Genetic Programming Theory

Riccardo Poli
Department of Computer Science
University of Essex

Overview

- Search space characterisation
  - Program search spaces
  - Recursive structure
  - Limiting fitness distributions
  - Halting probability
- GP search characterisation
  - Schema Theory
  - Lessons and implications
- Conclusions

Introduction

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 hypothesize explanations about the behaviour of the system that are compatible with (and could explain) the empirical observations.

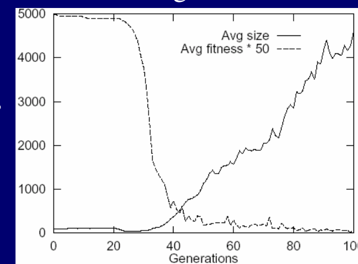
Problem with Empirical Studies

- 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.

fitness

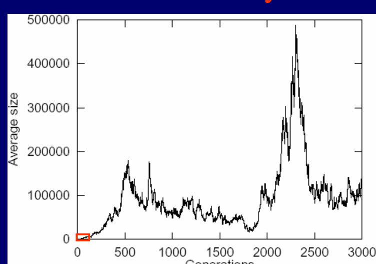


size

- Bloat exists and continues forever, right?

Why do we need mathematical theory?

- Empirical studies are rarely conclusive

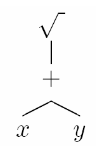


- Qualitative theories can be incomplete

Search Space Characterisation

GP search spaces

- Representation



$\sqrt{\ } + x\ y$

mov R1 R2
add R1 72
jump 1

mov R1 R2 opcode	add R1 2 opcode	jump 1 opcode

Grammar

$$
\begin{aligned}
E &\rightarrow \mathcal{P}_0 \\
E &\rightarrow \mathcal{P}_1 E \\
E &\rightarrow \mathcal{P}_2 E E \\
&\ \vdots \\
E &\rightarrow \mathcal{P}_{a_{\max}} \overbrace{E \cdots E}^{a_{\max}} \\
\mathcal{P}_0 &\rightarrow p_{01} \,|\, p_{02} \,|\, \cdots \,|\, p_{0|\mathcal{P}_0|} \\
\mathcal{P}_1 &\rightarrow p_{11} \,|\, p_{12} \,|\, \cdots \,|\, p_{1|\mathcal{P}_1|} \\
&\ \vdots \\
\mathcal{P}_{a_{\max}} &\rightarrow p_{a_{\max}1} \,|\, p_{a_{\max}2} \,|\, \cdots \,|\, p_{a_{\max}|\mathcal{P}_{a_{\max}}|}
\end{aligned}
$$

GP primitives

Recursive nature of search space

$\mathcal{P} = \bigcup_a \mathcal{P}_a$ **= Primitive Set**

Ω_d^t **= Set of trees of depth at most *d***

$$\Omega_0^t = \mathcal{P}_0 \qquad \Omega_d^t = \bigoplus_{a=0}^{a_{\max}} \mathcal{P}_a \otimes (\Omega_{d-1}^t)^{\otimes a}$$

Union ***Cartesian product***

$X \times Y = \{(x,y) | x \in X \text{ and } y \in Y\}$.

Example

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

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

$$\Omega_0^t = \{x, y\}$$

$$
\begin{aligned}
\Omega_1^t &= \bigoplus_{a=0}^{2} \mathcal{P}_a \otimes (\Omega_0^t)^{\otimes a} \\
&= \mathcal{P}_0 \oplus \mathcal{P}_1 \otimes \Omega_0^t \oplus \mathcal{P}_2 \otimes \Omega_0^t \otimes \Omega_0^t \\
&= \{x, y\} \oplus \{\sqrt{\ }\} \otimes \{x, y\} \oplus \{+, \times\} \otimes \{x, y\} \\
&\quad \otimes \{x, y\}
\end{aligned}
$$

= {x, y, (√x), (√y), (+ x x), (+ x x), (+ x y), (+ y x), (+ y y), (+ x x), (× x x), (× x y), (× y x), (× y y) }

$$
\begin{aligned}
\Omega_2^t = & \bigoplus_{a=0}^{2} \mathcal{P}_a \otimes (\Omega_1^t)^{\otimes a} \\
= & \{x,y\} \oplus \{\sqrt{}\} \otimes \Omega_1^t \oplus \{+,\times\} \otimes \Omega_1^t \otimes \Omega_1^t \\
= & \{x,y\} \\
\oplus & \{\sqrt{}\} \otimes \{x,y\} \\
\oplus & \{\sqrt{}\} \otimes \{\sqrt{}\} \otimes \{x,y\} \\
\oplus & \{\sqrt{}\} \otimes \{+,\times\} \otimes \{x,y\} \otimes \{x,y\} \\
\oplus & \{+,\times\} \otimes \{x,y\} \otimes \{x,y\} \\
\oplus & \{+,\times\} \otimes \{\sqrt{}\} \otimes \{x,y\} \otimes \{x,y\} \\
\oplus & \{+,\times\} \otimes \{+,\times\} \otimes \{x,y\} \otimes \{x,y\} \otimes \{x,y\} \\
\oplus & \{+,\times\} \otimes \{x,y\} \otimes \{\sqrt{}\} \otimes \{x,y\} \\
\oplus & \{+,\times\} \otimes \{\sqrt{}\} \otimes \{x,y\} \otimes \{\sqrt{}\} \otimes \{x,y\} \\
\oplus & \{+,\times\} \otimes \{+,\times\} \otimes \{x,y\} \otimes \{x,y\} \otimes \{\sqrt{}\} \otimes \{x,y\} \\
\oplus & \{+,\times\} \otimes \{x,y\} \otimes \{+,\times\} \otimes \{x,y\} \otimes \{x,y\} \\
\oplus & \{+,\times\} \otimes \{\sqrt{}\} \otimes \{x,y\} \otimes \{+,\times\} \otimes \{x,y\} \otimes \{x,y\} \\
\oplus & \{+,\times\} \otimes \{+,\times\} \otimes \{x,y\} \otimes \{x,y\} \otimes \{+,\times\} \otimes \{x,y\} \otimes \{x,y\}
\end{aligned}
$$

How many programs in the search space?

