Course: Natural Computing

4. Genetic Programming

J. Michael Herrmann
School of Informatics, University of Edinburgh
michael.herrmann@ed.ac.uk, +44 131 6 517177

Genetic and evolutionary algorithms

	Encoding	Operators	Tasks
GA	Binary strings	Mutation, crossover	Optimisation search
GP	Trees	As above and additional operators	Computer programs with genuine fitness
ES	Problems with real-valued parameters	Mutation, adaptive mutation rates, CMA	Optimisation search
EP	Real-valued vectors	As above and additional operators, gradients	Parametrised computer programs

Genetic Programming

- A biological genome is actually a *program* to produce an individual, rather than a direct encoding of the individual.
- Programs that realise the same function can differ in structure, efficiency, style etc.
- Syntactic rules for programs can help to restrict the complexity of the vast search space.
- Programs run on computers, i.e., fitness evaluation can be relatively cheap.
- Some types of programs can be more easily evolved than others. Success may depend also on the progamming language.

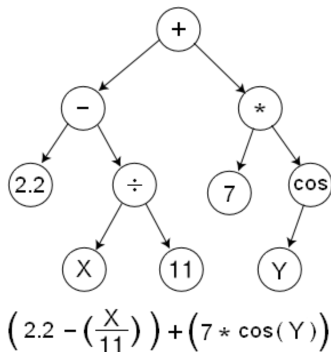
Genetic Programming

- Genetic programming now routinely delivers high-return human-competitive machine intelligence.
- Genetic programming is an automated invention machine.
- Genetic programming can automatically create a general solution to a problem in the form of a parametrised topology.
- Computer programs are the *lingua franca* for expressing the solutions to a wide variety of problems

Statements by John R. Koza et al. (2003)

Genetic Programming (GP)

- Evolutionary algorithm-based methodology inspired by biological evolution
- Finds computer programs that perform a user-defined task
- Similar to genetic algorithms (GA), but in GPs each individual is a computer program represented by a tree*
- Optimise a population of computer programs according to a fitness landscape determined by a program's ability to perform a given computational task.
- Evolution of programs is generally open-ended: The search space cannot be exhausted because GPs do not usually use fixed-length encoding.

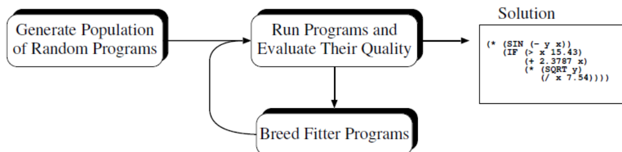


$$\left(2.2 - \left(\frac{X}{11}\right)\right) + \left(7 * \cos(Y)\right)$$

*works better in some languages than in others

Evolving Programs

- Let $P(0)$ be a population of randomly generated programs p_i
- For each p_i, run it on some input and see what it does. Rate it for fitness based on how well it does.
- Breed the fitter members of $P(0)$ to produce $P(1)$
- If happy with the behaviour of the best program produced then stop.
- . . . but how?



See: Riccardo Poli, William B. Langdon, Nicholas F. McPhee (2008) A Field Guide to Genetic Programming (http://www.gp-field-guide.org.uk)

How?

- What language should the candidate programs be expressed in? C, Python, Java, Pascal, Perl, Lisp, Machine code?
- How can you generate an initial population?
- How can you run programs safely? Consider errors, infinite loops, etc.?
- How can you rate a program for fitness?
- Given two selected programs, how can they be bred to create offspring?
- What about subroutines, procedures, data types, encapsulation, etc.?
- What about small, efficient programs?

