Natural Computing

Lecture 9

Michael Herrmann
mherrman@inf.ed.ac.uk
phone: 0131 6 517177
Informatics Forum 1.42

18/10/2011

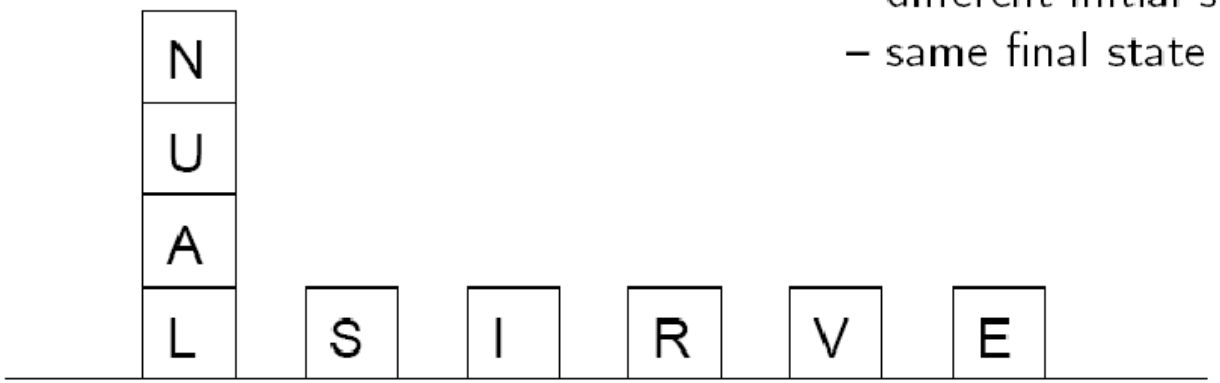
Genetic Programming: Examples and Theory:

see: http://www.genetic-programming.org, http://www.geneticprogramming.us

Example 1: Learning to Plan using GP

A planning problem (Koza):

Initial state:



Koza's data set:

166 fitness cases
– different initial states
– same final state

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

Aim:

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

Genetic Programming: 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

Generation 0: (EQ (MT CS) NN) 0 fitness cases

Generation 5: (DU (MS NN) (NOT NN)) 10 fitness cases

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

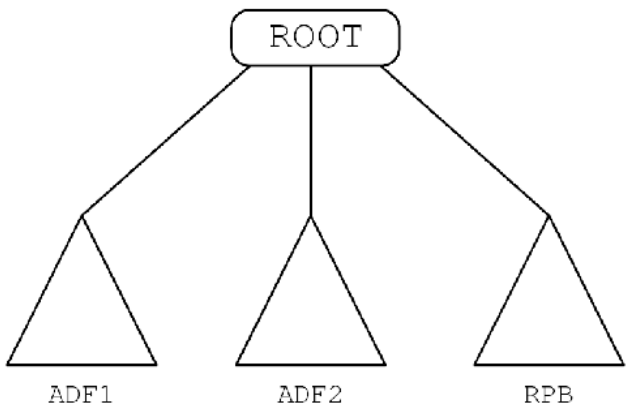
population size 500

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

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

Automatically Defined Functions

- "Efficient code": Loops, subroutines, functions, classes, or ... variables
- Automatically defined iterations (ADIs), automatically defined loops (ADLs) and automatically defined recursions (ADRs) provide means to re-use code. (Koza)
- Automatically defined stores (ADSs) provide means to re-use the result of executing code.
- Solution: function- defining branches (i.e., ADFs) and result-producing branches (the RPB)



- e.g. RPB: ADF(ADF(ADF(x))), where ADF: arg0×arg0

Example 2: The Santa Fe Trail

Objective: To evolve a program which eats all the food on a trail without searching too much when there are gaps in the trail. Sensor can see the next cell in the direction it is facing

Terminals: move, (turn) left, (turn) right

Functions: if-food-ahead, progn2, progn3 (unconditional connectives: evaluate 2 or 3 arguments in the given order)

Program with high fitness:

```
(if-food-ahead move
    (progn3
        left
        (progn2 (if-food-ahead move right)
            (progn2 right (progn2 left right)))
        (progn2 (if-food-ahead move left) move)
    )
)
```