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

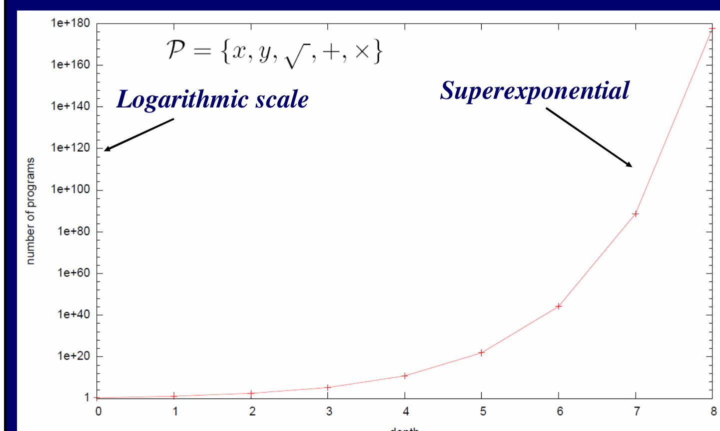
$$n_0 = |\mathcal{P}_0| \qquad n_d = \sum_{a=0}^{a_{\max}} |\mathcal{P}_a| \times (n_{d-1})^a$$

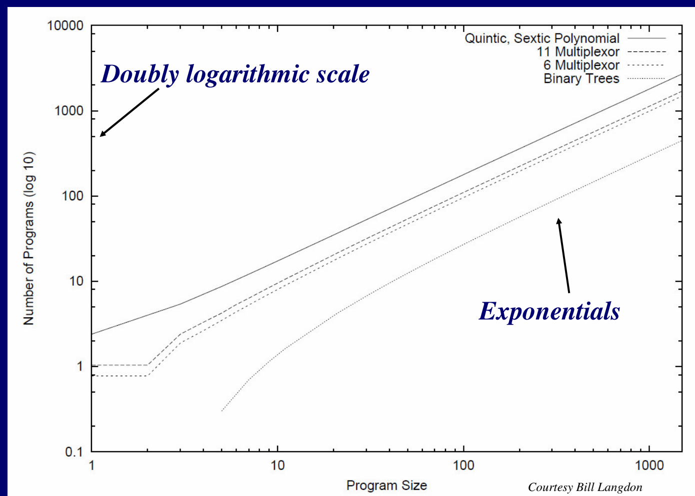
Example

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

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

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



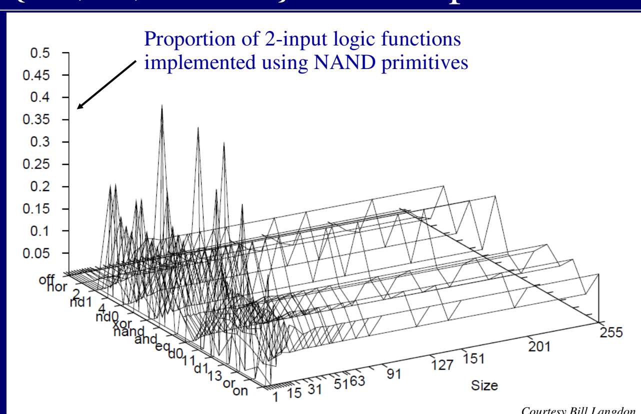


GP cannot possibly work!

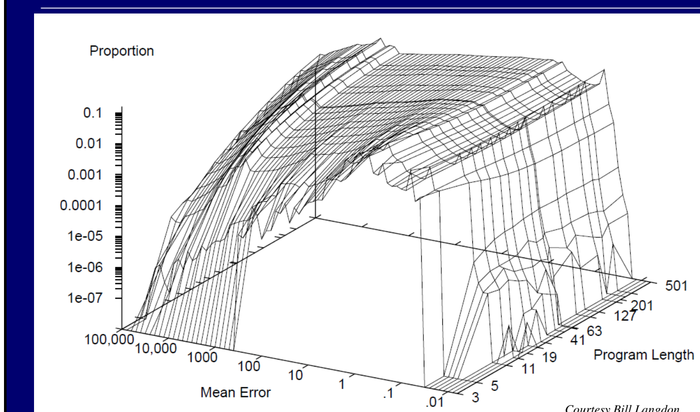
- The GP search space is immense, and so any search algorithm can only explore a tiny fraction of it (e.g. 10^{-1000} %).
- Does this mean GP cannot possibly work?

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

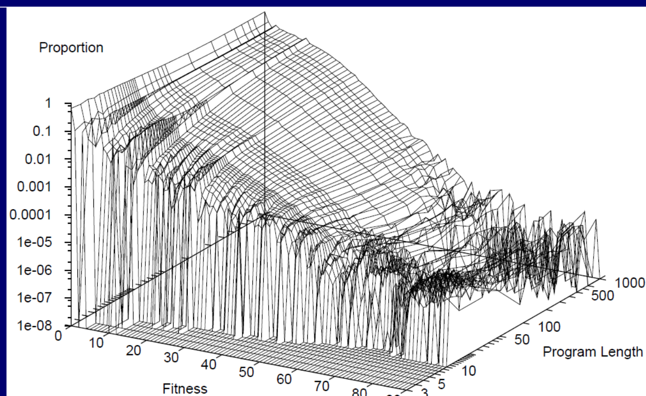
{d0,d1,NAND} search space



Sextic polynomial



Artificial Ant



Courtesy Bill Langdon

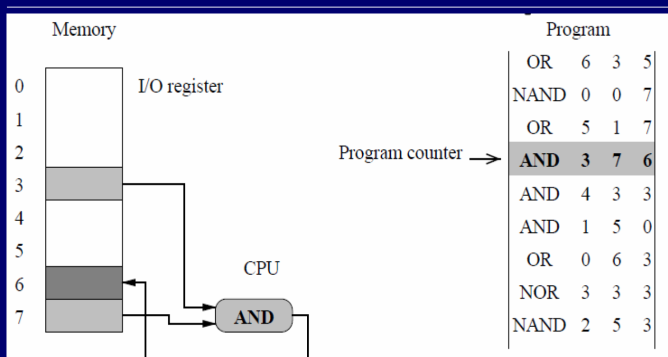
Limiting distribution

- 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

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

- Proofs?

Linear model of computer



Courtesy Bill Langdon

- Input and output registers are part of memory.
- Memory is initially zero (except input registers).
- Linear GP program is a sequence of instructions.
- CPU fetches operands from memory, performs operation and writes answer into memory (overwriting previous contents).
- When a program stops, the final answer can be read from the output register.

