Genetic Programming

A Tutorial Introduction

Una-May O’Reilly
Evolutionary Design & Optimization Group

unamay@csail.mit.edu

Copyright is held by the author/owner(s).

GECCO’11, July 12–16, 2011, Dublin, Ireland.
ACM 978-1-4503-0690-4/11/07.
Agenda

CONTEXT: Evolutionary computation and algorithms
- Zoom in from evolutionary principles
- Evolutionary Algorithms: programming the biological evolutionary process
- Genetic programming in relation to other evolutionary algorithms

1. Genetic evolution of executable expressions
2. Example: Block Stacking problem
3. Nuts and Bolts Descriptions of Algorithm Components
4. Finish with more example problems
5. How does it work?
6. Issues and Challenges
Introductions

- Leader of Evolutionary Design and Optimization group, MIT CSAIL
- Focus on solving real world, complex problems requiring machine learning including evolutionary computation for classification, optimization and regression
- Applications include
  - Circuits, network coding
  - Sparse matrix data mapping on parallel architectures
  - Finance, Flavor design
  - Wind energy
    » Turbine layout
    » Resource assessment
    » Cabling
  - ICU clinical data mining
- Algorithms: Eas
  - Legacy in GP since 1993
Tutorial Goals

- Introductory
- Recognize GP design properties when you hear about it
- Teach it in an undergrad lecture
- Try it “out of the box” - with software libraries of others
- Preparation for advanced topics
  - Theory
  - Specialized workshops - S/R, bloat, etc
  - GP Track talks at GECCO, EuroGP
Neo-Darwinian Evolution

- Survival and thriving in the environment
- Offspring quantity - based on survival of the fittest
- Offspring variation: genetic crossover and mutation
- Population-based adaptation over generations
Problem Types and Applications

- Generating complex solutions - evolution is a process that gives rise to complexity
  - a continually evolving, adapting process, potentially with changing environment from which emerges modularity, hierarchy, complex behavior and complex system relationships

- GA: discrete variables - Combinatorial optimization
  - NP-complete and/or poorly scaling solutions via LP or convex optimization
  - unyielding to approximations (SQP, GEO-P)
  - eg. TSP, graph coloring, bin-packing, flows
  - for: logistics, planning, scheduling, networks, bio gene knockouts
Problem Types and Applications

- **ES: continuous variables: Continuous Optimization**
  - non-differentiable, discontinuous, multi-modal, large scale objective functions
  - for: engineering, mechanical, material, physics

- **Genetic Programming**
  - system identification aka symbolic regression
    » chemical processes, financial strategies
  - design: creative blueprints, generative designs - antennae, Genr8, chairs, lens
  - automatic programming: compiler heuristics
  - AI ODEs, invariants, knowledge discovery
Key EA Components

POPULATION
- array of struct ind with fields genome, phenotype fitness
- random initialization

- GENOME is an array of gene(s)
- GENOME is input parameter to decoder procedure that returns PHENOME
- PHENOME is input parameter to fitness-evaluation routine that returns a numeric variable called FITNESS
## EA Examples

<table>
<thead>
<tr>
<th>Problem</th>
<th>Gene</th>
<th>Genome</th>
<th>Phenotype</th>
<th>Fitness Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSP</td>
<td>110</td>
<td>sequence of cities</td>
<td>tour</td>
<td>tour length</td>
</tr>
<tr>
<td>Function optimization</td>
<td>3.21</td>
<td>variables x_of function</td>
<td>f(x)</td>
<td></td>
</tr>
<tr>
<td>graph k-coloring</td>
<td>permutation element</td>
<td>sequence for greedy coloring</td>
<td>coloring</td>
<td></td>
</tr>
<tr>
<td>investment strategy</td>
<td>rule</td>
<td>agent rule set</td>
<td>trading strategy</td>
<td>portfolio change</td>
</tr>
</tbody>
</table>
EA Generation Loop

- **generations**
  - select
  - breed
  - replace

```plaintext
population = random_pop_init()
generation = 0
while needToStop == false
  generation++
  phenotypes = decoder(genotypes)
  calculateFitness(phenotypes)
  parents = select (phenotypes)
  offspring = breed(parents.genotypes)
  population = replace(parents, offspring)
  solution = bestOf(population)
  recheck(needToStop)
```
EA Selection

Principles:
• everyone has non-zero probability of being an ancestor
• individual fitness relative to population mean fitness or rank of fitness is important
• Sometimes the best of a population is always bred directly into next generation: “elitism”

