Genetic Algorithms and Genetic Programming

Lecture 9: (23/10/09)

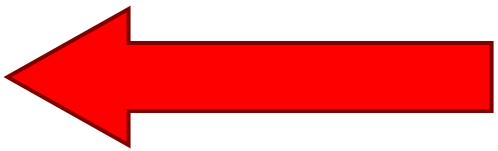
Genetic programming II

Michael Herrmann

michael.herrmann@ed.ac.uk, phone: 0131 6 517177, Informatics Forum 1.42

Overview

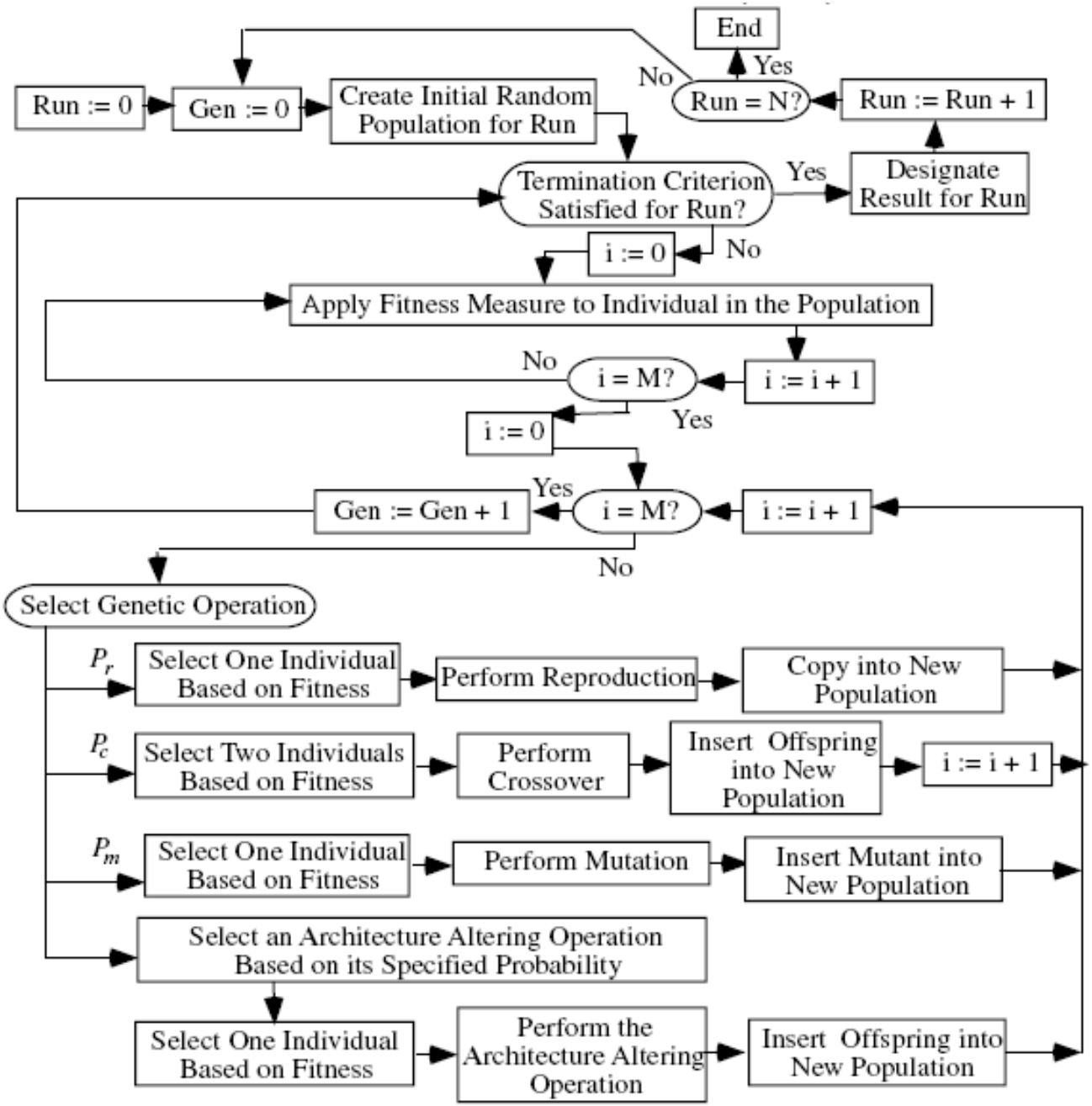
1. Introduction: History
2. The genetic code
3. The canonical genetic algorithm
4. Examples & Variants of GA
5. The schema theorem
6. Hybrid algorithms
7. Evolutionary robotics
8. Genetic Programming
9. Genetic Programming II
10. Practical issues