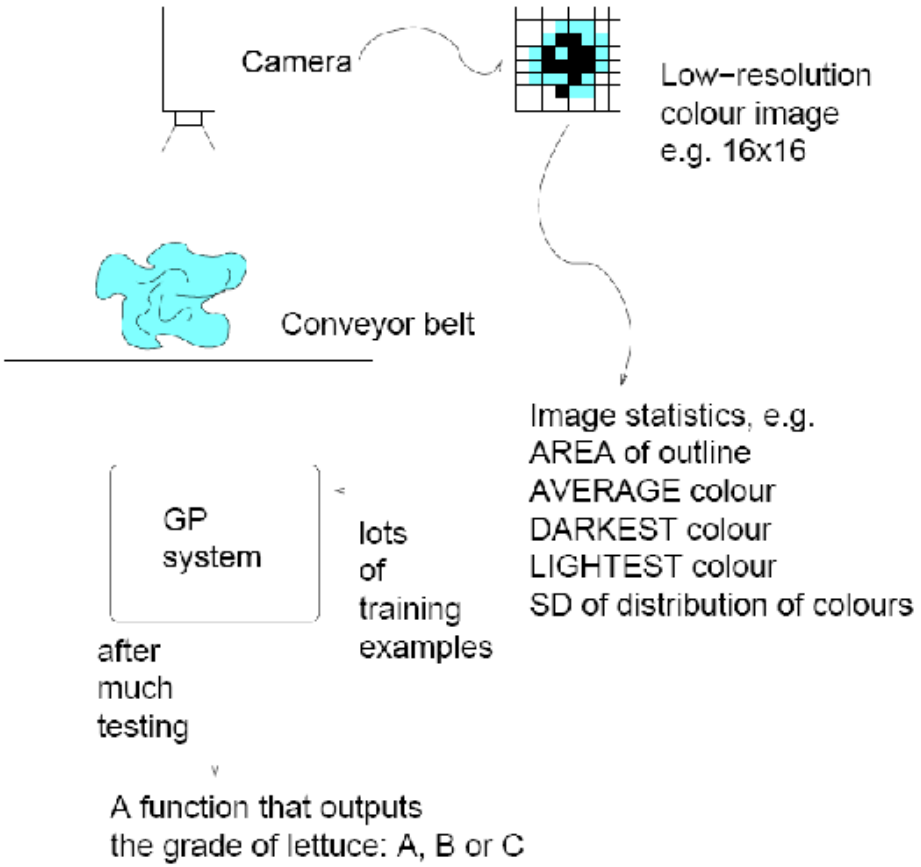
Fitness: E.g. amount of food collected in 400 time steps

Genetic programming: A practical example

Photo: Selena von Eichendorf

Grading lettuces . . . uses proprietary form of GP, by Evis Technologies GmbH, Vienna.

Much faster and more accurate than humans.



Evolving Structures

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)

Initialisation

- The initial population might be lost quickly, but general features may determine the solutions
- Assume the functions and terminal are sufficient
- Structural properties of the expected solution (uniformity, symmetry, depth, ...)
- Practical: 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 $k = 0$)
- 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 initialization
- Seeding: Start with many copies of good candidates

Riccardo Poli, William B Langdon, Nicholas F. McPhee (2008) A Field Guide to Genetic Programming.

Genetic Programming: General Points

- **Sufficiency** of the representation: Appropriate choice of non-terminals
- **Variables:** Terminals (variables) implied by the problem
- Is there a bug in the code? **Closure**: Typed algorithms, grammar based encoding
- **Program structure:** Terminals also for auxiliary variables or pointers to (automatically defined) functions
- There are no silver bullets: Expect **multiple runs** (each with a population of solutions)
- **Local search:** Terminals (numbers) can often be found by hill-climbing
- Can you trust your results? **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: Troubleshooting