States, inputs and outputs

- Assume n bits of memory of which k are input bits and h are output bits
- There are 2^n states.
- At each time step the machine is in a state, s
- By setting the inputs (and zeroing the rest) we can place the machine in 2^k different initial states $s_1 \ldots s_{2^k}$ out of the 2^n available

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_i to corresponding final states s'_i

- 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

0	0	0	1	0	0	1	1

A

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	0	0	1	1

C

0	0	1	1	1	0	1	1

B

AND m0 m1 → m0 1/2 | 1/2 OR m0 m1 → m0

0	0	1	1	0	0	1	1

C

0	0	1	1	1	0	1	1

B

0	0	1	1	0	0	1	1

C

AND m0 m1 → m0 1/2 | 1/2 OR m0 m1 → m0

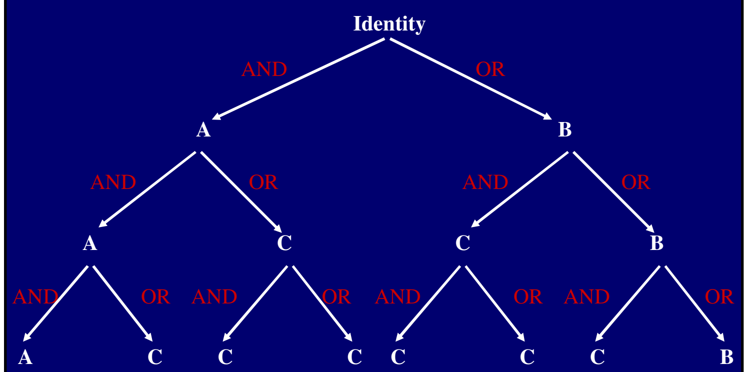
0	0	1	1	0	0	1	1

C

0	0	1	1	0	0	1	1

C

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.

Behaviour vs. functionality

- If the number of input bits k is smaller than n, since we zero the remaining bits, only 2^k different initial states s_1 … s_{2^k} out of the 2^n available are possible.
- So the behaviour of a program can be described as a mapping from the input states s_i to corresponding final states s'_i

  (not necessarily distinct)

- Often there are more memory bits than output bits
- So, many final states s'_i are indistinguishable from the output viewpoint.
- Therefore, the functionality of a program is a truth table with k inputs and h outputs.
- So, functionality is a coarse-grained version of behaviour
- The limiting distribution for functionality can be derived from the limiting distribution for behaviour.

Example

- Two primitives: AND m0 m1 → m0 and OR m0 m1 → m0
- Inputs: m0 and m1
- Output: m1
- Behaviours B =

Behaviours B =	0	0	1	1	1	0	1	1
C =	0	0	1	1	0	0	1	1
and Identity =	0	0	0	1	1	0	1	1
	↑		↑		↑		↑	

are indistinguishable

Distribution of functionality

Program length	Functionality A	Functionality B/C/Identity
0	0	1
1	½	½
2	¼	¾
3	1/8	7/8
4	1/16	15/16
∞	0	1

Markov chain proofs of limiting distribution

- Using Markov chain theory Bill Langdon has proven that a limiting distributions of functionality exists for a large variety of CPUs
- These include:
  - Cyclic. Increment, decrement and NOP.
  - Bit flip. Flip bit$_i$ and NOP.
  - Any non-reversible
  - Any reversible
  - CCNOT (Toffoli gate).
  - The "average" computer
  - AND, NAND, OR, NOR

- 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 win 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.

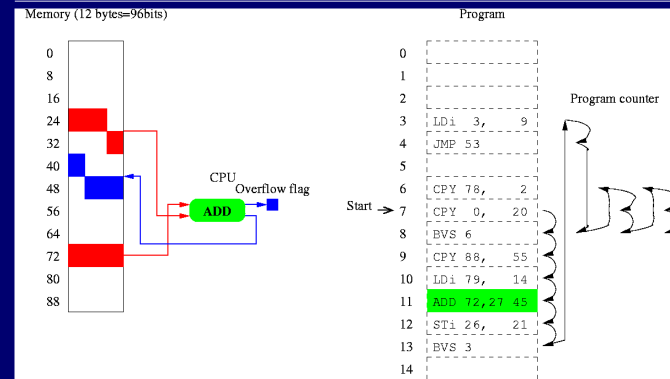
What about Turing complete GP?

- Memory and loops make linear GP Turing complete, but what is the effect search space and fitness?
- Does the distribution of functionality of Turing complete programs tend to a limit as programs get bigger?

T7 Minimal Turing Complete CPU

- 7 instructions
- Arithmetic unit is ADD. From + all other operations can be obtained. E.g.
  - Boolean logic
  - SUB, by adding complement
  - Multiply, by repeated addition (subroutines)
- Conditional (Branch if oVerflow flag Set)
- Move data in memory
- Save and restore Program Counter (i.e. Jump)
- Stop if reach end of program

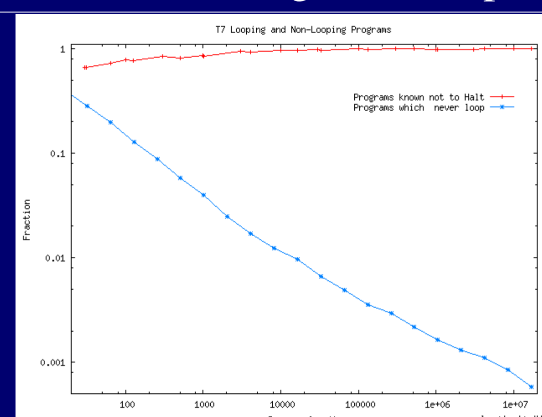
T7 Architecture



Experiments

- There are too many programs to test them all. Instead we gather statistics on random samples.
- Chose set of program lengths 30 to 16777215
- Generate 1000 programs of each length
- Run them from random start point with random input
- Program terminates if it obeys the last instruction and this is not a jump
- How many stop?

Almost all T7 Programs Loop



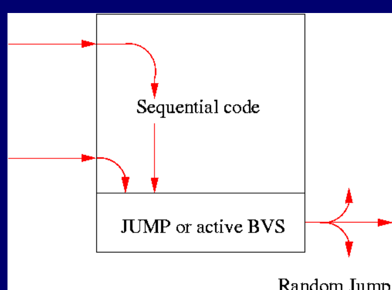
Model of Random Programs

- Before any repeated instructions;
  - random sequence of instructions and
  - random contents of memory.
- 1 in 7 instructions is a jump to a random location

Model of Random Programs

- T7 instruction set chosen to have little bias.
  - I.e. every state is ≈equally likely.
  - Overflow flag set half the time.
  - So 50% of conditional jumps BVS are active.
- (1+0.5)/7 instructions takes program counter to a random location.
- Implies for long programs, lengths of continuous instructions (i.e. without jumps) follows a geometric distribution with mean 7/1.5=4.67

Program segment = random code ending with a random jump



Forming Loops:Segments model

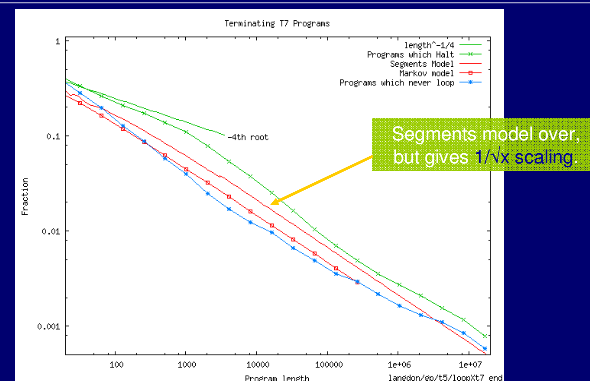
- Segments model assumes whole program is broken into N=L/4.67 segments of equal length of continuous instructions.
- Last instruction of each is a random jump.
- By the end of each segment, memory is re-randomised.
- Jump to any part of a segment, part of which has already been run, will form a loop.
- Jump to any part of the last segment will halt the program.