GP: Overview

- Evolution of random programs of a purpose specified by a fitness function
- Choose: non-terminals (functions), terminals
- Initialization (termination criterion)
- Closure, defaults, sufficiency
- Fitness cases
- Choose parameters (population size, probabilities)
- Selection
- Crossover and mutation.
- Particularly operators on trees: subtree replacement, exchange, shift (shrink and hoist)

GP Flowchart



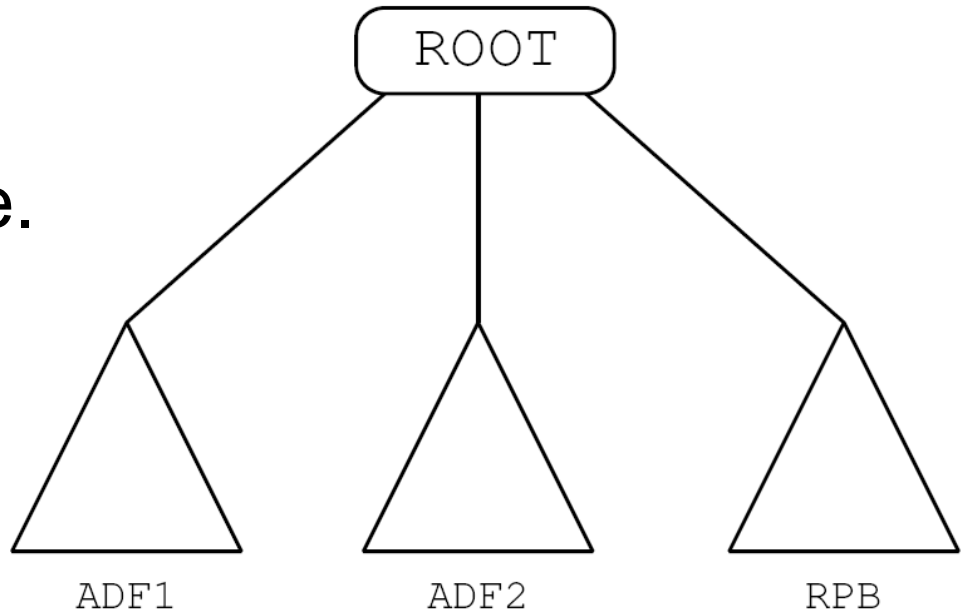
Initialization

- 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, …)
- Lagrange initialization:
  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.* For free at http://www.lulu.com/content/2167025

