Genetic Programming

A Tutorial Introduction

Una-May O' Reilly
Evolutionary Design & Optimization Group
unamay@csail.mit.edu

CSAIL

Massachusetts Institute of Technology

GECCO' 12, Companion July 7-11, 2012, Philadelphia, PA, USA.
ACM 978-1-4503-1178-6/12/07.

GECCO

Instructor

- Leader: Evolutionary Design and Optimization group, MIT CSAIL
- Focus on solving real world, complex problems requiring machine learning where evolutionary computation is the core component for cloud scale 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
  - ICU clinical data mining

EvoDesignOpt

1

Tutorial Goals

- Introduction to GP algorithm, given some knowledge of genetic algorithms or evolutionary strategies
- Recognize GP design properties when you hear about them
- Teach it in an undergrad lecture
- Try it "out of the box" - with software libraries of others
- Groundwork for advanced topics
  - Theory
  - Specialized workshops – Symbolic Regression, bloat, etc
  - GP Track talks at GECCO
  - Proceedings of EuroGP

2

Agenda

Context: Evolutionary Computation and Evolutionary Algorithms

1. GP is the genetic evolution of executable expressions
2. Nuts and Bolts Descriptions of Algorithm Components
3. Resources and reference material
4. Examples
5. Deeper discussion (time permitting)

Agenda

3

Agenda

Context: Evolutionary Computation and Evolutionary Algorithms

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 Domains where EAs are Used

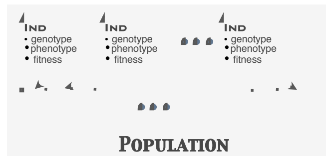
- Where there is need for complex solutions
  - evolution is a process that gives rise to complexity
  - a continually evolving, adapting process, potentially with changing environment from which emerges modularity, hierarchy, complex behavior and complex system relationships
- Combinatorial optimization
  - 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
  - Typified by discrete variables
  - Solved by Genetic Algorithm (GA)

Problem Domains where EAs are Used

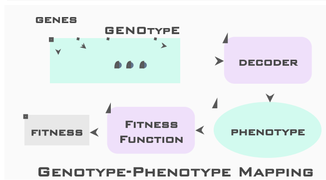
- Continuous Optimization
  - non-differentiable, discontinuous, multi-modal, large scale objective functions
  - applications: engineering, mechanical, material, physics
  - Typified by continuous variables
  - Solved by Evolutionary Strategy (ES)
- Program Search
  - system identification aka symbolic regression
    - » chemical processes, financial strategies
  - design: creative blueprints, generative designs - antennae, Genr8, chairs, lens
  - automatic programming: compiler heuristics
  - AI ODEs, invariants, knowledge discovery
  - Solved by Genetic Programming (GP)

Key EA Data Structures

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



- GENOTYPE is an array of gene(s)
- GENOTYPE is input parameter to decoder procedure that returns PHENOTYPE
- PHENOTYPE is input parameter to fitness-evaluation routine that returns a numeric variable called FITNESS



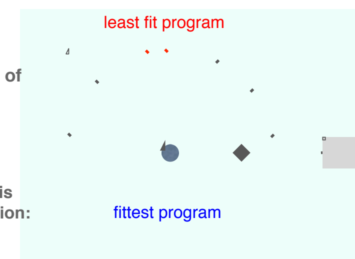
EA Generation Loop

Each generation
- select
- breed
- replace

```
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

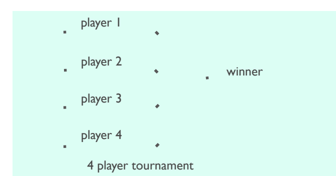
Principles:
- everyone has non-zero probability of being an ancestor
- individual fitness relative to population mean fitness or rank of fitness is important
- Sometimes the best of a population is always bred directly into next generation: "elitism"



Some standard methods:
- Roulette wheel
- Tournament Selection
    - n tournments of size k

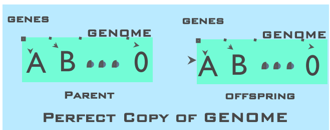
*We give the algorithm a "seed" for its RNG.

EA Tournament Selection

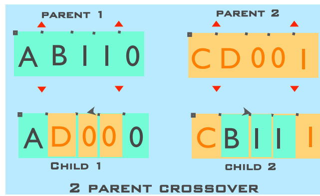


EA Breeding

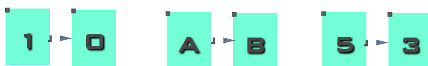
- Replication of parent [inheritance]



- 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()                                birth
population.phenotypes =decoder(population.genotypes)                    development
population.fitness= calculate_fitness(population.phenotypes)            fitness for breeding

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

solution = bestOf(population)
```