Koza: Evolving LISP programs

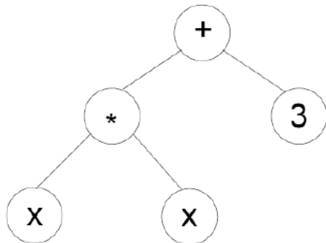
Lisp as a functional language

$f(x, y)$ is written as $(f\ x\ y)$

$10 - (3 + 4)$ is written as $(-\ 10\ (+\ 3\ 4))$

Lisp programs can be represented as trees

$f(x) = x^2 + 3$



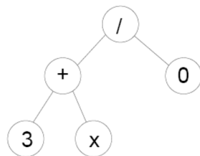
$f(x) = (+\ (*\ x\ x)\ 3)$

Here, $+$ and $*$ are function symbols (non-terminals) of arity, x and 3 are terminals. Given a random bag of terminals and non-terminals, we can make programs.

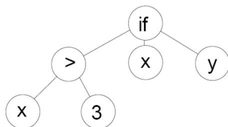
(Peter Seibel: Practical Common Lisp, 2004)

Random Programs and Closure

If we generate a random program:
How can we avoid an error?



Another random program
How can we evaluate this?



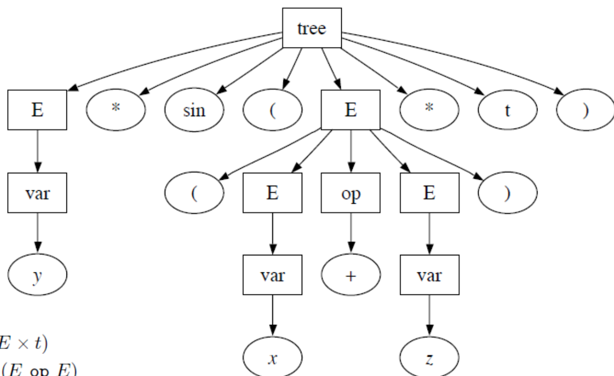
All function calls need to return a result

Closure: E.g. redefine division by zero to return FLT_MAX or zero; Overload operators to deal with variable numbers of arguments

Sufficiency: Set of nonterminals need to be sufficiently large, terminals need to be defined (if not given by data)

Grammar-Based Constraints

Typing or grammar-based approaches help to achieve closure



$$
\begin{aligned}
tree &::= E \times \sin(E \times t) \\
E &::= \texttt{var} \quad | \quad (E \ \texttt{op} \ E) \\
\texttt{op} &::= + \quad | \quad - \quad | \quad \times \quad | \quad \div \\
\texttt{var} &::= x \quad | \quad y \quad | \quad z
\end{aligned}
$$

R. Poli et al. A Field Guide to Genetic Programming (2008)

How do we rate a program for fitness?

Answer: Run it on some "typical" input data for which we know what the output should be (*Fitness cases*). The hope is the evolved program will work for all other cases (use **crossvalidation**!!).

Example: Symbolic regression on planetary orbits (Kepler's law). Given a set of values of independent and dependent variables, come up with a function that gives the values of the dependent variables in terms of the values of the independent variables.

Planet	A	P
Mercury	0.39	0.24
Venus	0.72	0.61
Earth	1.00	1.00
Mars	1.52	1.84
Jupiter	5.20	11.9
Saturn	9.53	29.4
Uranus	19.1	83.5
Neptune	30.1	165

Kepler's third law: Square of the period P of the planet proportional to cube of semimajor axis A, i.e. $P = A^{3/2}$.

Fitness Function

Given a number of example pairs (x_j, y_j) of data vectors x_j and desired outputs y_j, we could use the deviations of the generated program for an evaluation:

$$E_{\text{raw}} = \sum_j \| GP(x_j) - y_j \|^2$$

For a fitness function we could adjust

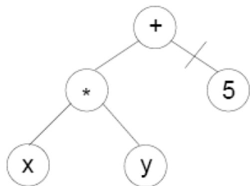
$$F_{\text{adj}} = \frac{1}{1 + E_{\text{raw}}}$$

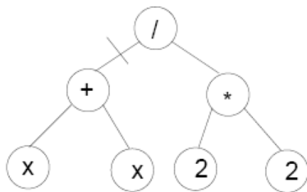
and normalise (or rank)

