Genetic Programming Practice and Theory

Riccardo Poli
Department of Computing and Electronic Systems
University of Essex

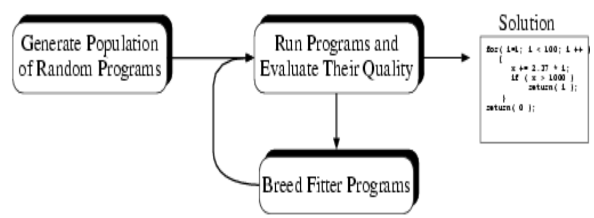
Overview

- Basics
- Examples and Applications
- Theory
- Conclusions

Genetic Programming

- GP is a *systematic* method for getting computers to *automatically* solve a problem starting from a *high-level statement* of what needs to be done.
- GP is a *domain-independent* method that genetically breeds a population of computer programs to solve a problem.

- GP *iteratively* transforms a population of programs into a new generation of programs.

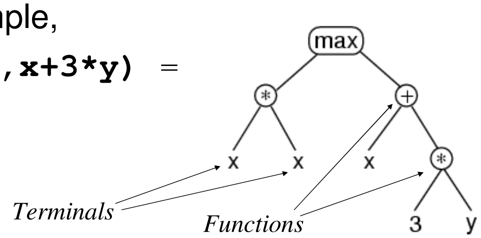


- GP applies analogues of *genetic operations* like sexual recombination, mutation, etc.

Program Representation

- Programs are expressed in GP as *syntax trees* rather than as lines of code.
- For example,

`max(x*x,x+3*y)` =



Prefix Notation

- GP trees and the corresponding expressions can be represented in *prefix notation.*
- In this notation, functions always precede their arguments. E.g.

`max(x*x,x+3*y)`

↓

`(max (* x x)(+ x (* 3 y)))`

Linear Representation

- If all functions have a fixed arity, the brackets become redundant in prefix-notation expressions.
- E.g.

```
(max (* x x)(+ x (* 3 y)))
```

↓

```
max * x x + x * 3 y
```

- So, often GP trees are stored internally as linear sequences of instructions.

Preparatory Steps of GP

Users need to specify:

1. The terminal set
2. The function set
3. The fitness measure
4. Certain parameters for controlling the run
5. The termination criterion and method for designating the *result* of the run

Terminal Set (Step 1)

- Steps 1 and 2 specify the ingredients that are available to create the computer programs (*primitive set*).
- The terminal set may consist of
  - The program's external inputs (e.g. `x`, `y`),
  - 0-arity functions (e.g. `rand()`, `go_left()`),
  - Numerical constants (e.g. 0.1, 3, $\mathcal{R}$).

Function Set (Step 2)

- For some problems, the function set may consist of merely the arithmetic functions (+, -, *, /) and a *conditional* branching operator.
- But all sort of functions are allowed, e.g.

Kind	Example(s)
Arithmetic	+, *, /
Mathematical	sin, cos, exp
Boolean	AND, OR, NOT
Conditional	IF-THEN-ELSE
Looping	FOR, REPEAT

- For many other problems, the primitive set includes specialized functions and terminals. E.g.
  - If the goal is to program a robot to mop the floor

    Function set = {*moving*, *turning*, *swishing the mop*}
  - If the goal is the automatic creation of a controller

    Function set = {*integrators*, *differentiators*, *leads*, *lags*, *gains*}

    Terminal set = {*reference signal, plant output*}
  - If the goal is the synthesis of analog electrical circuits

    Function set = {*transistors*, *capacitors*, *resistors*}

Syntax Errors Are Impossible

- **IF** all programs in the initial population of a run of GP are syntactically valid, executable programs,
- **AND** the genetic operations performed during the run are designed to produce offspring that are syntactically valid, executable programs,
- **THEN** every individual created during a run of GP is a syntactically valid, executable program.

Run-time Errors Can Be Avoided

- **IF** all functions in the primitive set can take as input the results produced by any other function or terminal (closure)
- **THEN** run-time errors are avoided.
- Sometime this requires modifying the primitive set appropriately, e.g. using *protected* versions of division, logarithm, square root, etc.

Fitness Measure (Step 3)

- The fitness measure is the mechanism for giving a high-level statement of the problem's requirements to the GP system.
- The first two preparatory steps define the search space whereas the fitness measure implicitly specifies the search's desired goal.

Fitness can be measured in terms of …

- The amount of error between its output and the desired output
- The amount of time (fuel, money, etc.) required to bring a system to a desired target state
- The accuracy of the program in recognizing patterns or classifying objects into classes
- The payoff that a game-playing program produces
- The compliance of a structure with user-specified design criteria, ...

- The fitness measure is, for many practical problems, multi-objective, i.e. it combines two or more different elements that are often in competition with one another.
- For many problems, each program in the population is executed over a representative sample of different fitness cases.
- Fitness cases may represent different values of the program's input(s), different initial conditions of a system, or different environments.

Control Parameters (Step 4)

- An important control parameter is the population size.
- Other control parameters include
  - The probabilities of performing the genetic operations,
  - The maximum size for programs, and
  - Other details of the run.

Termination Criterion (Step 5)

- We need to specify the *termination criterion* and the *method of designating the result* of the run.
- The termination criterion may include a maximum number of generations to be run as well as a problem-specific success predicate.
- The best-so-far individual is then harvested and designated as the *result* of the run.

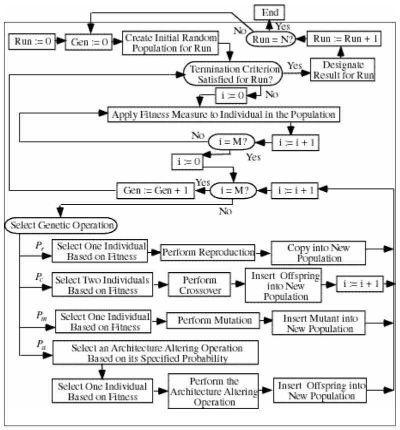
Executional Steps of GP

1. Randomly create an initial population of programs from the available primitives.
2. Iterate the following sub-steps until the termination criterion is satisfied:
   i. Execute each program and ascertain its fitness.
   ii. Select one or two program(s) from the population with a probability based on fitness to participate in genetic operations.
   iii. Create new individual program(s) by applying genetic operations with specified probabilities.
3. Return the best-so-far individual

Genetic Operations

- **Reproduction:** copy the selected individual program to the new population.
- **Crossover:** create new offspring program(s) by recombining randomly chosen parts from two selected programs.
- **Mutation:** create one new offspring program by randomly mutating a randomly chosen part of one selected program.
- **Architecture-altering operations:** choose an architecture-altering operation from the available repertoire and create one new offspring using it.

Flowchart of GP

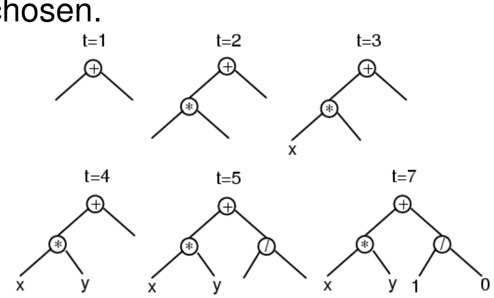


Random Program Generation

- The programs in the initial population are typically built by recursively generating a tree composed of random choices of functions and terminals.
- The initial individuals are usually generated subject to a pre-established maximum size.