Probability of Halting

- i segments run so far. Chance next segment will
  - Form first loop = i/N
  - Halt program = 1/N
  - (so 1-(i+1)/N continues)
- Chance of halting immediately after segment i

  = 1/N × (1-2/N) (1-3/N) (1-4/N) … (1-i/N)
- Total halting probability given by adding these gives $\boxed{\approx \mathrm{sqrt}(\pi/_{2N}) = O(N^{-½})}$

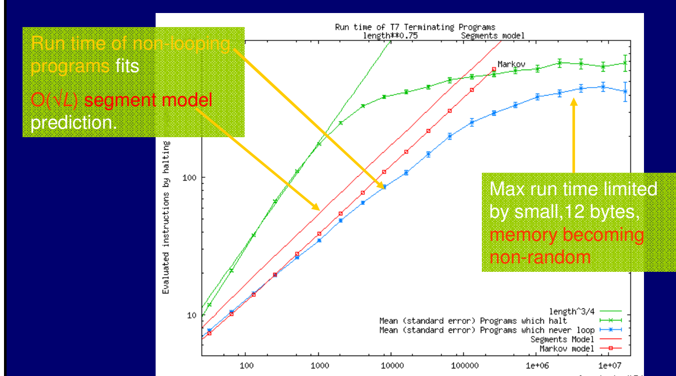
Proportion of programs without loops falls as 1/sqrt(length)



Average run time (non-looping)

- Segments model allows us to compute a bound for runtime
- Expected run time grows as $O(N^{½})$

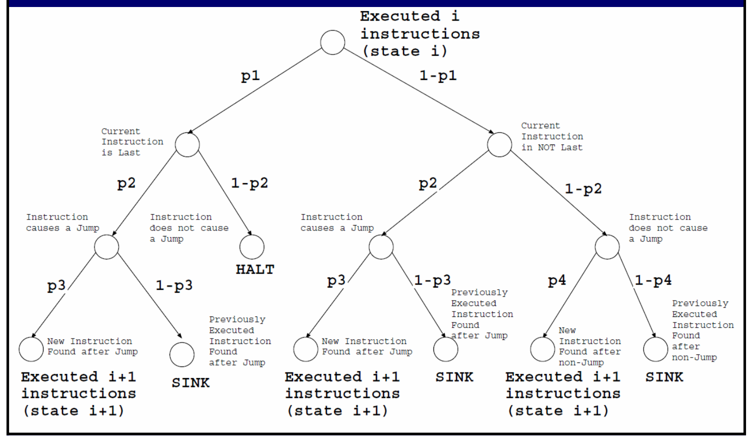
Run time on terminating programs



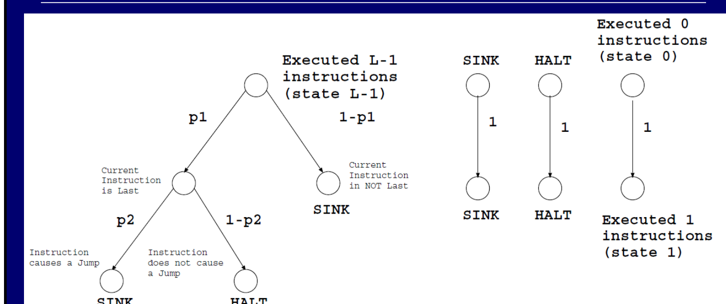
Markov model: States

- State 0 = no instructions executed, yet
- State i = i instructions but no loops have been executed
- Sink state = at least one loop was executed
- Halt state = the last instruction has been successfully executed and PC has gone beyond it.

Event diagram for program execution 1/2



Event diagram for program execution 2/2



p_1 = probability of being the last instruction

- Program execution starts from a random position
- Memory is randomly initialised and, so, any jumps land at random locations
- Then, the probability of being at the last instruction in a program is independent of how may (new) instructions have been executed so far.
- So, $$p_1 = \frac{1}{L}$$

p_2 = probability of instruction causing a jump

- We assume that we have two types of jumps
  - unconditional jumps (prob. p_{uj}, where PC is given a value retrieved from memory or from a register
  - conditional jumps (prob. p_{cj})
- Flag bit (which causes conditional jumps) is set with probability p_f
- The total probability that the current instruction will cause a jump is

$$p_2 = p_{uj} + p_{cj} \times p_f$$

For the CPU T7, we set $p_{uj} = \frac{1}{7}$, $p_{cj} = \frac{1}{7}$, and $p_f = \frac{1}{2}$, whereby $p_2 = \frac{3}{14}$

p_3= probability of new instruction after jump

- Program counter after a jump is a random number between 1 and L
- So, the probability of finding a new instruction is

$$p_3 = \frac{L-i}{L}$$

p_4 = probability of new instruction after non-jump

- The more jumps we have executed the more the map of visited instructions will be fragmented.
- So, we should expect p_4 to decrease as a function of the number of jumps/fragments.
- Expected number of fragments (jumps) in a program having reached state i

$$E[J] = i \times p_2 = i \times (p_{uj} + p_{cj} \times p_f)$$

In the case of T7 this gives us $E[J] = \frac{3i}{14}$

- Each block will be preceded by at least one unvisited instruction
- So, the probability of a previously executed instruction after a non-jump is

$$\frac{J}{L-i-1}$$

and

$$p_4 \approx 1 - \frac{J}{L-i-1} = \frac{L-E[J]-i-1}{L-i-1}$$

- A more precise model considers the probability of blocks being contiguous.
- Expected number of actual blocks