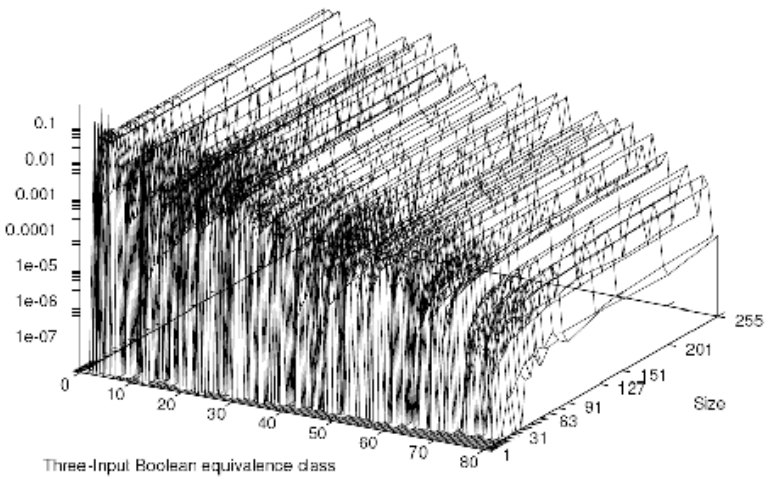
- Study your populations: Analyse means and variances of fitness, depth, size, code used, run time, ... and correlations among these
- Runs can be very long: Checkpoint results (e.g. mean fitness)
- Control bloat in order to obtain small efficient programs: Size limitations prevent unreasonable growth 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

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 analog 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 computerized techniques are needed to analyse the data

Genetic programming: Theory

- Schema theorem (sub-tree at a particular position)
  - worst case (Koza 1992)
  - exact for one-point crossover (Poli 2000)
  - for many types of crossover (Poli et al., 2003)
- Markov chain theory

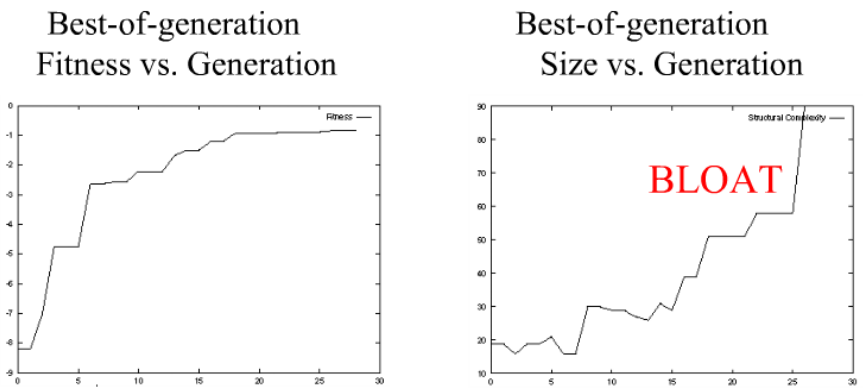


Two functions are equivalent if they coincide after a permutation of inputs. Program trees are composed of NAND gates.

- Distribution of fitness in search space (s. figure)
  - as the length of programs increases, the proportion of programs implementing a function approaches a limit
- Halting probability
  - for programs of length L is of order 1/L1/2, while the expected number of instructions executed by halting programs is of order L1/2.

Genetic programming: Bloat

- **Bloat** is an increase in program size that is not accompanied by any corresponding increase in fitness. **Problem:** The optimal solution might still be a large program