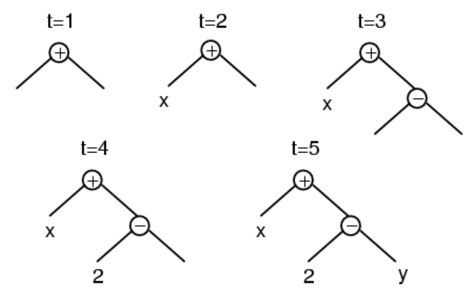
"Full" Initialisation Method

- Nodes are taken from the function set until a maximum tree depth is reached. Beyond that depth only terminals can be chosen.
- E.g.



"Grow" Initialisation Method

- It behaves like "full" except it allows the selection of nodes from the whole primitive set until the depth limit is reached.
- E.g.

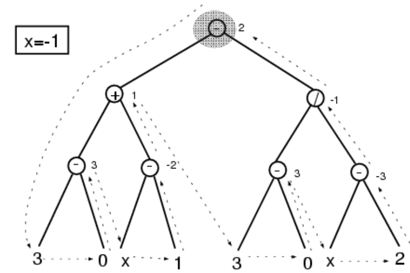


Fitness

- Normally, fitness evaluation requires executing the programs in the population multiple times within the GP system.
- A variety of execution strategies exist, including:
  - off-line or on-line compilation and linking,
  - virtual-machine-code compilation,
  - interpretation.

Program Interpretation

- Interpreting a program tree means recursively traversing the tree and executing its nodes only when their arguments are known.
- E.g.

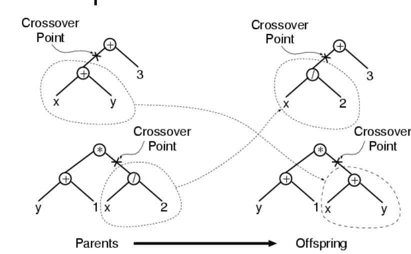


Selection

- Genetic operators are applied to individual(s) that are probabilistically selected based on fitness.
- Better individuals are favoured over inferior individuals.
- The most commonly employed methods for selecting individuals are tournament selection and fitness-proportionate selection.
- Both methods are not greedy.

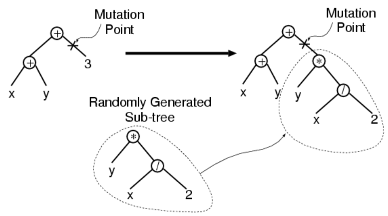
Sub-tree Crossover

- Given two parents crossover randomly selects a crossover point in each parent tree and swaps the sub-trees rooted at the crossover points.



Sub-tree Mutation

- Mutation randomly selects a mutation point in a tree and substitutes the sub-tree rooted there with a randomly generated sub-tree.



Examples

Toy Example

- Goal: to automatically create a computer program whose output is equal to the values of the quadratic polynomial x^2+x+1 in the range from –1 to +1.
- Step 1 – Definition of the Terminal Set:
  - The problem is to find a mathematical function of one independent variable, so the terminal set must include *x*.
  - In order to evolve any necessary coefficients, the terminal set also includes numerical constants.
  - That is: $T = \{X, \Re\}$, where $\Re$ denotes constant numerical terminals in some range (e.g. [–5.0,+5.0]).

- Step 2 – Definition of the Function Set:
  - One possible choice consists of the four ordinary arithmetic functions of *addition, subtraction, multiplication,* and *division*:

    F = {+, -, *, %}.
  - To avoid run-time errors, the division function % is protected: it returns a value of 1 when division by 0 is attempted, but otherwise returns the quotient of its two arguments.

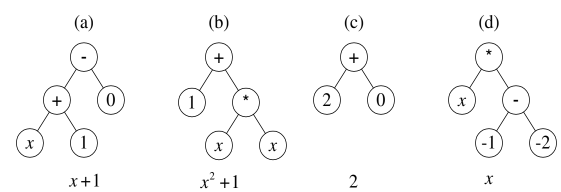
- Step 3 – Definition of the Fitness Function:
  - The fitness of an individual in the population must reflect how closely the output of a program comes to x^2+x+1.
  - The fitness measure could be defined as the value of the integral of the errors between the value of the individual mathematical expression and x^2+x+1.
  - Often this numerically approximated using dozens or hundreds of different values of the independent variable x.

- Step 4 – Fixing GP Parameters:
  - Population size: 4 (typically thousands or millions of individuals).
  - Crossover probability: 50% (commonly about 90%).
  - Reproduction probability: 25% (typically about 8%).
  - Mutation probability: 25% (usually about 1%)
  - Architecture-altering operation probability: 0% (frequently around 1%).

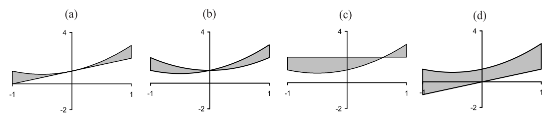
- Step 5 – Termination Criterion:
  - A reasonable termination criterion for this problem is that the run will continue from generation to generation until the fitness (error) of some individual gets below 0.01.
  - Often a maximum number of generations is also used as an additional stopping criterion.

Example Run

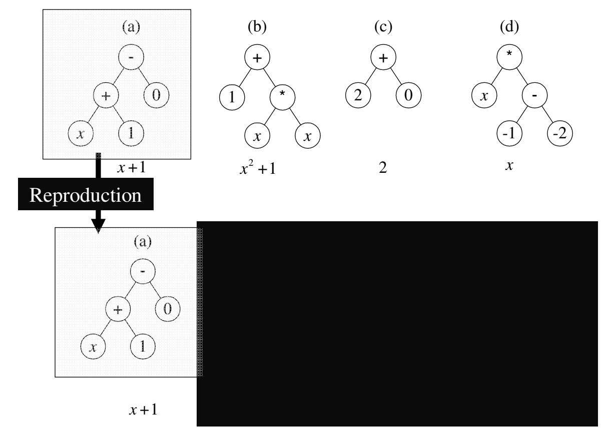
- Initial population of four randomly created individuals of generation 0

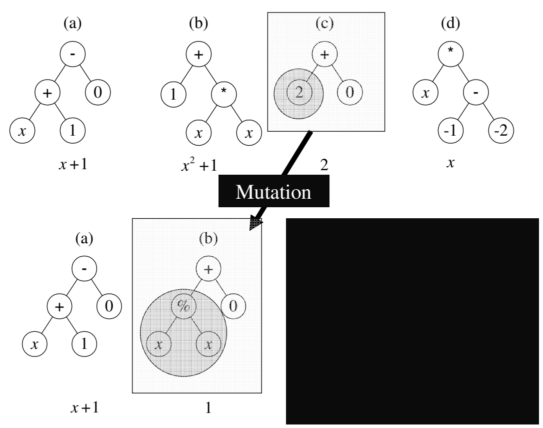


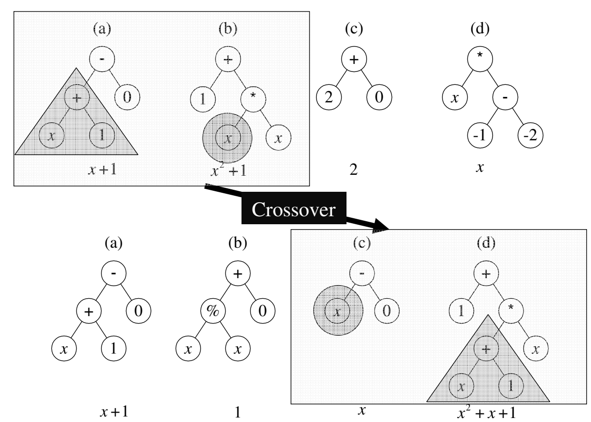
- The fitness of each of the four randomly created individuals of generation 0 is equal to the area between two curves.