Agenda

Context: Evolutionary Computation and Evolutionary Algorithms

1. **GP is the genetic evolution of executable expressions**
2. **Nuts and Bolts Descriptions of Algorithm Components**
3. **Examples**
4. **Resources and reference material**
5. **Deeper issues (time permitting)**

Agenda – section review

Context: Evolutionary Computation and Evolutionary Algorithms

- Shown problem domains where EAs are used
- EA Data Structure: Individual
- EA Loop
  - » Evolutionary computation which is agnostic of representation
  - » Selection
  - » Replication
  - » Inheritance and Variation -> crossover and mutation
- Examples of genotypes and phenotypes

Agenda 16

Agenda

Context: Evolutionary Computation and Evolutionary Algorithms

1. GP is the genetic evolution of executable expressions

Agenda 17

EA Individual Examples

Problem	Gene	Genome	Phenotype	Fitness Function
TSP	110	sequence of cities	tour	tour length
Function optimization	3.21	variables x of function	f(x)	lmin-f(x)l
graph k-coloring	permutation element	sequence for greedy coloring	coloring	# of uncolored nodes
investment strategy	rule	agent rule set	trading strategy	portfolio change

Evolutionary Computation and Evolutionary Algorithms 18

Koza's Executable Expressions

Pioneered circa 1988

- Lisp S-Expressions
  - Composed of primitives called 'functions' and 'terminals'

Example:

- primitives: + - * div a b c d 4
- (*(- (+ 4 c) b) (div d a))

In a Lisp interpreter:

1. bind a b c and d
2. Evaluate expressions

```
% Lisp interpreter
(set! a 2) -> 2
(set! b 4) -> 4
(set! c 6) -> 6
(set! d 8) -> 8
(*(- (+ 4 c) b) (div d a)) -> 12
; Rule Example
(if (= a b) c d) -> 8
;Predicate:
(> c d) -> nil
```

GP Evolves Executable Expressions 19

More Lisp details

A Lisp GP system is a large set of functions which are interpreted by evaluating the entry function

GP expressions are first class objects in LISP so the GP functions can manipulate them as data as well as have the interpreter read and evaluate them

Functions Used in GP Expressions

Arithmetic

- +, - , div, mult
  - Division must be protected
  - Return 1 if divisor = 0
- Transcendental: log, exp,
- Trigonometric: cos, sine,

Boolean

- AND NOT OR NAND

Logical

- (IF <pred> <True> <False>)

Iteration

- (OVER <collection> <function>)

Predicate

- > < == <>
- (isBlue <arg>)

Other functions

- (addOne <arg>)
- (Max <collection>), Max(x,y)
- (Mean<collection>), Mean(x,y)

See Eureqa user guide for other examples

- http://creativemachines.cornell.edu/sites/default/files/Eureqa_User_Guide.pdf

Details When Using Executable Expressions

- Sufficiency
  - Make sure a solution can be plausibly expressed when choosing your primitive set
    - » Functions must be wisely chosen but not too complex
    - » General primitives: arithmetic, boolean, condition, iteration, assignment
    - » Problem specific primitives
  - Can you handcode a naïve solution?
  - Balance flexibility with search space size
- Closure
  - Design functions with wrappers that accept any type of argument
  - Often types will semantically clash…have a default way of dealing with this
- The value of typing
  - Strongly typed GP only evolves expressions within type rules
  - Trades off semantic structure with flexible search

Abstract Syntax Trees

Motivation: GP needs to be able to crossover and mutate executable expressions, how?