$$E[B] \approx \max\left(E[J] - \frac{E[J](1+E[J])}{2(L-i)}, 0\right)$$

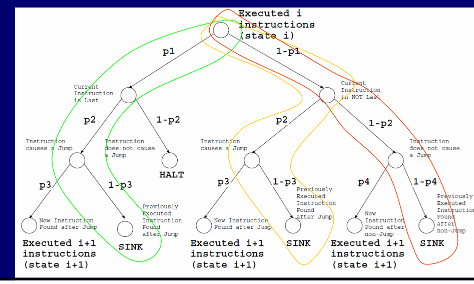
hence

$$p_4 = \frac{L - E[B] - i - 1}{L - i - 1}$$

Markov Model: state transition probabilities

- These are obtained by adding up "paths" in the program execution event diagram

E.g. looping probability



Less than L-1 instructions visited

$$p(i \to halt) = p_1(1-p_2) = \frac{1 - p_{uj} + p_{cj} \times p_f}{L}$$

For T7, $p(i \to halt) = \frac{11}{14L}$

$$p(i \to sink) = p_1p_2(1-p_3) + (1-p_1)p_2(1-p_3) + (1-p_1)(1-p_2)(1-p_4)$$

$$p(i \to i+1) = p_1p_2p_3 + (1-p_1)p_2p_3 + (1-p_1)(1-p_2)p_4$$

L-1 instructions visited

$$p(L-1 \to halt) = p_1(1-p_2)$$

$$p(L-1 \to sink) = p_1p_2 + (1-p_1)$$

Transition matrix

- For example, for T7 and L = 7 we obtain

$$
\begin{array}{cc}
M = \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0.8312 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.7647 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.6812 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0.566 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.3868 & 0 & 0 & 0 \\
0 & 0.05655 & 0.1231 & 0.2065 & 0.3217 & 0.501 & 0.8878 & 1 & 0 \\
0 & 0.1122 & 0.1122 & 0.1122 & 0.1122 & 0.1122 & 0.1122 & 0 & 1
\end{pmatrix}
&
\begin{array}{l}
\textit{0 instructions} \\
\textit{1 instructions} \\
\textit{2 instructions} \\
\textit{3 instructions} \\
\textit{4 instructions} \\
\textit{5 instructions} \\
\textit{6 instructions} \\
\textit{loop} \\
\textit{halt}
\end{array}
\end{array}
$$

0 instructions, 1 instructions, 2 instructions, 3 instructions, 4 instructions, 5 instructions, 6 instructions, loop, halt

Computing future state probabilities

- The distribution of future states can be computed bytaking appropriate powers of the Markov matrix M

$$p_{states} = M^i x$$

$$x = (1, 0, 0, \cdots, 0)^T$$

Examples

For T7, L=7 and i=3

$$p_{states} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0.6356 \\ 0 \\ 0 \\ 0 \\ 0.1589 \\ 0.2055 \end{pmatrix}$$

prob. looping in 3 instructions

prob. halting in 3 instructions

For T7, L=7 and i=L

total halting probability

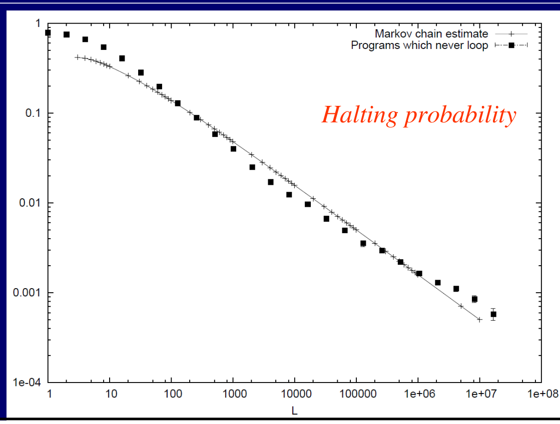
$$p_{states} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0.6364 \\ 0.3636 \end{pmatrix}$$

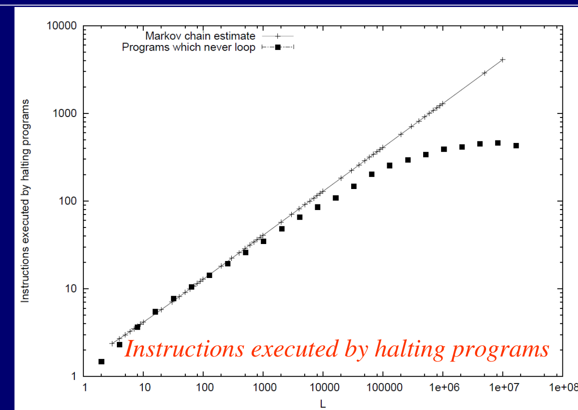
Efficiency

- Computing halting probabilities requires a potentially exponentially explosive computation to perform (M^L)
- We reordered calculations to obtain very efficient models which allow us to compute
  - halting probabilities and
  - expected number of instructions executed by halting programs

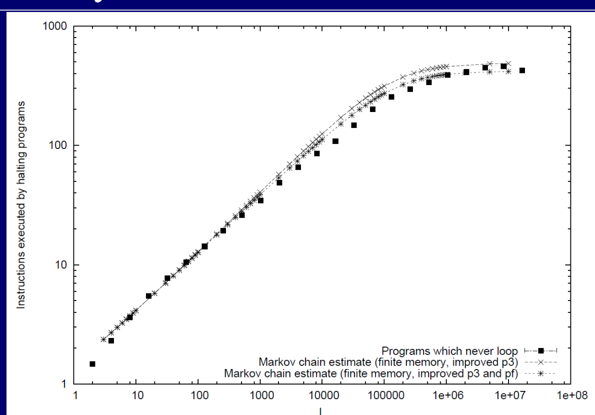
  for L = 10,000,000 or more (see paper for details)

A good model?

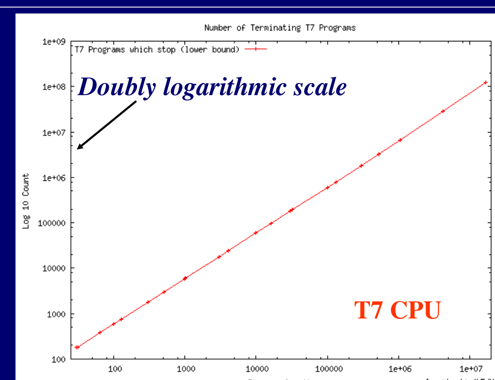




Improved model accounting for memory correlation



Number of halting programs rises exponentially with length



$10^{100\ 000\ 000}$

Turing complete GP cannot possibly work?

- Only halting programs can be solutions to problems, so

  |solution space|/|search space| < p(halt)

- In T7, p(halt) → 0, so,

  |solution space|/|search space| → 0

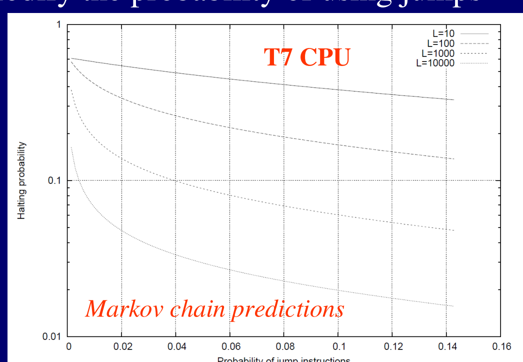
- Since the search space is immense, GP with T7 seems to have no hope of finding solutions.