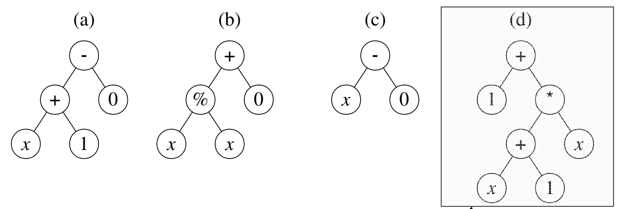
0.67 1.0 1.67 2.67







New Generation



	(a)	(b)	(c)	(d)
	$x+1$	1	x	x^2+x+1
Fitness	0.67	1.00	2.67	0

Solution (0<0.01)

Symbolic Regression

- Regression is a technique used to interpret experimental data. It consists in finding the coefficients of a prefixed function such that the resulting function best fits the data.
- If the fit is not good then the experimenter has to try with a different function until a good model for the data is found.
- The problem of symbolic regression consists in finding a good function (with its coefficients) that fits well the data points.

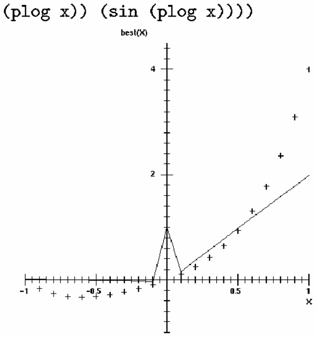
Real Symbolic Regression Run

- **Problem**: find the symbolic expression that best fits the data:
  - $\{x_i, y_i\}$ = {(-1.0,0.0) (-0.9,-0.1629) (-0.8,-0.2624)…(1.0,4.0)}
- GP parameters:

Parameter	*Value*
Population size	1000
Function set	{+ - * plog pexp sin cos pdiv}
Terminal set	{x}
Initial max depth	4
Initialisation method	Full
Number of generations	50
Crossover probability	0.7
Mutation probability	0
Fitness	- Sum of absolute errors

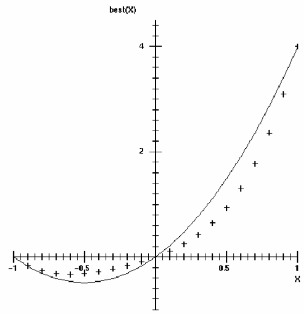
Best Program of Generation 1

```
(+ (- (plog (pexp x)) (+ (sin x) (- x x)))
   (+ (pexp (plog x)) (sin (plog x))))
```



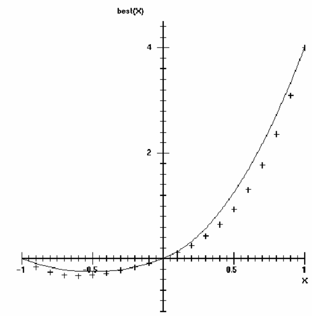
Best Program of Generation 3

```
(* (plog (- (sin x) (pexp x))) (+ (cos (* x x))
(+ (+ x x) (cos x))))
```



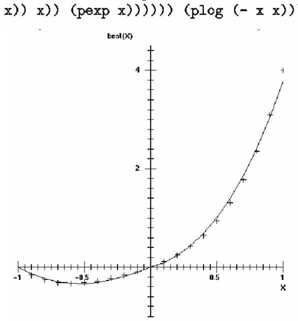
Best Program of Generation 6

```
(* (+ (+ (+ x (pexp (plog x))) (pdiv x x)) x) (plog (pexp x)))
```



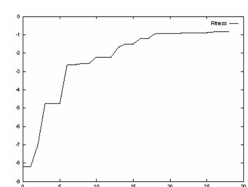
Best Program of Generation 26

```
(* (+ (+ (+ (* (+ (pexp (plog x)) (sin (plog (+ x x)))) x)
(pexp (plog x))) (pdiv x x)) x) (plog (pexp (+ (plog (- (sin
(+ (plog (pexp (plog (- (sin (+ (+ (+ x x) (pdiv (* (+ (+ (+ x
(pexp (plog x))) (pdiv x x)) x) x) x)) x)) (pexp x))))) x))
(pexp (plog (- (sin (+ (+ (+ x x) (pdiv (* (+ (+ (+ x (pexp (plog x)))
(pdiv x x)) x) x) x)) x)) (pexp x)))))) (plog (- x x))))))
```

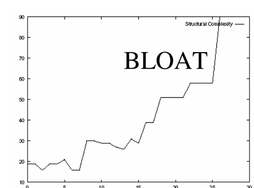


Evolution of Fitness and Size

Best-of-generation
Fitness vs. Generation



Best-of-generation
Size vs. Generation



Real World Applications

Human-competitive Results

- Getting machines to produce human-like results is *the* reason for the existence of the fields of artificial intelligence and machine learning.
- A result cannot acquire the rating of “human competitive” merely because it is endorsed by researchers *inside* the specialized fields that are attempting to create machine intelligence.
- A result produced by an automated method must earn the rating of “human competitive” independent of the fact that it was generated by an automated method.

Criteria for Human-competitiveness

A. The result was *patented* as an invention in the past, is an improvement over a patented invention, or would qualify today as a patentable new invention

B. The result is equal to or better than a result that was accepted as a *new scientific result* at the time when it was published in a peer-reviewed scientific journal

C. The result is equal to or better than a result that was placed into a *database or archive of results* maintained by an internationally recognized panel of scientific experts

D. The result is *publishable in its own right* as a new scientific result – independent of the fact that the result was mechanically created

Criteria for Human-competitiveness

E. The result is equal to or better than the most recent *human-created solution to a long-standing problem* for which there has been a succession of increasingly better human-created solutions

F. The result is equal to or better than a result that was considered an *achievement in its field* at the time it was first discovered

G. The result solves a *problem of indisputable difficulty* in its field

H. The result *holds its own or wins a regulated competition* involving human contestants (in the form of either live human players or human-written computer programs)

Pre-2004 GP Human-competitive Results

- 36 human-competitive results
- 23 instances where GP has *duplicated* the functionality of a previously patented invention, *infringed* a previously *patented invention*, or created a patentable *new invention*
- 15 instances where GP has created an entity that either *infringes* or *duplicates* the functionality of a previously patented *20th-century invention*
- 6 instances where GP has done the same with respect to an *invention patented after January 1, 2000*
- 2 instances where GP has created a *patentable new invention* (general-purpose controllers).