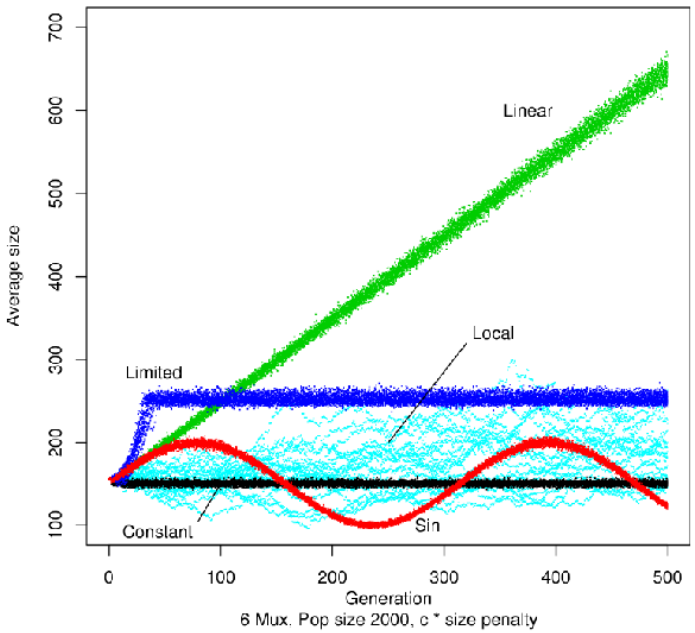


From: Genetic Programming by Riccardo Poli

- Theories (none of these is universally accepted) focus on
  - replication accuracy theory
  - inactive code
  - nature of program search-spaces theory
  - crossover bias (1-step-mean constant, but "Lagrange" variance)
- Size-evolution equation (similar to exact schema theorem)
- Practical solutions: Size and depth limits, parsimony pressure (fitness reduced by size: f - c l(i))

Genetic programming: Bloat Control

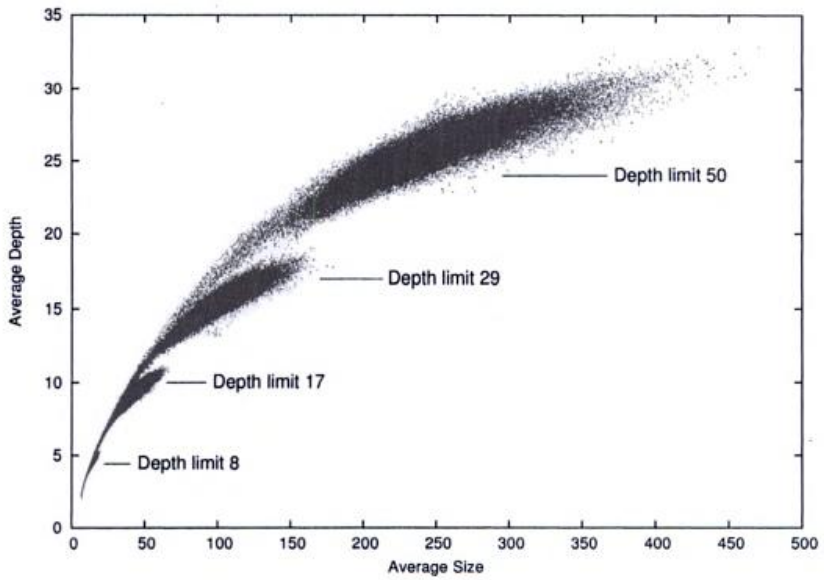
Constant: constant target size of 150.
Sin: target size
$\sin((\text{generation}+1)/50)\times 50+150$.
Linear: target size $(150+\text{generation})$.
Limited: no size control until the size reached 250 then hard-limited.
Local: adaptive target size
$c = -\text{Cov}(l,f)/\text{Var}(l)$: a certain amount of drift, avoided runaway bloat.



Average program size over 500 generations for multiple runs of the 6-MUX problem [decode a 2-bit address and return the value from the corresponding register] with various forms of parsimony pressure.

Riccardo Poli, William B Langdon, Nicholas F. McPhee (2008) A Field Guide to Genetic Programming.

Genetic programming: Theory



GP applied to "one-then-zeros" problem: independently of tree structure fitness is maximal if all nodes have a identical symbol. Expected to bloat, but doesn't. Why?

E. Crane and N. McPhee The Effects of Size and Depth Limits on Tree Based Genetic Programming

Genetic programming: Control parameters

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

Exact Schema Theory

Following: Genetic Programming by Riccardo Poli (University of Essex)

- Exact schema theoretic models of GP have become available only recently (first proof for a simplified case: Poli 2001)
- For a given schema H the selection/crossover/mutation process can be seen as a Bernoulli trial, because a newly created individual either samples or does not sample H
- Therefore, the number of individuals sampling H at the next generation, $m(H, t+1)$ is a binomially stochastic variable
- So, if we denote with $\alpha(H, t)$ the success probability of each trial (i.e. the probability that a newly created individual samples H), an exact schema theorem is simply $E[m(H, t+1)] = M\alpha(H, t)$, where M is the population size and $E[\cdot]$ is the mathematical expectation.