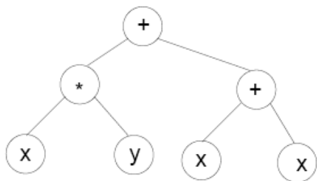
$$F_{\text{norm}}(i) = \frac{F_{\text{adj}}(i)}{\sum_k F_{\text{adj}}(k)}$$

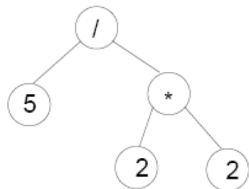
... so most fit indiviudal has fitness ≤ 1, least fit one has a fitness ≥ 0 (i, k denote individuals, and j counts the fitness cases)

Crossover









How can we cross programs?

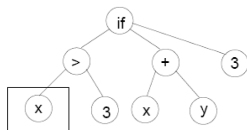
Subtree crossover

Koza's orginal (1988-92) GP system used only crossover, to try to demonstrate that GP is "more than a mutation".

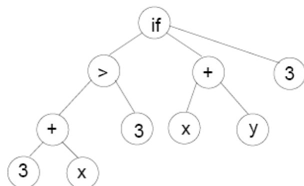
Mutation

How can we mutate a program?
Lots of other forms of mutation are possible, e.g. hoist, shrink

- shrink: replace a subtree by one of its terminals
- hoist: use only a subtree as a mutant



subtree or “grow”-mutation



or: vary numbers, exchange symbols, exchange subtrees, . . .

GP Algorithm

1. Choose a set of functions and terminals for the program you want to evolve:
   - non-terminals e.g.: if, /,* , +, −, sqrt, <, >...
   - terminals e.g.: x, y , −10, −9, . . . , 9, 10
2. Generate an initial random population of trees of maximum depth d
3. Calculate the fitness of each program in the population using the chosen fitness cases.
4. Apply selection, subtree crossover (and subtree mutation) to form a new population.

Example parameter values:

Population size = 10000
Crossover rate = 0.9

Selection: Fitness proportional

Genetic programming: Examples

Natural Computing (week 4)

J. Michael Herrmann
School of Informatics, University of Edinburgh
michael.herrmann@ed.ac.uk, +44 131 6 517177

Genetic Programming: General Points

- **Sufficiency** of the representation: Appropriate choice of non-terminals
- **Variables:** Terminals (variables) implied by the problem
- **Closure**: Typed algorithms, grammar based encoding
- **Program structure:** Terminals also for auxiliary variables or pointers to (automatically defined) functions
- Expect **multiple runs** (each with a population of solutions)
- **Local search:** Terminals (numbers) can often be found by hill-climbing
- **Fitness:** From fitness cases using crossvalidation (e.g. for symbolic regression)
- Tree-related **operators**: Shrink, hoist, grow (in addition to standard mutation and crossover)

Genetic programming: Control parameters

- Representation and fitness function
- Population size (thousands or millions of individuals)
- Probabilities of applying genetic operators
  - reproduction (unmodified cloning) 0.08
  - crossover: 0.9
  - mutation (various forms): 0.01
  - architecture altering operations 0.01
- Limits on the size, depth, run time of the programs

Finding good parameters can be difficult in GP: Try to define parameters relative to population size, program size and depth.

Example 1: GP for Symbolic Regression

Data:

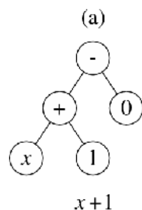
x	y
-1	1.00
-0.8	0.84
-0.6	0.76
-0.4	0.76
-0.2	0.84
0.0	1.00
0.2	1.24
0.4	1.56
0.6	1.96
0.8	2.44
1	3.00

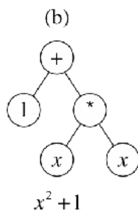
Design:

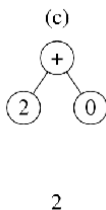
	goal	Find a program that produces y when given x, while fitting a data set
1	terminals	x, c_i (random constants)
2	functions	$+$, $-$, $*$, $/$
3	fitness	sum of absolute errors over test data set
4	parameters	Populations size, p_c, p_m, ...
5	termination	error smaller than θ