Some standard methods:
• Roulette wheel
• Tournament Selection
  • n tournaments of size k

*We give the algorithm a “seed” for its RNG.
EA Tournament Selection

4 player tournament
EA Breeding

- Replication of parent [inheritance]

- Crossover - [sexual recombination]

- Mutation - [imperfect copy]

Choose crossover points and apply mutation randomly

Use a random number generator
EA Replacement

Deterministic
- use best of parents and offspring to replace parents
- replace parents with offspring

Stochastic
- some sort of tournament or fitness proportional choice
- run a tournament with old pop and offspring
- run a tournament with parents and offspring
EA Pseudocode

population.genotypes = random_pop_init()
population.phenotypes = decoder(population.genotypes)
population.fitness = calculate_fitness(population.phenotypes)

generation = 0
while needToStop == false
  generation++
  parents.genotypes = select (population.fitness)
  offspring.genotypes = crossover_mutation(parents.genotypes)
  offspring.phenotypes = decoder(offspring.genotypes)
  offspring.fitness = calculate_fitness(offspring.phenotypes)
  population = replace(parents.fitness, offspring.fitness)
  refresh(needToStop)

solution = bestOf(population)
Agenda

CONTEXT: Evolutionary computation and algorithms
- brief history
- Evolutionary Algorithms: programming the biological evolutionary process
- Genetic programming in relation to other evolutionary algorithms

1. Genetic evolution of executable expressions
   - as first introduced in 1988 by John R. Koza
   - Executable expressions
    - Operators and operands

2. Example: Block Stacking problem
   - Definition
   - Operators and operands
   - What random solutions look like Block stacking
   - Test cases and fitness function
   - Evolved solutions
Agenda

3. **Nuts and Bolts Descriptions of Algorithm Components**
   - Initialization of population of random expressions
     - Selection of operators and operands
     - Closure and sufficiency
   - Fitness of an expression
   - Genetic crossover and mutation
   - Selection
   - Preparatory Steps of GP
   - Control parameters of a GP-tree “run”

4. **Finish with more example problems**
   - Symbolic regression, (simple, with constants)

5. **How does it work?**

6. **Issues and Challenges**
   - Problems and solutions for tree overhead
   - Alternate representations for expression genome
     - Linear and graph-based genomes
GP: Evolution of executable expressions

As introduced in 1988 by John R. Koza

- Operators and operands were derived from lisp built-in functions, problem oriented high-level “language”
- Expression genomes represented as trees,
  - What are expressions and expression trees?
  - What are GP operators and operands?
Executable Expressions

- Context of an interpreter or compiler
  - 3+2
  - (+ 2 3) ; same as above, different syntax
  - 3 + square(a)
  - myFunction(arg1 arg2) - could have side effect!

- Note that expressions have a universal way of being described via a tree
  - Tree traversal order creates syntax and control flow

Common terminology and my terms today

<table>
<thead>
<tr>
<th>Components</th>
<th>What is executable</th>
<th>Genotype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operators</td>
<td>Operands</td>
<td>expression</td>
</tr>
<tr>
<td>Functions</td>
<td>Terminals</td>
<td>structure</td>
</tr>
</tbody>
</table>
Expressions as Trees

- Whether parsed preorder (node, left-child, right-child) or postorder (left-child, right-child, node) or inorder (left, node, right) the expression evaluates to the same result

- (tree)GP uses an expression tree as its genotype structure
Operators and operands in GP

GP uses operators and operands as the genetic material of its expressions

Possible Operators

- **Arithmetic**: $+$, $-$, div, mult
- **Transcendental**: log, exp,
- **Trigonometric**: cos, sine,
- **Variable assignment**
  - (setq a 10)
  - (seta 10)
- **Register read and write**
- **Index memory r/w**

- **Conditionals**
  - If <pred> <then>
  - Ifpred <then>
- **Iteration**
  - Do-until action predicate
  - Over X
  - Reverse-Over X
- **Specialized Subroutines, Procedures, or functions from the problem domain**
The Block Stacking Problem

Current State

Goal: a plan to rearrange the current state of stack and table into the goal stack

Goal Stack

Koza-92

Genetic Programming Tutorial: GECCO 2011
Block Stacking Problem: Operators and Operands