Exact Schema Theory

- Variable size tree structure does not permit the same definition of a schema as in GA
- A schema is a (sub-)tree with some "don't-care" nodes ($\Box$)
- A schema represents a primitive function (or a terminal)
- E.g. $H = (\Box\, x\, (+y\, \Box)$
  represents the programs
  $\{(+x(+x\, y)\, , (+x\, (+y\, y))\, , (*x\, (+y\, x))\, , \ldots\}$
  (prefix notation, $\Box$ can be terminal or non-terminal)

Exact Schema Theory

- Assume: Only reproduction and one-offspring crossover are performed (no mutation)
- $\alpha(H,t)$ the success probability can be calculated because the two operators are mutually exclusive

$$\begin{aligned}\alpha(H,t) &= Pr\left[\text{an individual in } H \text{is obtained via reproduction}\right] \\ &+ Pr\left[\text{an offspring matching } H \text{is produced by crossover}\right]\end{aligned}$$

- Reproduction is performed with probability p_r and crossover with probability p_c $(p_r + p_c = 1)$, so

$$\begin{aligned}\alpha(H,t) &= p_r Pr\left[\text{an individual in } H \text{is selected for cloning}\right] \\ &+ p_c Pr\left[\begin{array}{c}\text{parents and crossover points are} \\ \text{such that the offspring matches } H\end{array}\right]\end{aligned}$$

where $Pr\left[\text{an individual in } H \text{is selected for cloning}\right] = p(H,t)$

Exact Schema Theory

$$Pr\left[\begin{array}{c}\text{parents and crossover points are}\\ \text{such that the offspring matches } H\end{array}\right]$$

$$= \sum_{\substack{\text{For all pairs of}\\ \text{parent shapes } k,\, l}} \sum_{\substack{\text{For all crossover points}\\ i,\, j \text{ in shapes } k,\, l}} Pr\left[\begin{array}{c}\text{Choosing crossover points}\\ i \text{ and } j \text{ in shapes } k \text{ and } l\end{array}\right]$$

$$\times \quad Pr\left[\begin{array}{c}\text{Selecting parents with shapes } k \text{ and } l \text{such that if}\\ \text{crossed over at points } i \text{ and } j \text{ produce and offspring in } H\end{array}\right]$$

Exact Schema Theory

- Crossover excises a subtree rooted at the chosen crossover point in a parent , and replaces it with a subtree excises from the chosen crossover point in the other parent.
- This means that the offspring will have the right shape and primitives to match the schema of interest *if and only if*, after the excision of the chosen subtree, the first parent has shape and primitives compatible with the schema, and the subtree to be inserted has shape and primitives compatible with the schema.
- Assume that crossover points are selected with uniform probability

$$Pr\left[\begin{array}{c}\text{Choosing crossover points}\\ i \text{ and } j \text{ in shapes } k \text{ and } l\end{array}\right] = \frac{1}{\text{Nodes in shape } k} \times \frac{1}{\text{Nodes in shape } l}$$

Exact Schema Theory

$$Pr\left[\begin{array}{c}\text{Selecting parents with shapes } k \text{ and } l\text{such that if} \\ \text{crossed over at points } i \text{ and } j \text{ produce and offspring in } H\end{array}\right]$$

$$Pr\left[\begin{array}{c}\text{Selecting a root-donating parent with shape } k\text{such that its upper} \\ \text{part w.r.t crossover point } i \text{ matches the upper part of } H\text{w.r.t. } j\end{array}\right]$$

$$Pr\left[\begin{array}{c}\text{Selecting a subtree-donating parent with shape } l\text{such that its lower} \\ \text{part w.r.t crossover point } j \text{ matches the lower part of } H\text{w.r.t. } i\end{array}\right]$$

These two selection probabilities can be calculated exactly, but this requires a bit more work .... cf. R. Poli and N. F. McPhee (2003) General schema theory for GP with subtree swapping crossover: Parts I&II. *Evolutionary Computation* **11** (1&2).

Conclusions on GP

- In order to be successful GP algorithms need well structured problems and lots of 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 in GA in the news (many of these were actually GPs)
- GP provide an interesting view on the art of programming
- Exact schema theoretic models of GP have started shedding some light on fundamental questions regarding the how and why GP works and have also started providing useful recipes for practitioners.

Next time: Ant Colony Optimisation (ACO)