Automatically Defined Functions

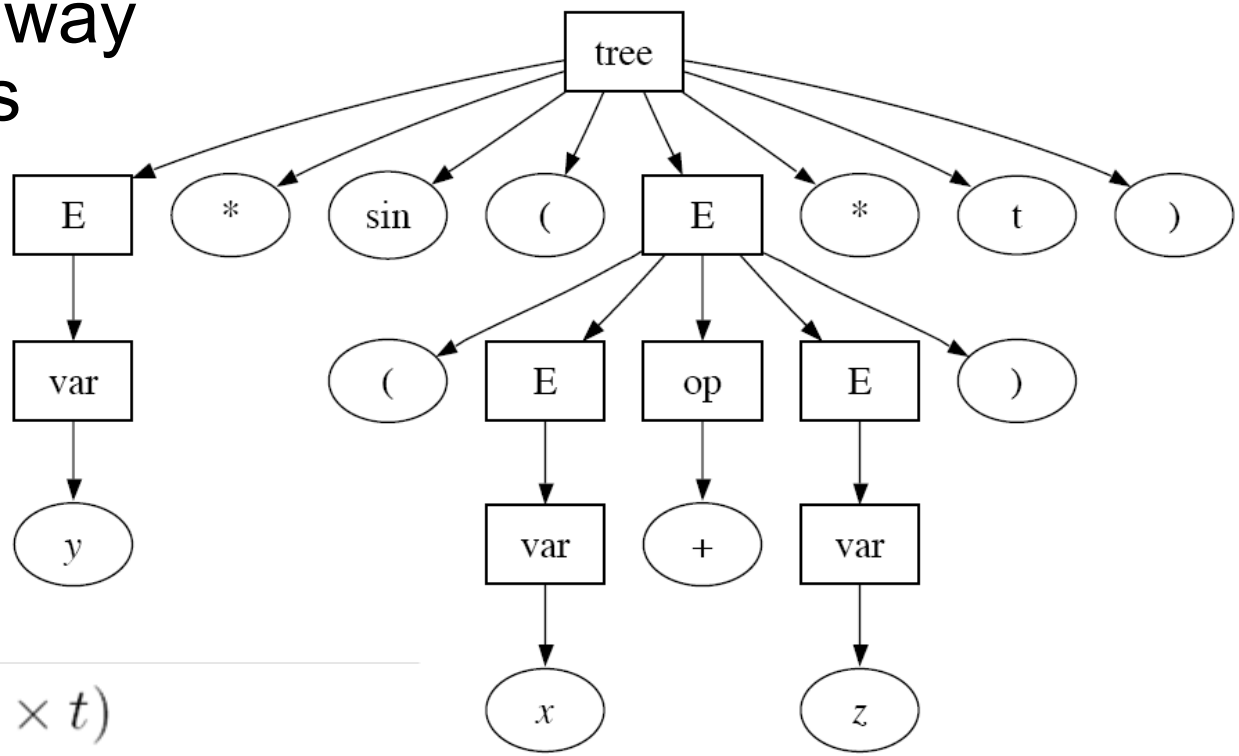
- “Efficient code”: Loops, subroutines, functions, classes, or … variables
- Automatically defined iterations (ADIs), automatically defined loops (ADLs) and automatically defined recursions (ADRs) provide means to reuse code. (Koza)
- Automatically defined stores (ADSs) provide means to reuse the result 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

Grammar-based Constraints

Constraints can either be included in the fitness function, but are more efficiently implemented by the operators. One way is to require individuals to be generatable by a grammar. Other approaches require types for terminals, functions and return values.



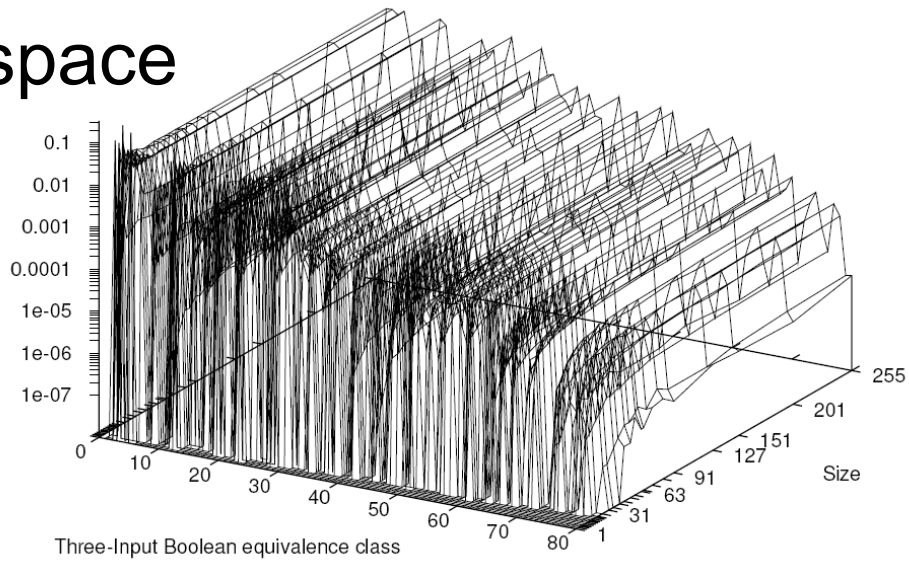
$$
\begin{aligned}
\mathit{tree} &::= E \times \sin(E \times t) \\
E &::= \texttt{var} \mid (E\ \texttt{op}\ E) \\
\texttt{op} &::= + \mid - \mid \times \mid \div \\
\texttt{var} &::= x \mid y \mid z
\end{aligned}
$$

Hybrid GP

- Information theoretic measures and minimum description length in fitness function
- Local search: Hill-climbing for adaptation of numerical values
- Co-evolution: Fitness depends on the individuals in other populations
- Editing: Apply regularly rules to increase efficiency (e.g. removing branches from a multiplication subtree that are always zero, type consistency check, grammatical GP)