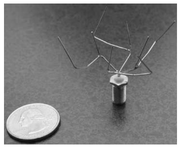
A selection of results

1. Creation of a better-than-classical quantum algorithm for Grover's database search problem
2. Creation of a quantum algorithm for the depth-two AND/OR query problem that is better than any previously published result
3. Creation of a soccer-playing program that won its first two games in the Robo Cup 1997 competition
4. Creation of four different algorithms for the transmembrane segment identification problem for proteins
5. Creation of a sorting network for seven items using only 16 steps
6. Synthesis of 60 and 96 decibel amplifiers
7. Synthesis of analog computational circuits for squaring, cubing, square root, cube root, logarithm, and Gaussian functions
8. Synthesis of a real-time analog circuit for time-optimal control of a robot
9. Synthesis of an electronic thermometer

10. Creation of a cellular automata rule for the majority classification problem that is better than the Gacs-Kurdyumov-Levin (GKL) rule and all other known rules written by humans
11. Synthesis of topology for a PID-D2 (proportional, integrative, derivative, and second derivative) controller
12. Synthesis of NAND circuit
13. Simultaneous synthesis of topology, sizing, placement, and routing of analog electrical circuits
14. Synthesis of topology for a PID (proportional, integrative, and derivative) controller
15. Synthesis of a voltage-current conversion circuit
16. Creation of PID tuning rules that outperform the Ziegler-Nichols and Astrom-Hagglund tuning rules
17. Creation of three non-PID controllers that outperform a PID controller that uses the Ziegler-Nichols or Astrom-Hagglund tuning rules

Human-competitive-result competition

- Held at GECCO 2004-2007
- Example winners:
  - Automated Quantum Programming, L. Spector
  - An Evolved Antenna for Deployment on NASA's Space Technology 5 Mission, J. Lohn et al.



Understanding GP

Motivation

Understanding GP Search Behaviour with Empirical Studies

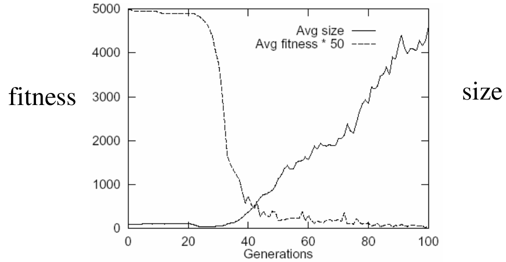
- We can perform many GP runs with a small set of problems and a small set of parameters
- We record the variations of certain numerical descriptors.
- Then, we suggest explanations about the behaviour of the system that are compatible with (and could explain) the empirical observations.

Problem with Empirical Studies

- GP is a complex adaptive system with zillions of degrees of freedom.
- So, any small number of descriptors can capture only a fraction of the complexities of such a system.
- Choosing which problems, parameter settings and descriptors to use is an art form.
- Plotting the wrong data increases the confusion about GP's behaviour, rather than clarify it.

Example: Bloat

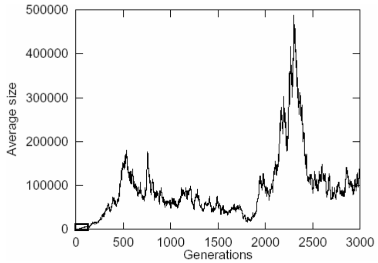
- *Bloat* = growth without (significant) return in terms of fitness. E.g.



- Bloat exists and continues forever, right?

Why do we need mathematical theory?

- Empirical studies are rarely conclusive



- Qualitative theories can be incomplete

Search Space Characterisation

How many programs in the search space?

n_d = **Number of trees of depth at most *d***

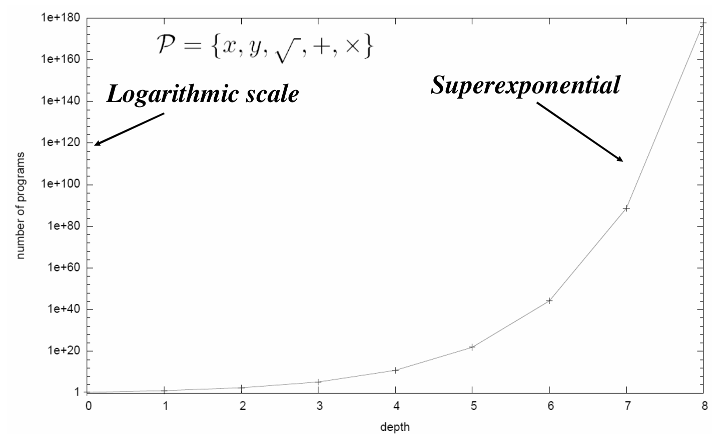
$$n_0 = |\mathcal{P}_0| \qquad n_d = \sum_{a=0}^{a_{\max}} |\mathcal{P}_a| \times (n_{d-1})^a$$

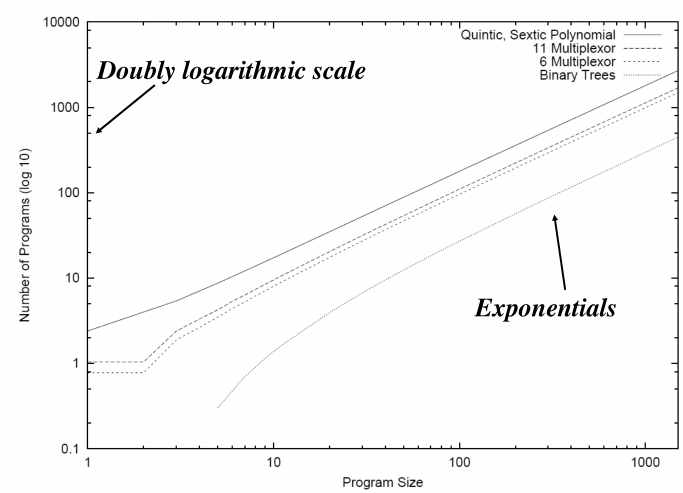
Example

$$\mathcal{P} = \{x, y, \sqrt{\ }, +, \times\}$$

$$a_{\max} = 2,\ \mathcal{P}_0 = \{x, y\},\ \mathcal{P}_1 = \{\sqrt{\ }\}\quad \mathcal{P}_2 = \{+, \times\}$$

$$\begin{aligned} n_0 &= 2 \\ n_1 &= 2 + 1 \times (n_0) + 2 \times (n_0)^2 = 12 \\ n_2 &= 2 + 1 \times (n_1) + 2 \times (n_1)^2 = 302 \end{aligned}$$



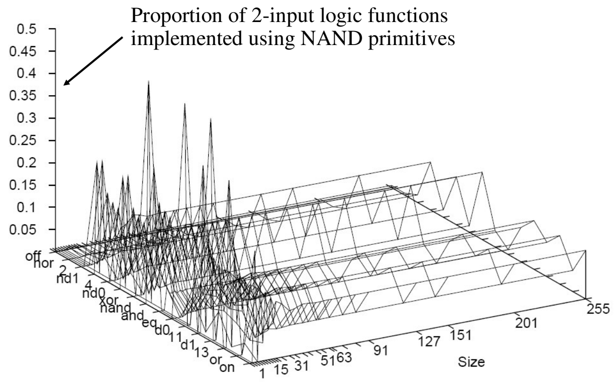


GP cannot possibly work!

