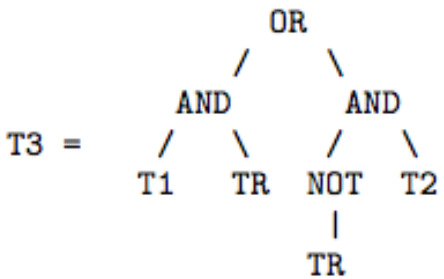


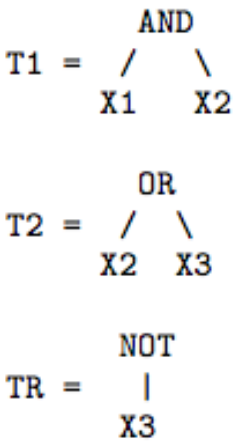
Selected theoretical results

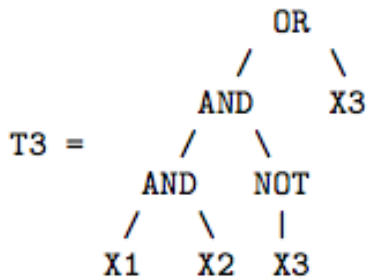
Schemata theorem for GP

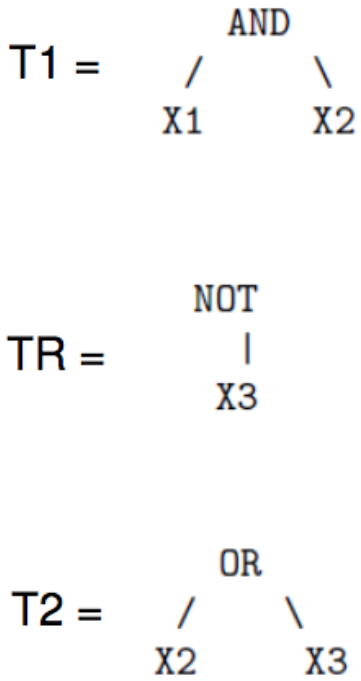
- Exact formula for the expected number of individuals sampling a schema a the next generation [37]
  - Plus later work for other types of crossover.

Semantic crossover









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[2].
- 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[3].
- 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[4].

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

[3] Arcuri, A., Yao, X., A novel co-evolutionary approach to automatic software bug fixing. In: Wang, J. (Ed.), 2008 IEEE World Congress on Computational Intelligence. IEEE Computational Intelligence Society, IEEE Press, Hong Kong.

[4] Schmidt, M., Lipson, H., 3 Apr. 2009. Distilling free-form natural laws from experimental data. Science 324 (5923), 81–85.

Humies

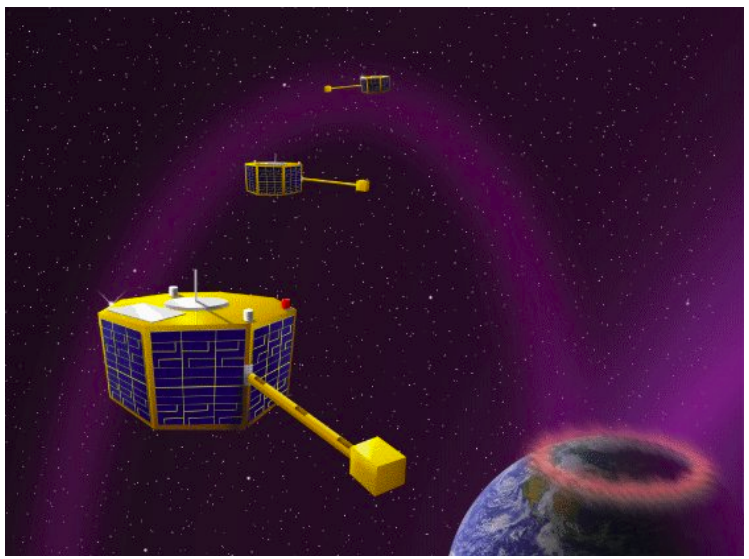
(...) *Entries were solicited for cash awards for human-competitive results that were produced by any form of genetic and evolutionary computation and that were published*