What can we do?

- Control p(halt)
- Size population appropriately
- Design fitness functions which promote termination
- Repair
- ....
- Any mix of the above

Controlling *p(halt)*

- Modify the probability of using jumps



Population sizing

- Programs that do not terminate are given zero fitness
- So the effective population size in the initial generation is

  $$Popsize \times p(halt)$$

- For evolution to work we must have at least some halting individuals. So, we must choose

  Popsize >> 1 / *p(halt)*

  for the particular program length of interest.

Limiting distribution of functionality for halting programs?

- Non-looping programs halt
- The distribution of instructions in non-looping programs is the same as with a primitive set without jumps

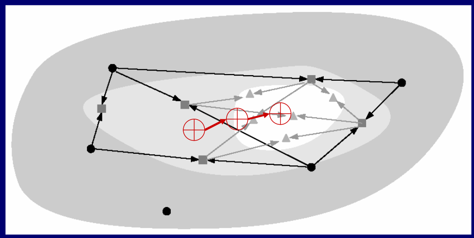
Limiting distribution of functionality for halting programs?

- So, as the number of instructions executed grows, the distribution of functionality of non-looping programs approaches a limit.
- Number of instructions executed, not program length, tells us how close the distribution is to the limit
- E.g. for T7, very long programs have a tiny subset of their instructions executed (e.g., 1,000 instructions in programs of L = 1,000,000).

GP Search Characterisation

GA and GP search

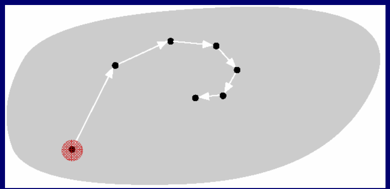
- GAs and GP search like this:



- How can we understand (characterise, study and predict) this search? Visualisation is not a solution
- Understanding = science → better systems

Microscopic Dynamical System Models

- We represent the population as a point in a multidimensional space, and we study the trajectory of this point

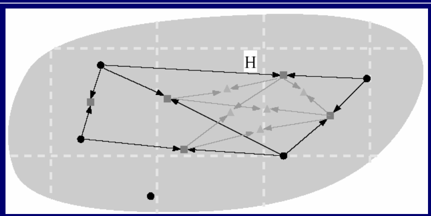


- This leads to exact models with huge numbers of parameters (microscopic models).

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*).

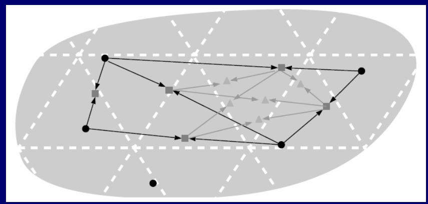
Example



- 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.

Schema Theorist's Questions

- **Q1:** How should the search space be divided? I.e. what is the right schema definition?
- E.g. how about



- There isn't a right schema definition: different definitions might be suitable for different purposes, algorithms, etc.
- Good definitions should:
  - have a simple syntactic representation (concise notation)
  - make the calculations doable.

- **Q2:** What are the right quantities one should use to describe schemata?
- We want quantities that lead to simple exact or reasonably accurate mathematical formulations.
- Also, we want macroscopic quantities (something equivalent to pressure, volume, mass, temperature, entropy, etc. in physics).

Traditionally, the following quantities have been used:

- number of individuals in a schema,
- average fitness of the individuals in the schema and in the population,
- size of the search space,
- size of the schema,
- "fragility" w.r.t. crossover and mutation, etc.

- **Q3:** What is the right schema theorem?
- EAs are non-deterministic, so exact predictions of the future state of the search cannot be made.
- However, the expected behaviour of an algorithm can be predicted using probability.
- Depending on the search space, on the schema definition and on the macroscopic quantities chosen, many different schema theorems have been obtained. They have different explanatory and predictive power.

Exact Schema Theorems

- The selection/crossover/mutation process is a Bernoulli trial: a newly created individual either samples or does not sample a schema H.
- 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

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

Pessimistic Schema Theorems

- Finding an exact formulation for $\alpha(H,t)$ can be very difficult
- Initially researchers have come up only with lower bounds which led to "pessimistic" schema theorems, i.e.

$$E[m(H,t+1)] \geq M\,\alpha_{min}(H,t)$$

GA Schema Theory

Holland's GA Schemata

- In GAs operating on binary strings, *syntactically* a schema is a string of symbols from the alphabet {0,1,*}, like *10*1.
- * is interpreted as a "don't care" symbol, so that, *semantically*, a schema represents a set of bit strings.
- E.g. *10*1 = {01001, 01011, 11001, 11011}

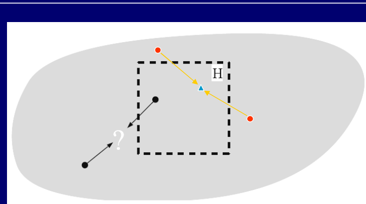
Holland’s Schema Theorem

- Holland's schema theory is approximate. It provides a lower bound for $\alpha(H,t)$ or, equivalently, for $E[m(H,t+1)]$.
- For one-point crossover and point mutation:

$$\alpha(H,t) \geq p(H,t)(1-p_m)^{O(H)}\left[1-p_c \times \frac{L(H)}{N-1} \times \sigma\right]$$

- $m(H,t)$ is number of individuals in the schema H at generation t,
- M is the population size,
- $p(H,t)$ is the selection probability for strings in H at generation t,
- p_m is the mutation probability,
- $O(H)$ is the schema order, i.e. number of defining bits,
- p_c is the crossover probability,
- $L(H)$ is the defining length, i.e. distance between the furthest defining bits in H,
- N is the bitstring length.

- Idea:

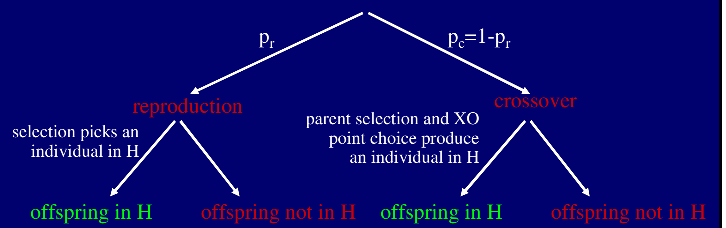


- Features:
  - The theorem includes an *expected value*
  - It provides a *lower bound*
  - So, it is difficult to make accurate predictions

- The factor σ differs in the different formulation of the schema theorem:
  - $\sigma=1-m(H,t)/M$ in (Holland, 1975),
  - $\sigma=1$ in (Goldberg, 1989),
  - $\sigma=1-p(H,t)$ in (Whitley, 1994).
- In 1997 Stephens and collaborators produced an exact formulation for $\alpha(H,t)$: an *“exact" schema theorem*.