- The GP search space is immense, and so any search algorithm can only explore a tiny fraction of it (e.g. 10^{-1000} %).
- Does this mean GP cannot possibly work?

  Not necessarily.
- We need to know the ratio between the size of solution space and the size of search space

{d0,d1,NAND} search space



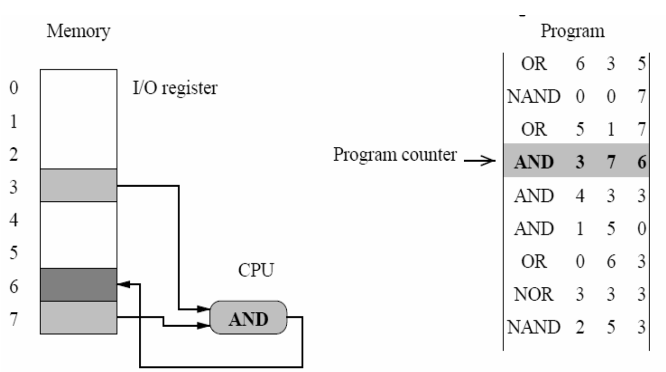
Limiting distribution

- Empirically is has been shown that as program length grows the distribution of functionality reaches a limit
- So, beyond a certain length, the proportion of programs which solve a problem is constant
- Since there are exponentially many more long programs than short ones, in GP

$$\boxed{\frac{\text{size of the solution space}}{\text{size of the search space}} = \text{constant}}$$

- Proofs?

Linear model of computer



States, inputs and outputs

- Assume *n* bits of memory
- There are 2^n states.
- At each time step the machine is in a state, *s*

Instructions

- Each instruction changes the state of the machine from a state *s* to a new *s*′, so instructions are maps from binary strings to binary strings of length *n*

  E.g. if $n = 2$, AND m_0 m_1 → m_0 is represented as

m_0	m_1	m'_0	m'_1
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	1

=

0	0	0	1	0	0	1	1

Behaviour of programs

- A program is a sequence of instructions
- So also the behaviour of a program can be described as a mapping from initial states *s* to corresponding final states *s*′

- For example,

  AND m0 m1 → m0

  NOP

  OR m0 m1 → m0

  AND m0 m1 → m0

→

m_0	m_1	m'_0	m'_1
0	0	0	0
0	1	1	1
1	0	0	0
1	1	1	1

↓

0	0	1	1	0	0	1	1

Does the behaviour tend to a limiting distribution?

- Two primitives: AND m0 m1 → m0 OR m0 m1 → m0

0	0	0	1	1	0	1	1

Identity function (no instruction executed yet)

AND m0 m1 → m0 1/2

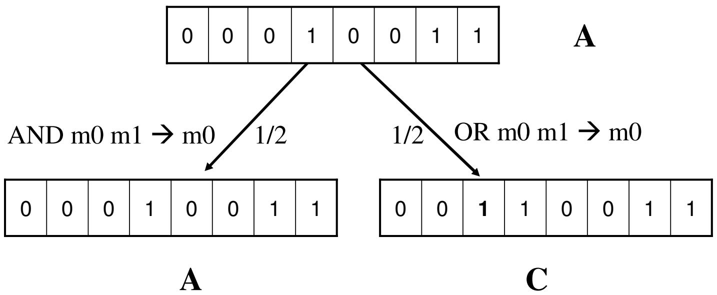
1/2 OR m0 m1 → m0

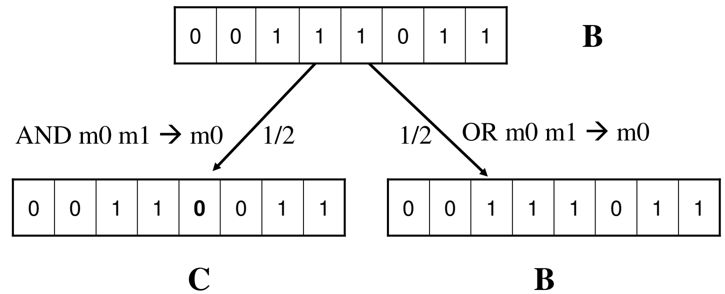
0	0	0	1	**0**	0	1	1

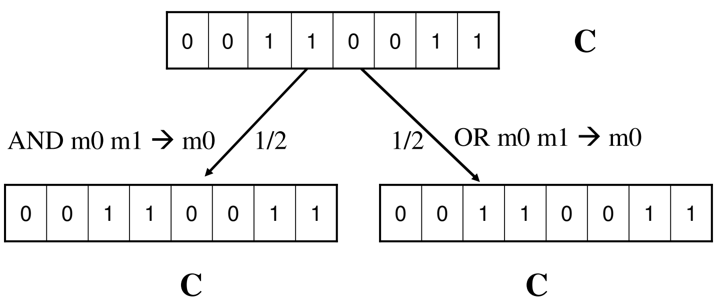
A

0	0	**1**	1	1	0	1	1

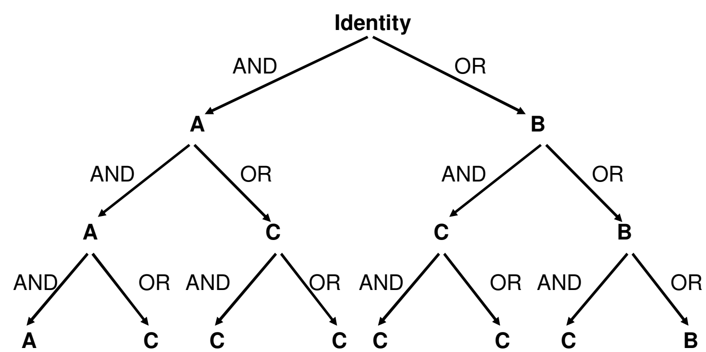
B







Probability tree



Distribution of behaviours

Program length	Behaviour A	Behaviour B	Behaviour C	Identity
0	0	0	0	1
1	½	½	0	0
2	¼	¼	½	0
3	1/8	1/8	¾	0
4	1/16	1/16	7/8	0
∞	0	0	1	0

Yes….

- …for this primitive set the distribution tends to a limit where only behaviour **C** has non-zero probability.
- Programs in this search space tend to copy the initial value of m1 into m0.

Markov chain proofs of limiting distribution

- Using Markov chain theory we have proved that a limiting distributions of functionality exists for a large variety of CPUs
- There are extensions of the proofs from linear to tree-based GP.
- See Foundations of Genetic Programming book for an introduction to the proof techniques.

So what?

- Generally instructions lose information. Unless inputs are protected, almost all long programs are constants.
- Write protecting inputs makes linear GP more like tree GP.
- No point searching above threshold?
- Predict where threshold is? Ad-hoc or theoretical.

Implication of
|solution space|/|search space|=constant

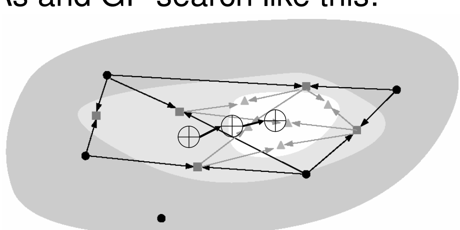
- GP can succeed if
  - the constant is not too small or
  - there is structure in the search space to guide the search or
  - the search operators are biased towards searching solution-rich areas of the search space

  or any combination of the above.