GP Theory

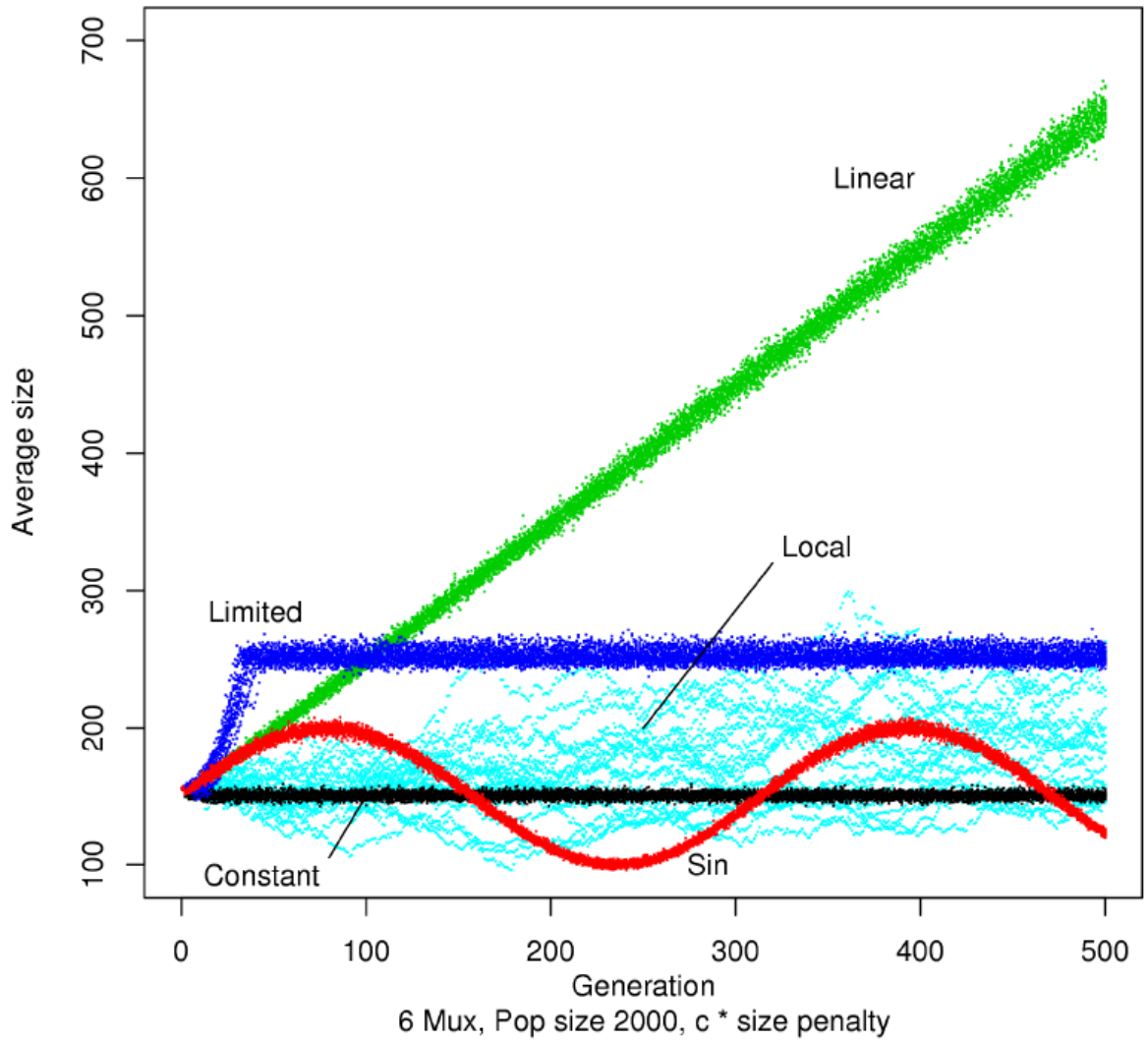
- Schema theorem (sub-tree at a particular position)
  - worst case (Koza 1992)
  - exact for one-point crossover (Poli 2000)
  - For many types of crossover (Poli et al., 2003)
- Markov chain theory
- Distribution of fitness in search space
  - as the length of programs increases, the proportion of programs implementing a function approaches a limit



- Halting probability
  - for programs of length L is of order $1/L^{1/2}$, while the expected number of instructions executed by halting programs is of order $L^{1/2}$.