How can we get an exact schema theorem?

- Let us assume that only reproduction and (one-offspring) crossover are performed.
- Creation probability tree for a schema *H*:



- Adding “paths” to success produces

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

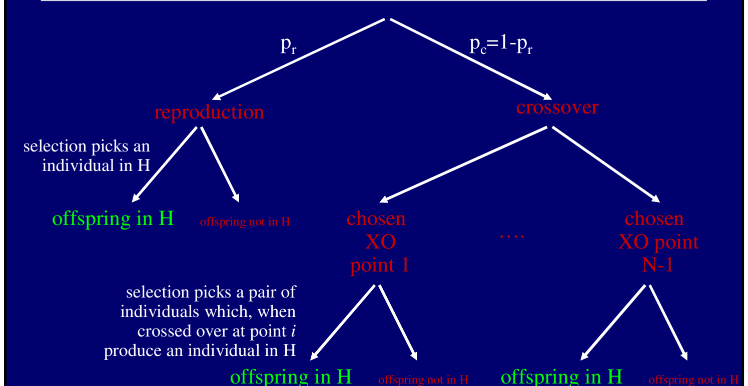
where $\Pr[\text{Selecting an individual in } H \text{ for cloning}] = p(H,t)$

- The process of crossover point selection is independent from the actual primitives in a parent.
- The probability of choosing a particular crossover depends only on the actual size of the parent.
- E.g., the probability of choosing any crossover point in

  1 1 0 1 0 1

  is identical to the probability of choosing any crossover point in

  0 0 0 1 1 0



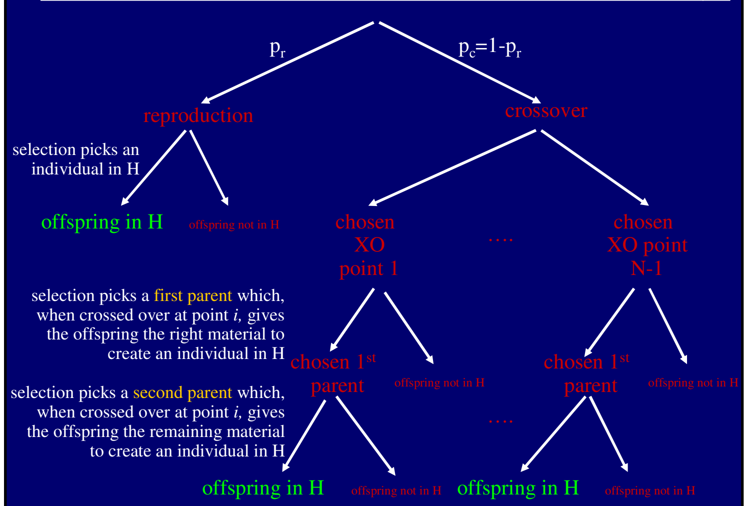
$$\Pr\left[\begin{array}{c}\text{The parents and the crossover points}\\ \text{are such that the offspring matches } H\end{array}\right]$$

$$= \sum_{\substack{\text{For all crossover}\\ \text{points } i}} \Pr\left[\begin{array}{c}\text{Choosing crossover}\\ \text{point } i\end{array}\right]$$

$$\times \Pr\left[\begin{array}{c}\text{Selecting parents such that if}\\ \text{crossed over at point } i \text{ produce an offspring in } H\end{array}\right]$$

- Let us assume that crossover points are selected with uniform probability:

$$\Pr\left[\begin{array}{c}\text{Choosing crossover}\\ \text{point } i\end{array}\right] = \frac{1}{\text{Number of bits} - 1}$$



$$\Pr\left[\begin{array}{c}\text{Selecting parents such that if}\\ \text{crossed over at point } i \text{ produce an offspring in } H\end{array}\right]$$

$$= \Pr\left[\begin{array}{c}\text{Selecting a first parent such that if crossed over at point } i\\ \text{provides the necessary material to create an offspring in } H\end{array}\right]$$

$$\times \Pr\left[\begin{array}{c}\text{Selecting a second parent such that if crossed over}\\ \text{at point } i \text{ provides the remaining necessary material}\end{array}\right]$$

Stephens and Waelbroeck's Exact GA Schema Theory (1997)

- For a binary GA with one point crossover applied with probability p_{xo} (and assuming p_m=0)

$$E[m(H,t+1)/M] = (1-p_{xo})p(H,t) + \frac{p_{xo}}{N-1}\sum_{i=1}^{N-1} p(L(H,i),t)p(R(H,i),t)$$

$$\Pr(L(H,i),t) = \Pr\left[\begin{array}{l}\text{Selecting a first parent such that if crossed over at point } i\\ \text{provides the necessary material to create an offspring in } H\end{array}\right]$$

$$\Pr(R(H,i),t) = \Pr\left[\begin{array}{l}\text{Selecting a second parent such that if crossed over}\\ \text{at point } i \text{ provides the remaining necessary material}\end{array}\right]$$

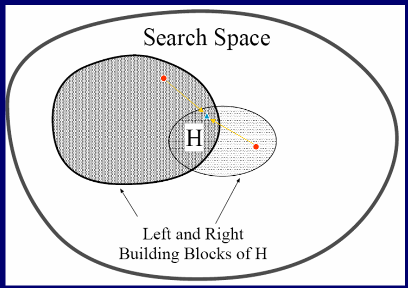
- *L(H,i)* is obtained by replacing the elements of *H* to the *right* of position *i* with "don't care" symbols
- *R(H,i)* is obtained by replacing the elements of *H* to the *left* of position *i*+1 with "don't care" symbols

- For example, if *H*=1*111, then *L(H,1)*=1****, *R(H,1)*=**111, *L(H,3)*=1*1**, *R(H,3)*=***11.
- For the schema *11, the theorem gives:

$$\begin{aligned} E[m(*11,t)/M] &= (1-p_{xo})p(*11,t) + \frac{p_{xo}}{2}(p(***,t)p(*11,t) + p(*1*,t)p(**1,t)) \\ &= (1-\frac{p_{xo}}{2})p(*11,t) + \frac{p_{xo}}{2}p(*1*,t)p(**1,t), \end{aligned}$$

since *p*(***,*t*)=1.

- In terms of sets:



- Note that the *L* and *R* building blocks are not necessarily all fitter than average, short or low-order.

Why should GPers be interested in GA theory?

- Bit strings and vectors of floating point parameters are graphs. E.g.

```
      010101  <—>  0 — 1 — 0 — 1 — 0 — 1
(0.12,0.4,3.1)  <—>  0.12 — 0.4 — 3.1
```

- So, evolutionary algorithms operating on fixed-length linear representations actually evolve graphs with a fixed linear topology.
- Different types of EAs use nodes from different primitive sets. E.g. in binary GAs $P=\{0,1\}$.