GP Search Characterisation

GA and GP search

- GAs and GP search like this:

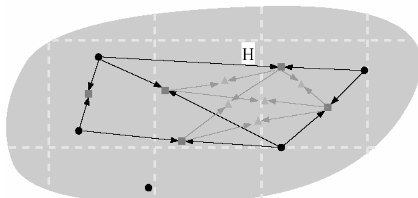


- How can we understand (characterise, study and predict) this search?

Schema Theories

- Divide the search space into subspaces (*schemata*)
- Characterise the schemata using *macroscopic* quantities
- Model how and why the individuals in the population move from one subspace to another (*schema theorems*).

Example



- The number of individuals in a given schema *H* at generation *t*, *m(H,t), is* a good descriptor
- A *schema theorem* models mathematically how and why *m(H,t)* varies from one generation to the next.

Exact Schema Theorems

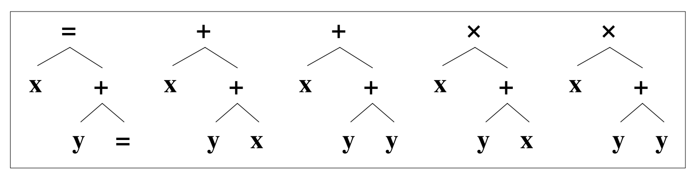
- The selection/crossover/mutation process is a random coin flip (Bernoulli trial). New individuals are either in schema *H* or not.
- So, $m(H,t+1)$ is a binomial stochastic variable.
- Given the success probability of each trial $\alpha(H,t)$, an **exact schema theorem** is

$$\boxed{E[m(H,t+1)] = M\,\alpha(H,t)}$$

Exact Schema Theory for GP with Subtree Crossover

GP Schemata

- Syntactically, a *GP schema* is a tree with some “don’t care” nodes (“=”) that represent *exactly one* primitive.
- Semantically, a schema is the set of all programs that match size, shape and defining nodes of such a tree.



Creation of individuals via crossover is a compound event

{create individual} =

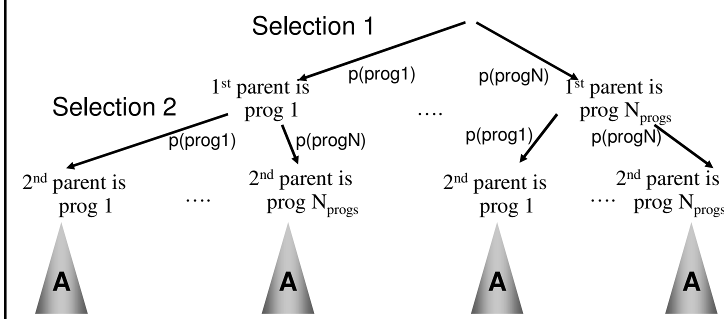
{select parent 1,

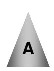
select parent 2,

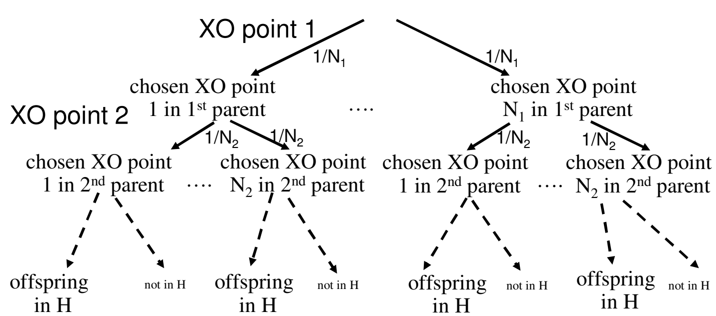
choose crossover point 1,

choose crossover point 2 }

Selection-Crossover Probability tree



Subtree 



N_1 = size(parent1)
N_2 = size(parent2)

Microscopic schema model

$\alpha(H,t)$ = sum of products of probabilities along paths leading to offspring in H

- Problems:
  - many paths ➔ many terms to evaluate (most=0)
  - r.h.s. quantities are not about schemata
  - model misses regularities in creation process
- Can we do better?

Regularities

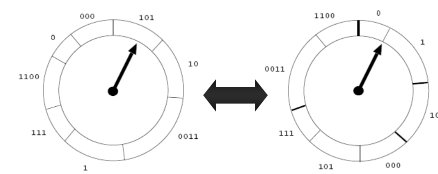
- The process of crossover point selection is independent from the actual primitives in the parent tree.
- The probability of choosing a particular crossover point depends only on the actual size and shape of the parent.
- For example, the probability of choosing any crossover point in the program

  (+ x (+ y x))

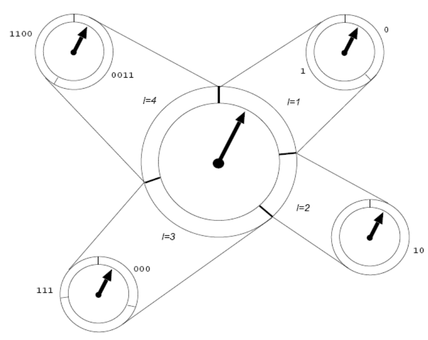
  is identical to the probability of choosing any crossover point in

  (AND D1 (OR D1 D2))