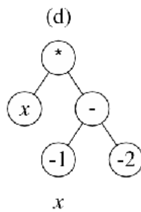
From: J. R. Koza: GA and GP (tutorial)

Example 1: GP for Symbolic Regression







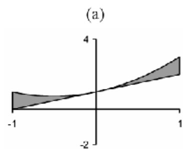


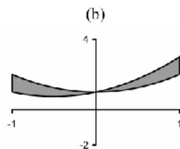
Population of 4 randomly generated individuals for generation 0

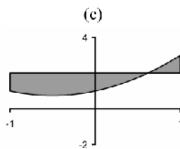
Example 1: GP for Symbolic Regression

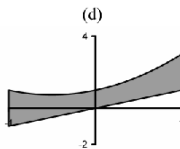
Goal function: $f(x) = x^2 + x + 1$

Performance of the individual programs





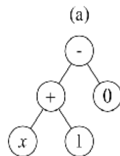




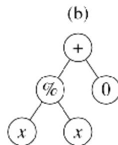
$x+1$	x^2+1	2	x
0.67	1.00	1.70	2.67

The algorithm uses only the fitness calculated from the given data.

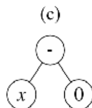
Example: GP for Symbolic Regression



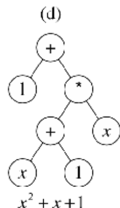
Copy of (a)



Mutant of (c) picking "2" as mutation point

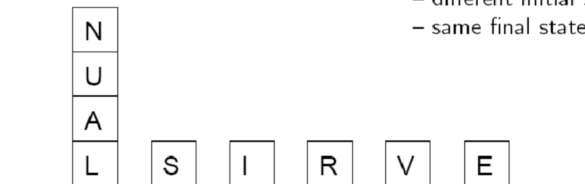


First offspring of crossover of (a) and (b) picking "+" of parent (a) and left-most "x" of parent (b) as crossover points



Second offspring of crossover of (a) and (b) picking "+" of parent (a) and left-most "x" of parent (b) as crossover points

Example 2: Learning to Plan using GP



Goal state: a single stack that spells out the word "UNIVERSAL"

Aim:

To find a program to transform any initial state into "UNIVERSAL"

Example 2: Learning to Plan

Terminals:

- CS – returns the current stack's top block
- TB – returns the highest correct block in the stack (or NIL)
- NN – next needed block, i.e. the one above TB in the goal

Functions:

- MS(x) – move block x from table to the current stack. Return T if does something, else NIL.
- MT(x) – move x to the table
- DU(exp1, exp2) – do exp1 until exp2 becomes TRUE
- NOT(exp1) – logical not (or exp1 is not executable)
- EQ(exp1, exp2) – test for equality

Learning to Plan: Results

Population size N =500 (best individual is shown below)

Generation 0: (EQ (MT CS) NN), 0/166 fitness cases correct

Generation 5: (DU (MS NN)(NOT NN)), 10/166 fitness cases correct

Generation 10: (EQ (DU (MT CS)(NOT CS))(DU (MS NN)(NOT NN)))
166/166 fitness cases correct

Koza shows how to amend the fitness function for efficient, small programs: Combined fitness measure rewards

- Correctness (number of solved fitness cases)
- AND efficiency (moving as few blocks as possible)
- AND small number of tree nodes (parsimony: number of symbols in the string)

Implementation

Many options, choose, e.g.,

- a language where trees are native data structure (Lisp)
- a language where trees are available as a library (Python, ...)
- a language where strings can be compiled (Python, ...)
- a language where functions can be variables (Javascript, ...)
- a language that conveniently supports grammars (Prolog)
- a stack and case statements (C++, ...)
- an addressable function: e.g. $f(x, 1) = \sin x, \;\; f(x, 2) = e^x$ etc. (GA?)
- a template, e.g. $a_0 + a_1 x + a_2 x^2 + \ldots$ with some $a_i = 0$ (PSO?)