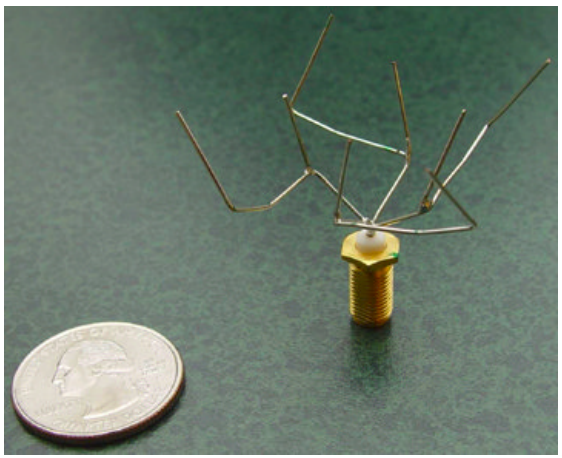
ANNUAL "HUMIES" AWARDS
FOR HUMAN-COMPETITIVE RESULTS
PRODUCED BY GENETIC AND EVOLUTIONARY COMPUTATION
HELD AT THE
ANNUAL GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE

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

- 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`

Selected Gold Humies using GP

- 2009: Stephanie Forrest Claire Le Goues ThanhVu Nguyen Westley Weimer Automatically finding patches using genetic programming: A Genetic Programming Approach to Automated Software Repair

```
1  void zunebug(int days) {
2    int year = 1980;
3    while (days > 365)  {
4      if (isLeapYear(year)){
5        if (days > 366)  {
6          days -= 366;
7          year += 1;
8        }
9        else {
10       }
11     }
12     else {
13       days -= 365;
14       year += 1;
15     }
16   }
17   printf("current year is %d\n", year);
18 }
```

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 GP challenge: evolving the energy function for protein structure prediction Automated design of energy functions for protein structure prediction by means of genetic programming and improved structure similarity assessment
- 2011: Achiya Elyasaf Ami Hauptmann Moshe Sipper
  GA-FreeCell: Evolving Solvers for the Game of FreeCell

Other applications

- classification problems in machine learning [?], object recognition [?, 33], or
- learning game strategies [?] .
- [2, 57] has demonstrated the ability of a GP system to automatically find and correct bugs in commercially-released software when provided with test data.
- In the context of this paper, it deserves particular attention that 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.
- In this spirit, in their seminal paper [42] have shown how GP can be used to induce scientific laws from experimental data. Many other studies have demonstrated the usefulness of GP for modeling different phenomena, including those of natural origins [45, 3, 7, 56, 46].

See [?] for an extensive review of GP applications.

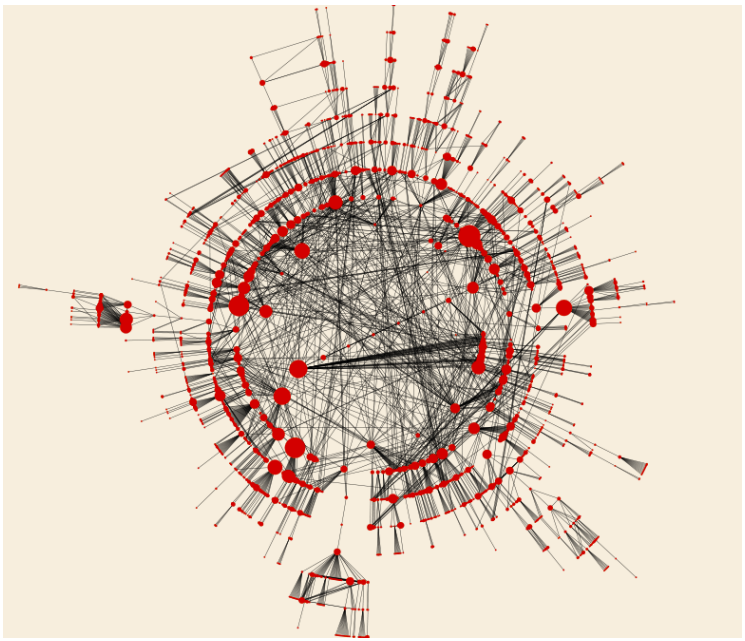
Case study 2: Evolution of feature detectors

Based on: Krzysztof Krawiec, Bartosz Kukawka and Tomasz Maciejewski (2010) *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.

Additional resources

Additional resources

- Evolutionary Computation in Java `cs.gmu.edu/~eclab/projects/ecj/`
  - Generic software framework for EA, well-prepared to work with GP
- The online GP bilbiography `www.cs.bham.ac.uk/~wbl/biblio/`



- The genetic programming 'home page' (a little bit messy, but still valuable) `http://www.genetic-programming.com/`

Bibliography


A. Agapitos and S. M. Lucas.
Evolving Modular Recursive Sorting Algorithms.
In *Proc. EuroGP*, 2007.


A. Arcuri and X. Yao.
A novel co-evolutionary approach to automatic software bug fixing.
In J. Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, pages 162–168, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.