Fragmenting selection

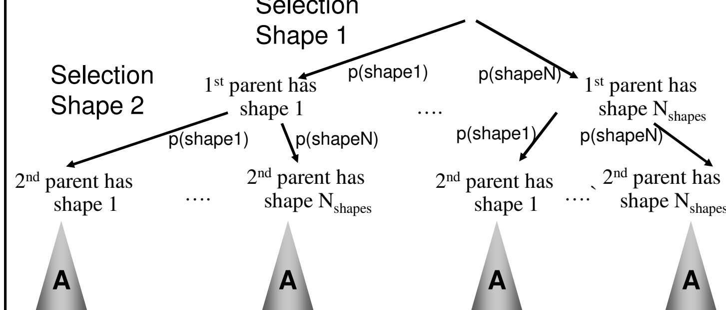


{select parent} = {select size/shape, select individual of that size/shape}

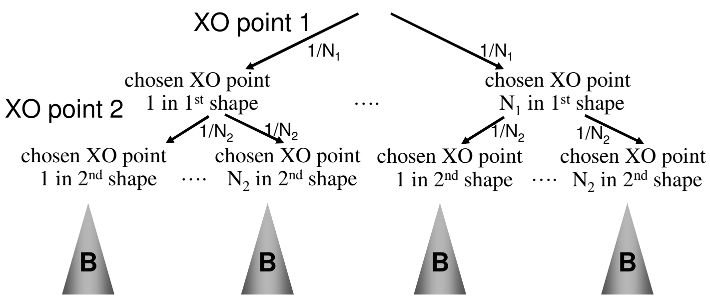


can be postponed

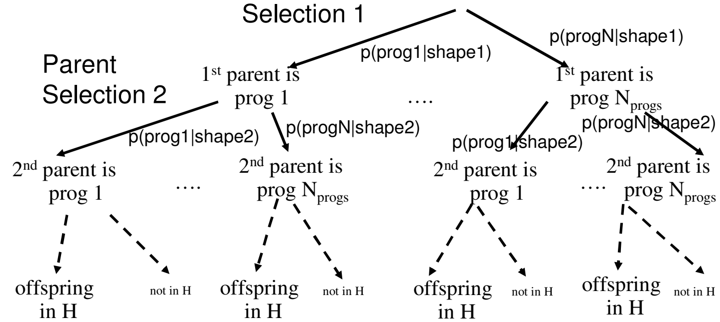
Selection-XO Probability Tree revisited



Subtree A revisited



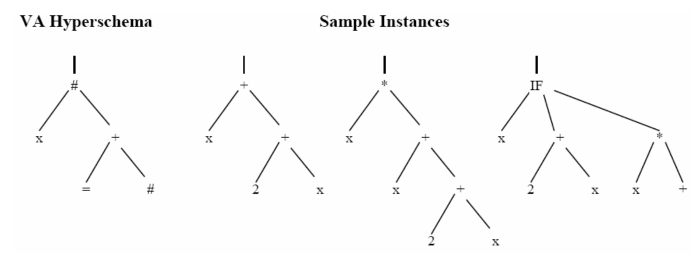
Subtree B (take 1)



Variable Arity Hyperschemata

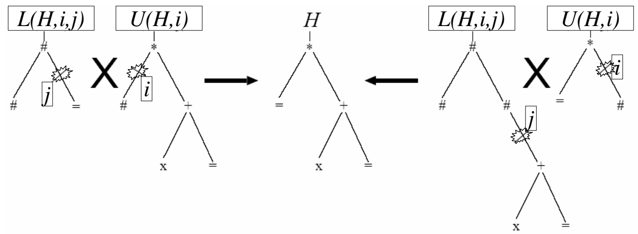
- A *GP* variable arity hyperschema is a tree with internal nodes from $F \cup \{=, \#\}$ and leaves from $T \cup \{=, \#\}$.
  - = is a “don't care” symbols which stands for exactly one node
  - \# is a more general “don’t care” that represents either a valid subtree or a tree fragment depending on its arity

- For example, (# x (+ = #))



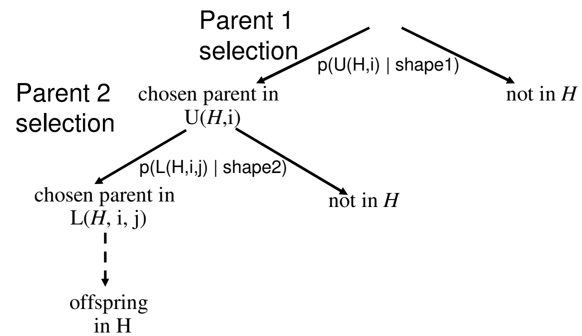
Upper and lower building blocks

Variable arity hyperschemata express which parents produce instances of a schema



Crossing over at points *i* and *j* any individual in *L(H,i,j)* with any individual in *U(H,i)* ➔ offspring in *H*

Subtree B (take 2)



Bayes

$$p(U(H,i) \mid shape1) = \frac{p(U(H,i) \cap shape1)}{p(\,shape1)}$$

$$p(L(H,i,j) \mid shape2) = \frac{p(L(H,i,j) \cap shape2)}{p(\,shape2\,)}$$

Exact GP Schema Theorem for Subtree Crossover (2001)

- Schema theorem for selection + 100% standard GP crossover

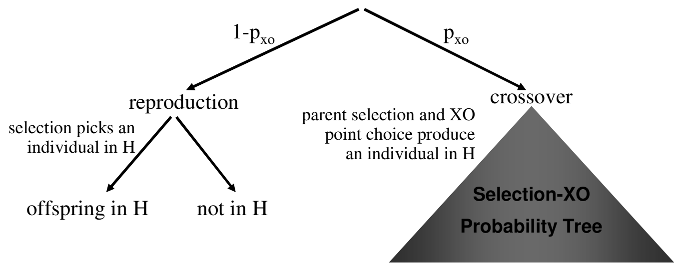
$$\alpha(H,t)= \sum_{k,l} \frac{1}{N(G_k)N(G_l)} \sum_{i \in H \cap G_k} \sum_{j \in G_l} p(U(H,i) \cap G_k, t)\, p(L(H,i,j) \cap G_l, t)$$

shape1 shape2 size(shape1)=N_1 size(shape2)=N_2

XO points in shape1 XO points in shape2

To reproduce or not to reproduce …

- Let us assume that also reproduction is performed.
- Creation probability tree for a schema *H*:



Exact GP Schema Theorem with Reproduction, Selection, Crossover

$$\alpha(H,t) = (1 - p_{xo})p(H,t) + p_{xo} \sum_{k,l} \frac{1}{N(G_k)N(G_l)} \sum_{i \in H \cap G_k} \sum_{j \in G_l} p(U(H,i) \cap G_k, t)\, p(L(H,i,j) \cap G_l, t).$$

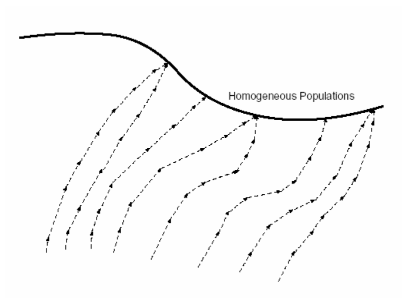
So what?

- A model is as good as the predictions and the understanding it can produce
- So, what can we learn from schema theorems?

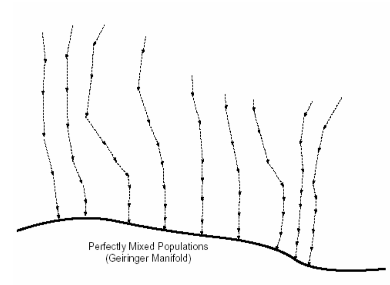
Lessons

- Operator biases
- Size evolution equation
- Bloat control
- Optimal parameter setting
- Optimal initialisation
- …

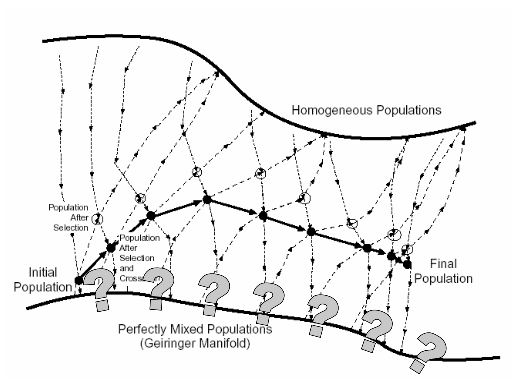
Selection Bias



Crossover Bias



So where is evolution going?