- State (updated via side-effects)
  - *currentStack*  
  - *currentTable*
- The operands
  - Each block by label
- Operators returning a block based on current stack
  - top-block
  - next-needed
  - top-correct
- Block Move Operators return boolean
  - Return nil if they do nothing, t otherwise
  - Update *currentTable* and *currentStack*  
  - to-stack(block)
  - to-table(block)
- Sequence Operator returns boolean
  - Do-until(action, test)
    » Macro, iteration timeouts
    » Returns t if test satisfied, nil if timed out
- Boolean operators
  - NOT(a), EQ(a b)
Random Block Stacking Expressions

- **eq(to-table(top-block) next-needed)**
  - Moves top block to table and returns nil

- **to-stack(top-block)**
  - Does nothing

- **eq(to-stack(next-needed))**
  - Moves next-needed block from table to stack 3 times

- **do-until(to-stack(next-needed))**
  - (not(next-needed))
  - completes existing stack correctly (but existing stack could be wrong)
Fitness Cases

• different initial stack and table configurations (Koza - 166)
  – stack is correct but not complete
  – top of stack is incorrect and stack is incomplete
  – Stack is complete with incorrect blocks

• Each correct stack at end of expression evaluation scores 1 “hit”

• fitness is number of hits (out of 166)
Evolved Solutions to Block Stacking

\[ \text{eq(do-until(to-table(top-block) \ (not top-block)) do-until(to-stack(next-needed) \ (not next-needed))} \]

- first do-until removes all blocks from stack until it is empty and top-block returns nil
- second do-until puts blocks on stacks correctly until stack is correct and next-needed returns nil
- eq is irrelevant boolean test but acts as connective
- wasteful in movements whenever stack is correct

- Add a fitness factor for number of block movements

\[ \text{do-until(eq (do-until (to-table(top-block)) (eq top-block top-correct)) (do-until (to-stack(next-needed) \ (not next-needed)) (not next-needed))} \]

- Moves top block of stack to table until stack is correct
- Moves next needed block from table to stack
- Eq is again a connective, outer do-until is harmless, no-op
Agenda Checkpoint

- Introduced to evolutionary algorithm
- GP is an EA that evolves executable expressions composed of operators and operands
  - Expressions and their parse trees
- The block stacking problem
  - Definition
  - Operators and operands
  - Fitness of a block stacking expression
  - GP-evolved solutions to block stacking problem
- Next,
  - How can we create random GP expressions?
  - How can we create a diverse population of expressions?
  - What is general procedure for fitness function design?
  - How do we mutate and crossover expressions?
  - Selection?
Population Initialization

• Fill population with random expressions
  – Create a operator set and a corresponding argument-count set
  – Create an operand set (arg-count = 0)
  – draw from operator set with replacement and recursively enumerate operator’s argument list by additional draws from operators U operands.
  – Recursion ends at draw of an operand
  – requires closure and/or typing

• maximum tree height parameter
  – At max-height-1, draw from operands only

• “ramped half-half” method ensures diversity
  – equal quantities of trees of each height
  – half of height’s trees are full
    » For full tree, only draw from operands at max-height-1
Things to Ensure to Evolve Programs

- **Sufficiency:** the operators and operands that can form executable expressions must be adequate to formulate a solution to the problem
  - I have my students hand code some solution (though not necessarily correct)
  - Operands are usually problem’s decision variables
  - Operators must be wisely chosen but not too complex
    » primitives like arithmetic, boolean, condition, iteration, assignment
    » Problem specific (eg next-needed)

- **Closure:** all functions must be coded so that they can accept parameters of any type
  - In block stacking, we can handle boolean or block

- Programs of varying length and structure must compose the search space

- Crossover of the genotype must preserve syntactic correctness so the program can be directly executed
Determining a Expression’s Fitness

• **One test case:**
  – Execute the expression with the problem decision variables (i.e., operands) bound to some test value and with side effect values initialized
  – Designate the “result” of the expression

• **Measure the error between the correct output values for the inputs and the final outputs of the expression**
  – Final output may be side effect variables, or return value of expression
  – Eg. Examine currentStack vs goalstack for block stacking
  – Eg. the heuristic in a compilation, run the binary with different inputs and measure how fast they ran.
  – EG, Configure a circuit from the genome, test the circuit with an input signal and measure response vs desired response

• **Usually have more than one test case but cannot enumerate them all**
  – Use rational design to create incrementally more difficult test cases (e.g., block stacking)
  – Use balanced data for regression
Genotype Representation: Tree