Initialisation

- The initial population might be lost quickly, but general features may determine the solutions
- Assert that the functions and terminals are sufficient
- Structural properties of the expected solution (uniformity, symmetry, depth, ...)
- In practice: Start at root and choose $k = 0, ..., K$ with probability $p(k)$, choose a non-terminal with $k > 0$ arguments or a terminal for $k = 0$. If $k > 0$ repeat until no non-terminals are left or if maximal depth is reached (then $k = 0$)
- Initial trees can be chosen to be irregular (*grow method*, similar to depth-first) or balanced (*full method*, similar to breadth-first) or mixed.
- Lagrange initialisation: Crossover can be shown to produce programs with a typical distribution (Lagrange distribution of the second kind) which can be used also for initialisation
- Seeding: Start with many copies of good candidates

R. Poli et al. A Field Guide to Genetic Programming (2008)

Initialisation by seeding

- Assume one or more good programs exists for a certain problem
- Evolve solutions that are not necessarily better but different (Fitness is based on at-least-equal functionality and difference)
- Usually a good application case for GP (what about IP?)

Question: Why not running a GP seeded with all known MHO algorithms as an initial population?

Evolving Structures (Koza et al., 1997)

Example: Design of electronic circuits by composing

- Non-terminals: e.g. frequency multiplier, integrator, rectifier, resistors, wiring ...
- Terminals: input and output, pulse waves, noise generator
- Structure usually not tree-like: Meaningful substructures ("boxes" or subtrees) for crossover and structural mutations
- Fitness by desired input-output relation (e.g. by wide-band frequency response)

Genetic programming: Troubleshooting

- Study your populations: Analyse means and variances of fitness, depth, size, code used, run time, ... and correlations among these
- Runs can be very long: Checkpoint results (e.g. mean fitness)
- Control bloat in order to obtain small efficient programs: Size limitations prevent unreasonable growth of programs e.g. by soft thresholds
- Control parameters during run-time
- Small changes can have big effects
- Big changes can have no effect
- Encourage diversity and save good candidates,
- Embrace approximation: No program is error-free

How to deal with numerical constants?

Finding a set of number can be time consuming, and for a slightly different program other constants may be needed

- Hill-climbing as local search
- Hybridise with PSO or other search algorithm
- "Local gradient search of numeric leaf values"

  (Topchy et al., 2001)
- Izzo et al. (2017) Differentiable genetic programming.

  [CGP, see below]

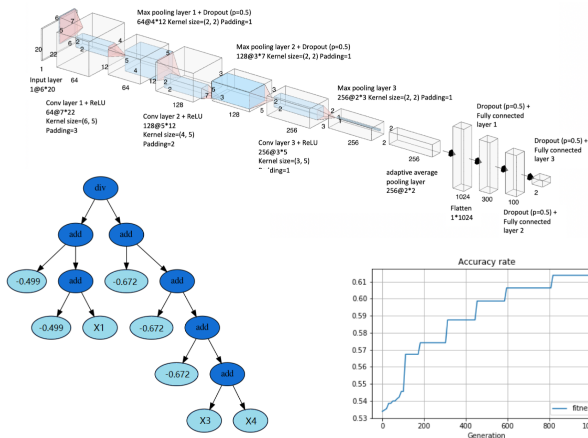
GP: Application Areas

- Problem areas involving many variables that are interrelated in a non-linear or unknown way (predicting electricity demand)
- A good approximate solution is satisfactory
  - design, control (e.g. in simulations), classification and pattern recognition, data mining, system identification and forecasting
- Discovery of the size and shape of the solution is a major part of the problem
- Areas where humans find it difficult to write programs
  - parallel computers, cellular automata, multi-agent strategies, distributed AI, FPGAs
- "Black art" problems
  - synthesis of topology and sizing of analogue circuits, synthesis of topology and tuning of controllers, quantum computing circuits
- Areas where you simply have no idea how to program a solution, but where the objective (fitness measure) is clear (e.g. generation of financial trading rules)
- Areas where large computerised databases are accumulating and computerised techniques are needed to analyse the data