- 3+2
- (+ 2 3) ; same as above, different syntax
- (3 2 +) ; same too

- Expressions can be represented universally by an abstract syntax via a tree
  - Tree traversal is syntax and control flow

Abstract Syntax Trees

+
2 3

Inorder: 2+3

preorder: + 2 3

Post-order: 2 3 +

+
- max
2 3 a best

Inorder: (2-3) + (max a best)

preorder: (+ (-2 3) (max a best))

Post-order: (2 3 -) (a best max) +)

- 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

Agenda Review

Context: Evolutionary Computation and Evolutionary Algorithms

1. GP is the genetic evolution of <u>executable</u> expressions
   - Lisp S-expressions
   - Functions and terminals
   - Closure and sufficiency
   - abstract syntax trees

Agenda

Context: Evolutionary Computation and Evolutionary Algorithms

1. GP is the genetic evolution of <u>executable</u> expressions
2. Nuts and Bolts Descriptions of Algorithm Components

Population Initialization

- Fill population with random expressions
  - Create a function set Φ and a corresponding function-count set
  - Create an terminal set (arg-count = 0), T
  - draw from F with replacement and recursively enumerate its argument list by additional draws from Φ U T.
  - Recursion ends at draw of a terminal
  - requires closure and/or typing
- maximum tree height parameter
  - At max-height-1, draw from T 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

Determining a Expression's Fitness

- One test case:
  - Execute the expression with the problem decision variables (ie 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 expression result and expected result for regression
  - 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 (eg block stacking)
  - Use balanced data for regression
  - See Eureqa user guide for other ideas
    - » http://creativemachines.cornell.edu/sites/default/files/Eureqa_User_Guide.pdf

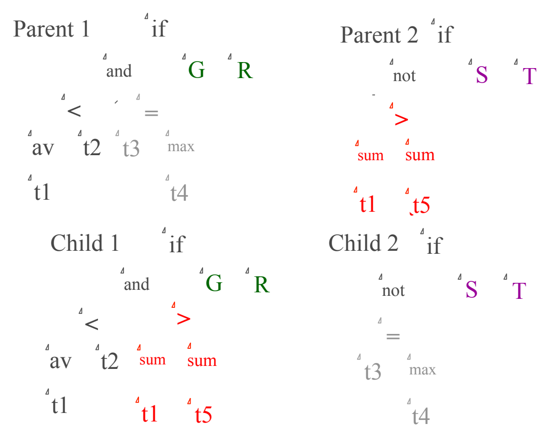
Nuts and Bolts GP Design 28

Things to Ensure to Evolve Programs

- Programs of varying length and structure must compose the search space
- Closure
- Crossover of the genotype must preserve syntactic correctness so the program can be directly executed

Nuts and Bolts GP Design 29

GP Tree Crossover



Nuts and Bolts GP Design 30

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 probility 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)

Nuts and Bolts GP Design 31

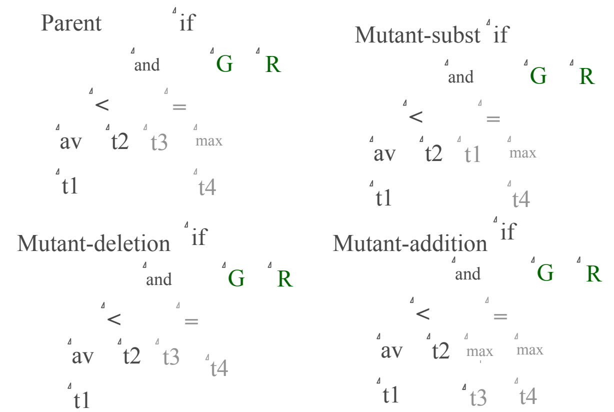
GP Tree Mutation

- 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

Nuts and Bolts GP Design

32

HVL-Mutation: substitution, deletion, insertion



Nuts and Bolts GP Design

33

Other Sorts of Tree 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)

Nuts and Bolts GP Design

34

Selection in GP

- 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

35

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
mutate
add 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