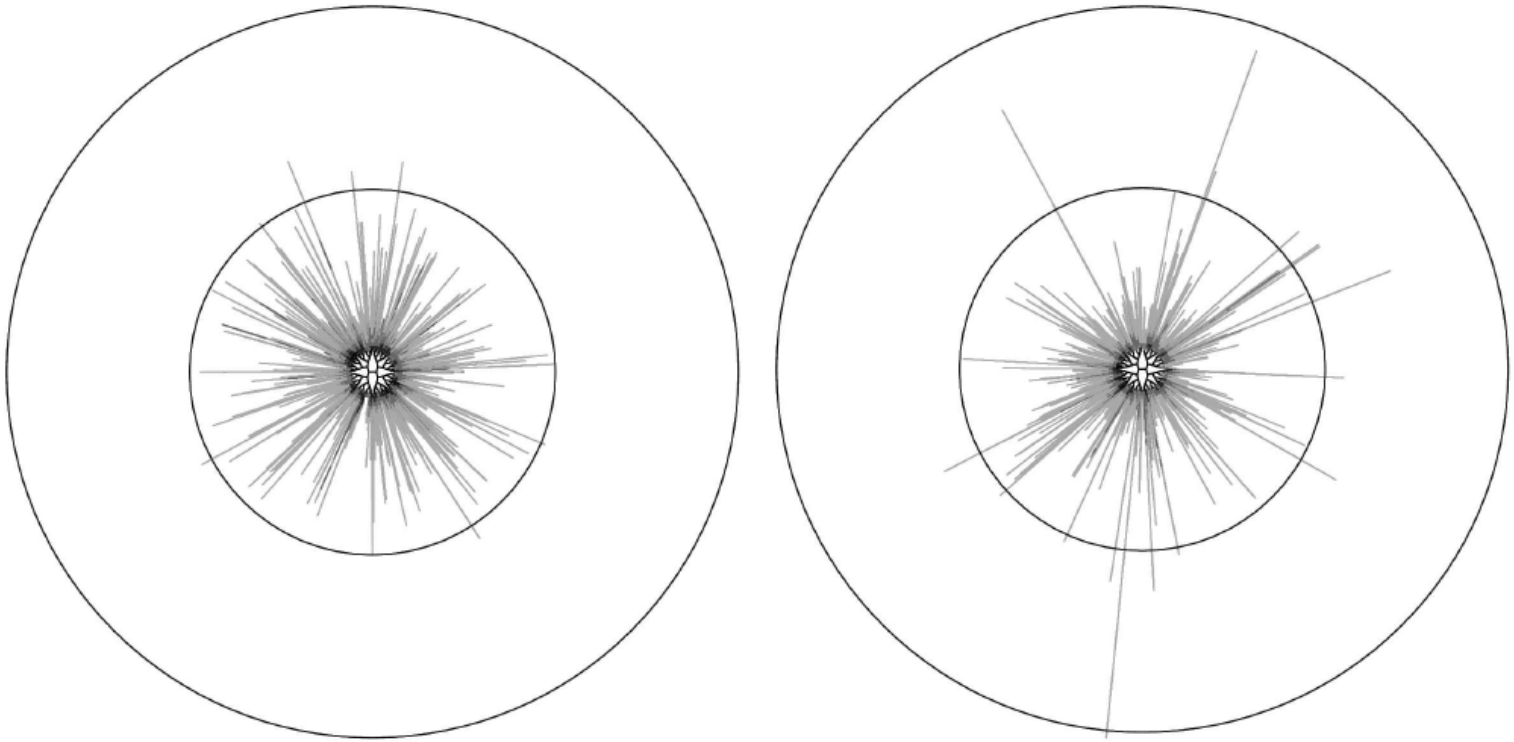
GP Theory: Bloat

- Bloat
  - increase in program size not accompanied by any corresponding increase in fitness
  - but: the optimal solution may be a large program
- Theories (non of these is universally accepted)
  - 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: $f - c\,\ell(i)$)



Plots of the evolution average size over 500 generations for multiple runs of the 6-MUX problem with various forms of covariant parsimony pressure. The “Constant” runs had a constant target size of 150. In the “Sin” runs the target size was sin((generation + 1)/50) × 50 + 150. For the “Linear” runs the target size was 150 + generation. The “Limited” runs used no size control until the size reached 250, then the target was held at 250. Finally, the “Local” runs used c = -Cov(ℓ, f)/Var(ℓ), which allowed a certain amount of drift but still avoided runaway bloat.