Genetic programming: practical examples

- Grading lettuce (Evis Technologies GmbH, Vienna): Based on data set from human ratings, identify features some combination of which gives the best overall agreement of the rating. Resulting program is much faster, more consistent and more accurate than human performance
- Generation of financial trading rules
- Designing neural network architectures
- Evolution of electronic circuits
- Security checks in transport
- Smart homes

Example: Financial time series prediction (DJIA)



GP best individual (DJIA) representing: $f^* = \frac{X_1 - 1}{(X_3 - 1) + (X_4 - 1)}$

Performance of simple rules and of DNN are indistinguishable for few data, short training times on complex problems

(courtesy of Junyi Wang)

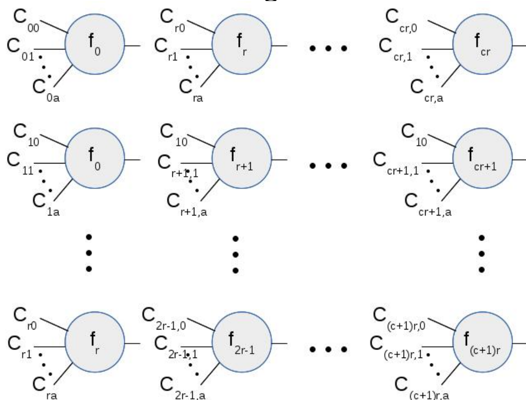
Cartesian Genetic Programming*

Natural Computing (week 4)

J. Michael Herrmann
School of Informatics, University of Edinburgh
michael.herrmann@ed.ac.uk, +44 131 6 517177

Cartesian Genetic Programming (Julian Miller 1999)

- Starting from the evolution of digital circuits
- representations as two-dimensional grids of program primitives
- flexible: variable-length network structures, skip connections.

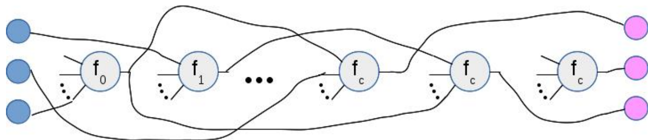


r rows
c columns
n inputs
m outputs

evolvable function, evolvable topology, no lateral connections

CGP

- Often just $r = 1$ for an arbitrary directed graph with a maximum depth



- Encoding: functions and the respective inputs, same for output units
- Functions e.g. "*", "+", "-", "/"   or as implied by problem, but also ADFs (modular) in variants of the approach
- Similar to a neural network, but without learning rule

Wilson et al. (2018) Evolving simple programs for playing Atari games

CGP

- Redundancy: many nodes do not affect output (95% redundancy is good for evolution, i.e. neutral evolution is encouraged)
- Therefore: Phenotypes are much simpler than genotypes
- Often no cross-over, one mutation per individual
- (1+4) evolution strategy: parent copied + 4 children
- Choose best out of 5
- In case of same fitness, the parent is **not** selected
- Starting from the evolution of digital circuits
- Variants: Modular, Typed, Self-modifying, Recurrent, Encoding of NN ...

Playing Atari games

- Functions include arithmetic, trigonometric, statistical, logical and list processing functions
- Cartesian GP
- Largely linear graphs with skip connections
- Functions are overloaded to accept vector or scalar inputs
- Outputs: actions (left, right, up down, shoot, ...)
- Evolved programs are very simple (e.g. 10 nodes), so results are human-readable
- Results are impressive, even better than DeepRL (Mnih et al. 2013), but the observable play behaviours are ... pragmatic.

Wilson et al. (2018) Evolving simple programs for playing Atari games

Conclusions on GP

- In order to be successful, GP algorithms need well-structured problems and sufficient computing power
- GPs have proven very successful in many applications, see the lists of success stories in Poli's talk, in Koza's tutorial, and work on CGP.
- GP provides an interesting view on the art of programming
- We will return to GP later when we talk about current developments