Prepare input data
Designate solution
Define error between actual and expected

•HVL-mutate
•Subtree subst
•Permute
•Edit
•Your own

Mutation probs

Sub-tree crossover

Prob to crossover

Max-tree-height

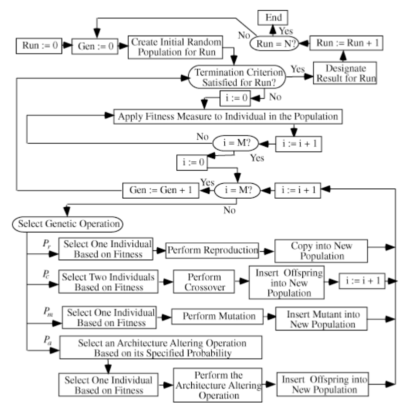
GP Preparatory Steps

1. Decide upon functions and terminals
   – Terminals bind to decision variables in problem
   – Defines the search space
2. Set up the fitness function
   – Translation of problem goal to GP goal
   – Minimization of error between desired and evolved
   – Maximization of a problem based score
3. Decide upon run parameters
   – Population size is most important
     » Budget driven or resource driven
   – GP is robust to many other parameter choices
4. Determine a halt criteria and result to be returned
   – Maximum number of fitness evaluations
   – Time
   – Minimum acceptable error
   – Good enough solution (satisficing)

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

Run Level GP Flowchart



From http://www.genetic-programming.com/gpflowchart.html

Agenda Checkpoint

Nuts and Bolts GP Design
- 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
- Put it together: one algorithm, at run level

Agenda

Context: Evolutionary Computation and Evolutionary Algorithms

1. GP is the genetic evolution of executable expressions
2. Nuts and Bolts Descriptions of Algorithm Components
3. Resources and reference material

Reference Material

Where to identify conference and journal material
- Genetic Programming Bibiliography
  - http://www.cs.bham.ac.uk/~wbl/biblio/

Online Material
- ACM digital library: http://portal.acm.org/
  - GECCO conferences,
  - GP conferences (pre GECCO),
  - 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"

GP Software

Commonly used in published research (and somewhat active):
- Java: ECJ, TinyGP,
- Matlab: GPLab, GPTips
- C/C++: MicroGP
- Python: DEAP, PyEvolve
- .Net: Aforge.NET

Others
- http://www.epochx.org/index.php
  Strongly typed GP, Grammatical evolution, etc
  Lawrence Beadle and Colin G Johnson
- http://www.tc33.org/genetic-programming/genetic-programming-software-comparison/
  - Dated Feb 15, 2011

Genetic Programming Benchmarks

Genetic programming needs better benchmarks

- James McDermott, David R. White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Ja´skowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, and Una-May O'Reilly.
- In Proceedings of GECCO 2012, Philadelphia, 2012. ACM.

- **Related benchmarks wiki**
  - http://groups.csail.mit.edu/EVO-DesignOpt/GPBenchmarks/index.php

44

Software Packages for Symbolic Regression

No Source code available

- Datamodeler - mathematica, Evolved Analytics
- Eureqa II - a software tool for detecting equations and hidden mathematical relationships in data
  - http://creativemachines.cornell.edu/eureqa
  - Plugins to Matlab, mathematica, Python
  - Convenient format for data presentation
  - Standalone or grid resource usage
  - Windows, Linux or Mac
  - http://www.nutonian.com/ for cloud version
- Discipulus™ 5 Genetic Programming Predictive Modelling

45

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: On the Programming of Computers by Means of Natural Selection, 1992 (MIT Press)
  - 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 IV: Routine Human-Competitive Machine Intelligence, 2003 with Martin A. Keane, Matthew J. Streeter, William Mydlowec, Jessen Yu, and Guido Lanza
- 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

46

Specific References in Tutorial

Classic Books

- Adaptation in Natural and Artificial Systems, John H Holland, (1992), MIT Press.
- Evolutionsstrategie, Ingo Rechenberg, (1994), Frommann-Holzboog.
- Artificial Intelligence through Simulated Evolution, L.J. Fogel, A.J. Owens, and M.J. Walsh (1966), John Wiley, NY.

Academic Papers

- On the Search Properties of Different Crossover Operators in Genetic Programming, Riccardo Poli and William B. Langdon, Genetic Programming 1998: Proceedings of the Third Annual Conference, pp. 293-301, Morgan Kaufmann, 22-25 July 1998.
- Where does the Good Stuff Go and Why? Goldberg and O'Reilly, Proceedings of the First European Workshop on Genetic Programming, LNCS, Vol. 1391, pp. 16-36, Springer-Verlag, 14-15 April 1998.
- Cartesian genetic programming, GECCO-2008 tutorials, pp. 2701-2726, ACM, 12-16 July 2008.