- Required: Crossover of the genotype must preserve syntactic correctness so the program can be directly executed.
- Given: expression are created from provided “language” of operators and operands (aka primitives, functions & terminals).
- Solution: Genetically manipulate program in an expression tree representation.
- Convenient for LISP or Scheme where this is expression tree.
- (if (and (< t1 t2) (= t3 t4) 0 5)
Tree Crossover

Parent 1

if
    and
        <
            av
                t1
            t2
        =
            t3
            max

Child 1

if
    and
        <
            av
                t1
            t2
        >
            sum
                t1
            sum
                t5

Parent 2

if
    not
        >
            sum
                t1
            sum
                t5

Child 2

if
    not
        =
            t3
            max
                t4
Tree Crossover Details

- Crossover point in each parent is picked at random
- Conventional practices
  - All nodes with equal probability
  - Leaf nodes chosen with 0.1 probability and non-leaf with 0.9 probability
- Probability of crossover
  - Typically 0.9
- Maximum depth of child is a run parameter
  - Typically ~ 15
  - Can be size instead

- Two identical parents rarely produce offspring that are identical to them
- Tree-crossover produces great variations in offspring with respect to parents
- Crossover, in addition to preserving syntax, allows expressions to vary in length and structure (sub-expression nesting)
Tree Mutation

- Often only crossover is used
- But crossover behaves often like macro-mutation
- Mutation can be better tuned to control the size of the change
- A few different versions
HVL-Mutation: substitution, deletion, insertion

Parent

Mutant-subst

Mutant-deletion

Mutant-addition
Other sorts of mutation

• Koza:
  – Randomly remove a sub-tree and replace it
  – Permute: mix up order of args to operator
  – Edit: + 1 3 -> 4, and(t t) -> t
  – Encapsulate: name a sub-tree, make it one node and allow re-use by others (protection from crossover)
    » Developed into advanced GP concept known as
      ✷ Automatic module definition
      ✷ Automatically defined functions (ADFs)

• Make your own
  – Could even be problem dependent (what does a subtree do? Change according to its behavior)
Selection in Genetic Programming

• Proceeds in same manner as evolutionary algorithm
  – Same set of methods
  – Conventionally use tournament selection
  – Also see fitness proportional selection
  – Cartesian genetic programming:
    » One parent: generate 5 children by mutation
    » Keep best of parents and children and repeat
      ✷ If parent fitness = child fitness, keep child
Top Level GP Algorithm

Begin

pop = random programs from a set of operators and operands
repeat

execute each program in pop with each set of inputs
measure each program’s fitness
repeat

select 2 parents

copy 2 offspring from parents

crossover

mutable

to new-pop

until pop-size

pop = new-pop

until max-generation
or
adequate program found

End

Grow or Full
Ramped-half-half
Max-init-tree-height

Tournament selection
Fitness proportional selection
Your favorite selection
Tournament size

HVL-mutate
Subtree subst
Permute
Edit
Your own

Mutation probs

Prob to crossover
Max-tree-height

Prepare input data
Designate solution
Define error between actual and expected

Sub-tree crossover

Max-init-tree-height
GP Preparatory Steps

1. Decide upon operators and operands
2. Set up the fitness function
3. Decide upon run parameters
4. Determine settings for the parameters
GP Parameters

- Population size
- Number of generations
- Max-height of trees on random initialization
  - Typically 6
- Probability of crossover
  - Higher than mutation
  - 0.9
  - Rest of offspring are copied
- Probability of mutation
  - Probabilities of addition, deletion and insertion
- Population initialization method
  - Ramped-half-half
  - All full
  - All non-full
- Selection method
  - Elitism?
- Termination criteria
- Fitness function
- what is used as “solution” of expression
Agenda Checkpoint

- Introduced to evolutionary algorithm
- GP is an EA that evolves executable expressions composed of operators and operands
  - Expressions and their parse trees
- The block stacking problem
  - Definition
  - Operators and operands
  - Fitness of a block stacking expression
  - GP-evolved solutions to block stacking problem
- How we create random GP expressions
- How we create a diverse population of expressions
- A general procedure for fitness function design
- How we mutate and crossover expressions
- Selection
- Next, MORE EXAMPLES
Simple Symbolic Regression

• Given a set of independent decision variables and corresponding values for a dependent variable