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



Visualisation of the size and shape of the entire population of 1,000 individuals in the final generation of runs using a depth limit of 50 (on the left) and a size limit of 600 (on the right). The inner circle is at depth 50, and the outer circle is at depth 100. These plots are from (Crane and McPhee, 2005) and were drawn using the techniques described in (Daida et al., 2005).

Troubleshooting

- Is there a bug in the code? Closure
- Can you trust your results? Crossvalidation
- There are no silver bullets: Expect multiple runs
- Small changes can have big effects
- Big changes can have no effect
- Study your populations
- Encourage diversity
- Embrace approximation: No program is error-free
- Control bloat
- Runs can be very long: Checkpoint results

Characteristics Suggesting the Use of GP

1. Discovering the size and shape of the solution
2. Reusing substructures
3. Discovering the number of substructures,
4. Discovering the nature of the hierarchical references among substructures,
5. Passing parameters to a substructure,
6. Discovering the type of substructures (e.g., subroutines, iterations, loops, recursions, or storage),
7. Discovering the number of arguments possessed by a substructure,
8. Maintaining syntactic validity and locality by means of a developmental process, or
9. Discovering a general solution in the form of a parameterized topology containing free variables

Strong Indicators for Using GA or ES

- The size and shape of the solution is known or fixed
- Ascertaining numerical parameters is the major issue
- Simplicity is a major consideration
- On-chip evolution the algorithm's logic is implemented on the chip in hardware

Fundamental Differences between GP and other Approaches to AI and ML

1. Representation: Genetic programming overtly conducts it search for a solution to the given problem in program space.
2. Role of point-to-point transformations in the search: Genetic programming does not conduct its search by transforming a single point in the search space into another single point, but instead transforms a set of points into another set of points.
3. Role of hill climbing in the search: Genetic programming does not rely exclusively on greedy hill climbing to conduct its search, but instead allocates a certain number of trials, in a principled way, to choices that are known to be inferior.
4. Role of determinism in the search: Genetic programming conducts its search probabilistically.
5. Role of an explicit knowledge base: None.
6. Role of formal logic in the search: None.
7. Underpinnings of the technique: Biologically inspired.

Cross-Domain Features

- Native representations are sufficient when working with genetic programming
- Genetic programming breeds “simulatability” (Koza)
- Genetic programming starts small
- Genetic programming frequently exploits a simulator’s built-in assumption of reasonableness
- Genetic programming engineers around existing patents and creates novel designs more frequently than it creates infringing solutions

Promising GP Application Areas

- Problem areas involving many variables that are interrelated in highly non-linear ways
- Inter-relationship of variables is not well understood
- A good approximate solution is satisfactory
  - design, control, 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, synthesis of designs for antennas
- Areas where you simply have no idea how to program a solution, but where the objective (fitness measure) is clear
- Problem areas where large computerized databases are accumulating and computerized techniques are needed to analyze the data

Open Questions/Research Areas

- Scaling up to more complex problems and larger programs
- Using large function and terminal sets.
- How well do the evolved programs generalise?
- How can we evolve nicer programs?
  - size, efficiency, correctness
- What sort of problems is GP good at/ not-so-good at?
- How does GP work? etc.

- **Reading:** J. Koza 1990, especially pp 8–14, 27–35, 42–43 (paper linked to web page)
- Riccardo Poli, William B Langdon, Nicholas F. McPhee (2008) *A Field Guide to Genetic Programming.* For free at http://www.lulu.com/content/2167025
- see also: http://www.genetic-programming.org http://www.geneticprogramming.us
- **Outlook:** Practical issues of EC

Alan Turing (1950) “Computing Machinery and Intelligence”

We cannot expect to find a good child-machine at the first attempt.

One must experiment with teaching one such machine and see how well it learns. One can then try another and see if it is better or worse. There is an obvious connection between this process and evolution:

- ‘Structure of the child machine’ = Hereditary material
- ‘Changes of the child machine’ = Mutations
- ‘Natural selection’ = Judgement of the experimenter