GP with subtree XO pushes the population towards a Lagrange distribution of the 2nd kind

$$\Pr\{n\} = (1 - a p_a) \binom{an+1}{n} (1 - p_a)^{(a-1)n+1} p_a^n$$

Proportion of programs with *n* internal nodes

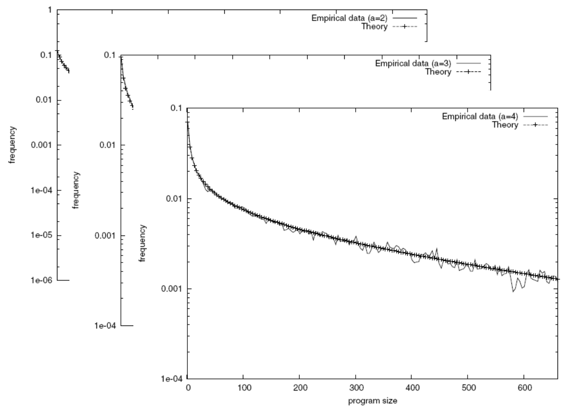
$$p_a = \frac{2\mu_0 + (a-1) - \sqrt{((1-a) - 2\mu_0)^2 + 4(1-\mu_0^2)}}{2a(1+\mu_0)}$$

Mean function arity

Mean program size

Note: uniform selection of crossover points

- Theory is right!



Sampling probability under Lagrange

- Probability of sampling a particular program of size *n* under subtree crossover

$$p_{\text{sample}}(n) = \frac{(1 - ap_a)}{\mathcal{F}^n \mathcal{T}^{(a-1)n+1}} \binom{an+1}{n} (1 - p_a)^{(a-1)n+1} \, p_a^n$$

- So, GP samples short programs much more often than long ones

Allele Diffusion

- The fixed-point distribution for linear, variable-length programs under GP subtree crossover is

$$\Phi(h_1 h_2 \ldots h_N, \infty) = \Phi((=)^N, \infty) \times \prod_{i=1}^{N} c(h_i)$$

with

$$c(a) = \sum_{n \geq 0} \Phi((=)^n a, 0)$$

- Crossover attempts to push the population towards distributions of primitives where each primitive of a given arity is equally likely to be found in any position in any individual.
- The primitives in a particular individual tend not just to be swapped with those of other individuals in the population, but also to diffuse within the representation of each individual.
- Experiments with unary GP confirm the theory.

Size Evolution

- The *mean size* of the programs at generation t is

$$\mu(t) = \sum_l N(G_l)\ \Phi(G_l,t)$$

where

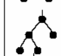
G_l = set of programs with shape l

$N(G_l)$ = number of nodes in programs in G_l

$\Phi(G_l,t)$ = proportion of population of shape l at generation t

- E.g., for the population:

x, (+ x y) (- y x) (+ (+ x y) 3)

l	G_l	$N(G_l)$	$\Phi(G_l,t)$
1		1	1/4
2		3	2/4
3		5	1/4
4		5	0
⋮	⋮	⋮	⋮

$$\mu(t) = 1 \times \frac{1}{4} + 3 \times \frac{2}{4} + 5 \times \frac{1}{4} = 3$$

Size Evolution under Subtree XO

- In a GP system with symmetric subtree crossover

$$E[\mu(t+1)] = \sum_l N(G_l)\ p(G_l,t)$$

where

$p(G_l,t)$ = probability of selecting a program of shape l from the population at generation t

- The mean program size evolves as if selection only was acting on the population

Conditions for Growth

- Growth can happen only if

$$E[\mu(t+1)-\mu(t)] > 0$$

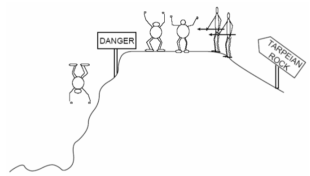
- Or equivalently

$$\sum_l N(G_l)\ [p(G_l,t) - \Phi(G_l,t)] > 0$$

$$\sum_{G_l \in G_{\text{large}}} (N(G_l) - \mu(t))p(G_l,t) > \sum_{G_l \in G_{\text{small}}} (\mu(t) - N(G_l))p(G_l,t)$$

Tarpeian Bloat Prevention

- To prevent growth one needs
  - To increase the selection probability for below-average-size programs
  - To decrease the selection probability for above-average-size programs



Conclusions

Theory

- In the last few years the theory of GP has seen a formidable development.
- Today we understand a lot more about the nature of the GP search space and the distribution of fitness in it.
- Also, schema theories explain and predict the syntactic behaviour of GAs and GP.
- We know much more as to where evolution is going, why and how.

- Theory primarily provides explanations, but many recipes for practice have also been derived (initialisation, sizing, parameters, primitives, anti bloat, …)
- So, theory can help design competent algorithms
- Theory is hard and slow: empirical studies are important to direct theory and to corroborate it.

Turing's Intuition

- In his seminal 1948 paper entitled "Intelligent Machinery," Turing identified three ways by which human-competitive machine intelligence might be achieved. In connection with one of those ways, Turing said:

  "There is the genetical or evolutionary search by which a combination of genes is looked for, the criterion being the survival value."

- Turing did not specify how to conduct the "genetical or evolutionary search" for machine intelligence, but in his 1950 paper "Computing Machinery and Intelligence," he wrote

  "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, by the identifications

  Structure of the child machine = Hereditary material

  Changes of the child machine = Mutations

  Natural selection = Judgment of the experimenter"

- So, *over 50 years ago* Turing perceived that one approach to machine intelligence would involve an evolutionary process in which
  - a description of a computer program (the hereditary material)
  - undergoes progressive modification (mutation)
  - under the guidance of natural selection (what we now call "fitness").

- Turing also understood the need to *evaluate objectively* the behaviour exhibited by machines, to avoid human biases when assessing their intelligence.
- This led him to propose an imitation game, now know as the *Turing test for machine intelligence*, whose goals are summarised by Arthur Samuel's position statement

  "[T]he aim [is] … to get machines to exhibit behavior, which if done by humans, would be assumed to involve the use of intelligence." [Arthur Samuel, 1983]

GP Has Started Fulfilling Turing and Samuel's Dreams

- *GP has started fulfilling Turing's dream by providing us with a systematic method, based on Darwinian evolution, for getting computers to automatically solve problems.*
- To do so, GP simply requires a high-level statement of what needs to be done (and enough computing power).

- Today GP certainly cannot produce computer programs that would pass the full Turing test for machine intelligence.
- But GP has been able to solve tens of difficult problems with human-competitive results.
- No other AI technique has done this

  "John Koza's genetic programming approach to machine discovery can invent solutions to more complex specifications than any other I have seen." [John McCarthy]

- These are a small step towards fulfilling Turing and Samuel's dreams, but they are also early signs of things to come.
- In a few years' time GP will be able to *routinely* and *competently* solve important problems for us in a variety of specific domains of application, becoming an essential collaborator for many of human activities.
- This will be a remarkable step forward towards achieving true, human-competitive machine intelligence.

More information

- W.B. Langdon & R. Poli, *Foundations of Genetic Programming*, Springer, 2002.
- J. Koza & R. Poli, Genetic Programming, in INTROS tutorial book, Chapter 5, Kluwer, 2005
- R. Poli, Exact Schema Theory for Genetic Programming and Variable-Length Genetic Algorithms with One-Point Crossover, *Genetic Programming and Evolvable Machines* 2(2), 2001.
- R. Poli & N. F. McPhee, Schema Theory for GP with Sub-tree Crossover, Parts I & II, *Evolutionary Computation*, 11(1):53-66 & 11(2):169–206, 2003.
- R. Poli, N. F. McPhee & J.E. Rowe, Exact Schema Theory and Markov Chain Models for Genetic Programming and Variable-length Genetic Algorithms with Homologous Crossover, *Genetic Programming and Evolvable Machines*, 5(1):31-70, 2004.