47

Agenda

Context: Evolutionary Computation and Evolutionary Algorithms

1. GP is the genetic evolution of executable expressions
2. Nuts and Bolts Descriptions of Algorithm Components
3. Resources and reference material
4. Examples

Agenda 48

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 coefficients (x/x div x +x = 1/2) but…

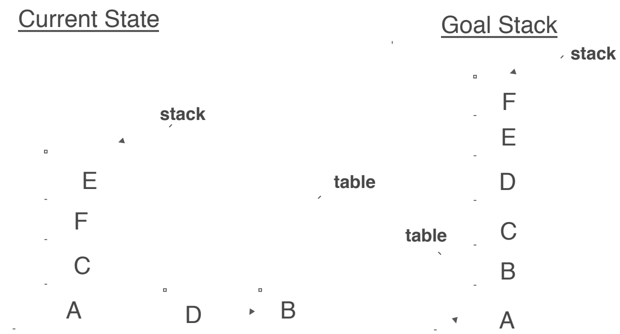
GP Examples 49

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

GP Examples 50

The Block Stacking Problem Koza-92



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

Block Stacking Example 51

Block Stacking Problem: Primitives

- 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 satisified, 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)

  eq (to-stack(next-needed) 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)

Block Stacking 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

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

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

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
  - Artifical Ant
  - Boolean Multiplexor
- FlexGP
  - Cloud scale, flexibly factored and scaled GP

GP Examples

56

Agenda

Context: Evolutionary Computation and Evolutionary Algorithms

1. GP is the genetic evolution of executable expressions
2. Nuts and Bolts Descriptions of Algorithm Components
3. Resources and reference material
4. Examples
5. Deeper discussion (time permitting)

Agenda

57

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

Time Permitting

58

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

Time Permitting

59

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

Time Permitting

60

Evolvability: are there building blocks?

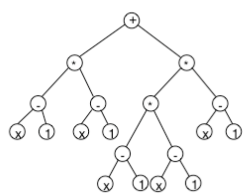
- Does a tree or expression have building blocks?
  - Context sensitivity of sub-expressions
  - What is the “gene” or unit of genetic transmission?
  - Building blocks may come and go depending on the context in which they are found
- Where does the Good Stuff Go and Why?
  - Goldberg and O’Reilly
- 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?

Time Permitting

61

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?



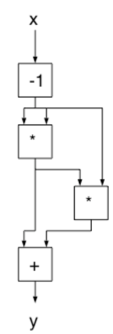
The representation of $(x-1)^2 + (x-1)^3$ in a tree–based genome

Time Permitting

62

Evolvability: modularity and reuse

```
(1)   x=x-1
(2)   y=x*x
(3)   x=x*y
(4)   y=x+y
```



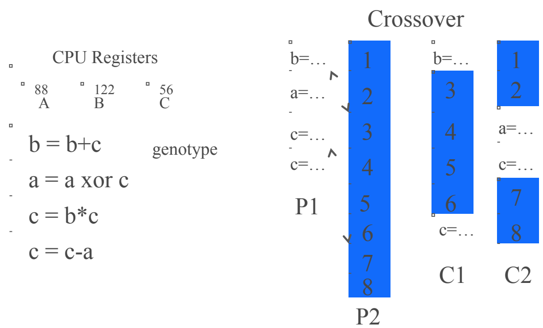
The dataflow graph of the $(x-1)^2 + (x-1)^3$ polynomial

Time Permitting

63

Register Machine Genotype

- linear genotype, varying length, direct data



Time Permitting

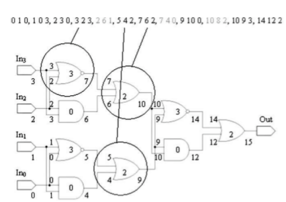
Register Machine Advantages

- 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

Time Permitting

Cartesian Genetic Programming

- Developer: 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



Time Permitting

Dealing with Bloat

- 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

Time Permitting

Agenda

Context: Evolutionary Computation and Evolutionary Algorithms

1. GP is the genetic evolution of executable expressions
2. Nuts and Bolts Descriptions of Algorithm Components
3. Resources and reference material
4. Examples
5. Deeper discussion (time permitting)

The End