• Want: a model that predicts the dependent variable
  – Eg: linear model with numerical coefficients
    » Y = aX1 + bX2 + c(X1X2)
  – Eg: non-linear model
    » y = a x1^2 + bx2^3
  – Prediction accuracy: minimum error between model prediction and actual samples

• Usually: designer provides a model and a regression (ordinary least squares, Fourier series) determines coefficients

• With genetic programming, the model (structure) and the coefficients can be learned

• Example: y=f(x)

• Domain of x [-1.0,+1.0]

• Choose the operands
  – X

• Choose the operators
  – +, -, *, / (protected)
  – Maybe also sin, cos, exp, log (protected)

• Fitness function: sum of absolute error between yi, and expression’s return values

• Prepare 20 points for test cases

• Test problem:
  – Y=x4 + x3 + x2 + x
  – GP can create coefficient (x/x div x+x = 1/2) but…
Symbolic Regression with Numeric Coefficients: Ephemeral Random Constants

- New Test problem:
  \[ Y = 3x^4 + 10x^3 + 2x^2 + 3x \]

- requires constant creation
- Ephemeral random constants provide GP with numerical solution components
- Provide ERC set
  \[ R = \{-10,-9,-8,...0...8,9,10\} \]

- Include \( R \) among the operands. When individual is to be randomly created and \( R \) is drawn, one of the elements in \( R \) becomes the new operand.

- GP only has the constants that are randomly drawn in the initial population
- Constants could be lost through the selection process (no expression with a constant survives reproduction)
- But, GP has more primitive material to work with
- It works... partially
- Issue with size of constants, coordination of model and coefficient search, as a “clump” of numbers grows, it is more vulnerable to crossover disruption
More Examples of Genetic Programming

• Evolve priority functions that allow a compiler to heuristically choose between alternatives in hyper-block allocation

• Evolve a model that predicts, based on past market values, whether a stock’s value will increase, decrease or stay the same
  – Measure-correlate-predict a wind resource
  – ICU clinical forecasting
  » FlexGP

• Flavor design
  – Model each panelist
    » Advanced methods for panelist clustering, bootstrapped flavor optimization

• Community Benchmarks
  – Artificial Ant
  – Boolean Multiplexor

• FlexGP
  – Cloud scale, flexibly factored and scaled GP
How Does it Manage to Work

- Exploitation and exploration
  - Selection
  - Crossover
- Selection
  - In the valley of the blind, the one-eyed man is king
- Crossover: combining
- Koza’s description
  - Identification of sub-trees as sub-solutions
  - Crossover unites sub-solutions
- For simpler problems it does work

- Current theory and empirical research have revealed more complicated dynamics
Why are we still here?
Issues and Challenges

• Trees use up a lot of memory
• Trees take a long time to execute
  – Change the language for expressions
    » C, Java
  – Pre-compile the expressions, PDGP (Poli)
  – Store one big tree and mark each pop member as part of it
    » Compute subtrees for different inputs, store and reuse
• Bloat: Solutions are full of sub-expressions that may never execute or that execute and make no difference
• Operator and operand sets are so large, population is so big, takes too long to run
• Runs “converge” to a non-changing best fitness
  – No progress in solution improvement before a good enough solution is found
 Runs “converge”: Evolvability

- Is an expression tree ideal for evolvability?
- Trees do not align, not mixing likes with likes as we would do in genetic algorithm
- Biologically this is called “non-homologous”
- One-point crossover
  - By Poli & Langdon
  - Theoretically a bit more tractable
  - Not commonly used
  - Still not same kind of genetic material being swapped
Evolvability: are there building blocks?

- Does a tree or expression have building blocks?
  - Context sensitivity of sub-expressions
  - What is the “gene” or unit of genetic transmission?
  - Building blocks may come and go depending on the context in which they are found

- Where does the Good Stuff Go and Why?
  - Goldberg and O’Reilly

- The semantics of the operators influences the shape of the expressed part of the tree

- A look at two extremes:
  - (iflte x a) - ORDER
    » Context sensitive
  - (+ a b) - MAJORITY
    » Aggregation

- Even with this simplification, predicting the dynamics is difficult

- Will an imperative expression language offer better building blocks?

- Will a linear genome provide less complicated genome dynamics?
Evolvability - modularity and reuse

- Expression tree must be big to express reuse and modularity
- Is there a better way to design the genome to allow modularity to more easily evolve?