M. Arganis, R. Val, J. Prats, K. Rodriguez, R. Dominguez, and J. Dolz.
Genetic programming and standardization in water temperature modelling.
Advances in Civil Engineering, 2009, 2009.


A. Brabazon, M. O'Neill, and I. Dempsey.
An Introduction to Evolutionary Computation in Finance.
IEEE Computational Intelligence Magazine, 3(4):42–55, 2008.


R. Bradley, A. Brabazon, and M. O'Neill.
Dynamic High Frequency Trading: A Neuro-Evolutionary Approach.
In *Proc. EvoWorkshops*, 2009.


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


M. Daga and M. C. Deo.
Alternative data-driven methods to estimate wind from waves by inverse modeling.
Natural Hazards, 49(2):293–310, May 2009.


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


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


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


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


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


D. Jakobović and L. Budin.
Dynamic Scheduling with Genetic Programming.
In *Proc. EuroGP*, 2006.


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


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


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


J. R. Koza.
Genetic Programming: On the Programming of Computers by Means of Natural Selection.
MIT Press, Cambridge, MA, USA, 1992.


J. R. Koza.
Genetic Programming II: Automatic Discovery of Reusable Programs.
MIT Press, Cambridge Massachusetts, May 1994.


W. Langdon and W. Banzhaf.
Repeated Patterns in Genetic Programming.
Natural Computing, 7:589–613, 2008.


W. B. Langdon.
Random search is parsimonious.
In E. Cantú-Paz, editor, *Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO-2002)*, pages 308–315, New York, NY, 9-13 July 2002. AAAI.


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


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


W. B. Langdon and R. Poli.
Foundations of Genetic Programming.
Springer-Verlag, 2002.


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


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


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


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


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


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


D. J. Montana.
Strongly typed genetic programming.
BBN Technical Report #7866, Bolt Beranek and Newman, Inc., 10 Moulton Street, Cambridge, MA 02138, USA, 7 May 1993.


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


P. Nordin and W. Banzhaf.
Genetic programming controlling a miniature robot.
In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 61–67, MIT, Cambridge, MA, USA, 10–12 Nov. 1995. AAAI.


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


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


M. O'Neill and C. Ryan.
Grammatical Evolution by Grammatical Evolution: The Evolution of Grammar and Genetic Code.
In *Proc. EuroGP*, pages 138–149. Springer-Verlag, 5–7 Apr. 2004.


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


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


R. Poli and W. B. Langdon.
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, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 293–301, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.


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


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


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


M. Schmidt and H. Lipson.
Distilling free-form natural laws from experimental data.
Science, 324(5923):81–85, 3 Apr. 2009.


S. Silva and L. Vanneschi.
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*, 2010.


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


C. Sivapragasam, R. Maheswaran, and V. Venkatesh.
Genetic programming approach for flood routing in natural channels.
Hydrological Processes, 22(5):623–628, 2007.


C. Sivapragasam, N. Muttil, S. Muthukumar, and V. M. Arun.
Prediction of algal blooms using genetic programming.
Marine Pollution Bulletin, 60(10):1849–1855, 2010.


L. Spector.
Towards practical autoconstructive evolution: Self-evolution of problem-solving genetic programming systems.
In R. Riolo, T. McConaghy, and E. Vladislavleva, editors, *Genetic Programming Theory and Practice VIII*, volume 8 of *Genetic and Evolutionary Computation*, chapter 2, pages 17–33. Springer, Ann Arbor, USA, 20-22 May 2010.


L. Spector, C. Perry, and J. Klein.
Push 2.0 programming language description.
Technical report, School of Cognitive Science, Hampshire College, Apr. 2004.


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


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


L. Vanneschi and G. Cuccu.
Variable Size Population for Dynamic Optimization with Genetic Programming.
In *Proc. GECCO*, 2009.


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


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


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


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


W.-C. Wang, K.-W. Chau, C.-T. Cheng, and L. Qiu.
A comparison of performance of several artificial intelligence methods for forecasting monthly discharge time series.
Journal of Hydrology, 374(3-4):294–306, 2009.


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


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


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