The representation of $(x - 1)^2 + (x - 1)^3$ in a tree-based genome
Evolvability: modularity and reuse

(1) \( x = x - 1 \)
(2) \( y = x \times x \)
(3) \( x = x \times y \)
(4) \( y = x + y \)

The dataflow graph of the \((x - 1)^2 + (x - 1)^3\) polynomial
Register Machine Genotype

- linear genotype, varying length, direct data

```plaintext
CPU Registers

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>88</td>
<td>122</td>
<td>56</td>
</tr>
</tbody>
</table>

Genotype

b = b+c
a = a xor c
c = b*c
c = c-a

Crossover

P1

b=...
a=...
c=...
c=...

P2

b=...
a=...
c=...

C1

b=...
a=...
c=...

C2

b=...
a=...
c=...
```

Genetic Programming Tutorial: GECCO 2011
Register Machine Advantages

- Easier on memory and crossover handling
- Supports aligned “homologous” crossover
- Registers can act as poor-man’s modules
- The primitive level of expressions allows for
  - Potentially more easily identifiable building blocks
  - Potentially less context dependent building blocks
- The register level instructions can be further represented as machine instructions (bits) and run native on the processor
  - AIM-GP (Auto Induction of Machine Code GP)
  - Intel or PPC or PIC, java byte code,
  - Experience with RISC or CISC architectures
  - Patent number: 5946673, DISCIPLUS system
Cartesian Genetic Programming

- Julian Miller
- Operators and operands are nodes and data flow is described by genome
- Fixed length genome but variable length phenome
  - Integers in blocks
  - For each block, integers to name inputs and operator
- Unexpressed genetic material can be turned on later
- No bloat observed (plus nodes are upper bounded)
Dealing with Bloat

Examples:
- (not (not x))
- (+ x 0)
- (* x 1)
- (Move left move-right)
- If (2=1) action

No difference to fitness (defn by Banzhaf, Nordin and Keller)
Can be local or global

Why does it occur?
- Crossover is destructive
- Effective fitness is selected for

Effective fitness
- Not just my fitness but the fitness of my offspring

Approaches
- Avoid - change genome structure
- Remove: Koza’s edit operation
- Pareto GP
- Penalize: parsimony pressure
  » Fitness = A(perf) + (1-a)(complexity

Examples:
- (not (not x))
- (+ x 0)
- (* x 1)
- (Move left move-right)
- If (2=1) action

No difference to fitness (defn by Banzhaf, Nordin and Keller)
Can be local or global
Reference Material

Where to Find It Online

- **ACM digital library**: http://portal.acm.org/
  - GECCO conferences,
  - GP conferences,
  - Evolutionary Computation Journal (MIT Press)
- **IEEE digital library**: http://www.computer.org/portal/web/csdl/home
  - Congress on Evolutionary Computation (CEC)
  - IEEE Transactions on Evolutionary Computation
- **Springer digital library**: http://www.springerlink.com/
  - European Conference on Genetic Programming: “EuroGP”
- **Genetic Programming Bibliography**
  - http://www.cs.bham.ac.uk/~wbl/biblio/
GP Software Libraries

- ECJ - Java, GMU, Sean Luke
- GPLab - matlab, Sara Silva
- Datamodeler - mathematica, Evolved Analytics
- Tiny GP - C/C++, Poli
- DR-EA-M
- BEAGLE - Christian Gagne
- Benchmarks Wiki
  - EvoDesignOpt Group homepage
- Eureqa - Schmidt, Lipson, Cornell U.
Reference Material - Books

• Advances in Genetic Programming
  – 3 years, each in different volume, edited
• Genetic Programming: From Theory to Practice
  – 10 years of workshop proceedings, on SpringerLink, edited
• John R. Koza
  – Genetic Programming II: Automatic Discovery of Reusable Programs, 1994 (MIT Press)
  – Genetic Programming III: Darwinian Invention and Problem Solving, 1999 with Forrest H Bennett III, David Andre, and Martin A. Keane, (Morgan Kaufmann)
• Genetic Programming: An Introduction, Banzhaf, Nordin, Keller, Francone, 1997 (Morgan Kaufmann)
• Linear genetic programming, Markus Brameier, Wolfgang Banzhaf, Springer (2007)
• A Field Guide to Genetic Programming, Poli, Langdon, McPhee, 2008, Lulu and online digitally
• Essentials of Metaheuristics, Sean Luke, 2010
Specific References in Tutorial

Classic Books

Academic Papers
The End