- GP, too, evolves special types of graphs, namely trees, but this time the topology is not necessarily fixed.
- Since linear graphs are special types of trees, in general *fixed-length linear EAs are special cases of some corresponding GP system* (more on this later).
- *So, in principle GA theory can be generalised to GP.*

Exact Schema Theory for GP with Subtree Crossover

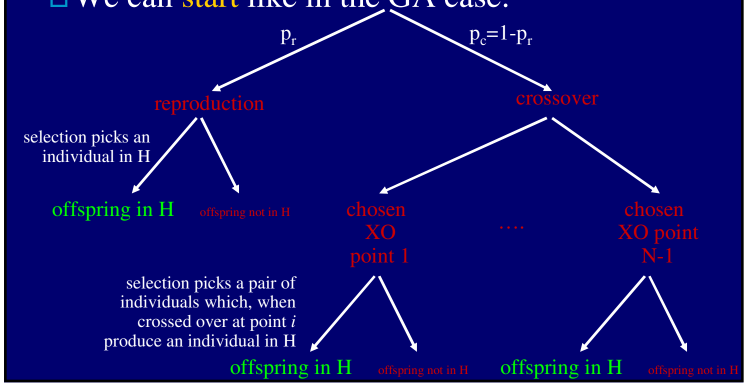
GP Schemata

- *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.
- For example, (= x (+ y =)) represents the set of programs

  {(+ x (+ y x)), (+ x (+ y y)), (* x (+ y x)), ...}

How can we get an exact schema theorem for GP with subtree crossover?

- We can start like in the GA case:



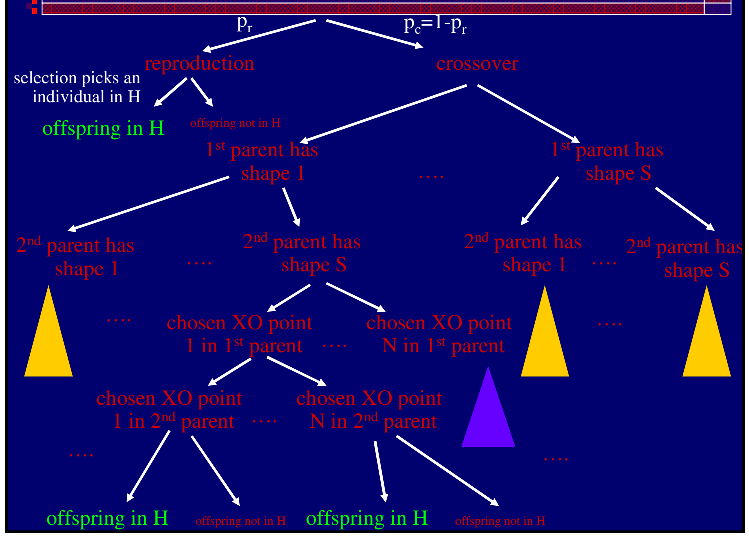
$$
\begin{aligned}
\alpha(H,t) &= \Pr[\text{An individual in } H \text{ is obtained via reproduction}] \\
&\quad + \Pr[\text{An offspring matching } H \text{ is produced by crossover}] \\
&= p_r \times \Pr[\text{An individual in } H \text{ is selected for cloning}] \\
&\quad + p_c \times \Pr\left[\begin{array}{l}\text{The parents and the crossover points} \\ \text{are such that the offspring matches } H\end{array}\right] \\
&= (1-p_c) \times p(H,t) \\
&\quad + p_c \times \Pr\left[\begin{array}{l}\text{The parents and the crossover points} \\ \text{are such that the offspring matches } H\end{array}\right]
\end{aligned}
$$

- 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))



$$\Pr\left[\begin{array}{c}\text{The parents and the crossover points}\\ \text{are such that the offspring matches } H\end{array}\right] = \sum_{\substack{\text{For all pairs of}\\ \text{parent shapes } k, l}} \sum_{\substack{\text{For all crossover}\\ \text{points } i, j \text{ in}\\ \text{shapes } k \text{ and } l}} \Pr\left[\begin{array}{c}\text{Choosing crossover points}\\ i \text{ and } j \text{ in shapes } k \text{ and } l\end{array}\right] \times \Pr\left[\begin{array}{c}\text{Selecting parents with shapes } k \text{ and } l \text{, such that if}\\ \text{crossed over at points } i \text{ and } j \text{ produce an offspring in } H\end{array}\right]$$

- Let us assume that crossover points are selected with uniform probability:

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

- The offspring has 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.

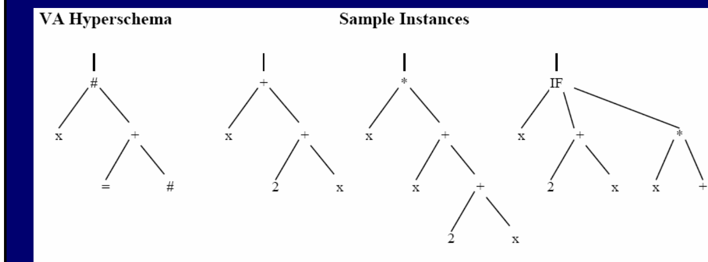
$$\Pr\left[\begin{array}{c}\text{Selecting parents with shapes } k \text{ and } l \text{, such that if}\\ \text{crossed over at points } i \text{ and } j \text{ produce an offspring in } H\end{array}\right] = \Pr\left[\begin{array}{c}\text{Selecting a root-donating parent with shape } k \text{ such that its upper}\\ \text{part w.r.t. crossover point } i \text{ matches the upper part of } H \text{ w.r.t. } i\end{array}\right] \times \Pr\left[\begin{array}{c}\text{Selecting a subtree-donating parent with shape } l \text{ such that its lower}\\ \text{part w.r.t. crossover point } j \text{ matches the lower part of } H \text{ w.r.t. } i\end{array}\right]$$

- Computing these two probabilities requires the introduction of a new concept: the variable arity hyperschema

Variable Arity Hyperschemata

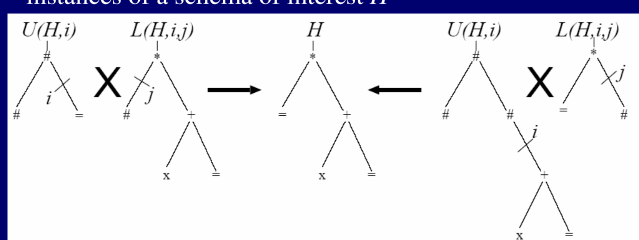
- A *GP variable arity hyperschema* is a tree with internal nodes from $F \cup \{=, \#\}$ and leaves from $T \cup \{=, \#\}$.
- = is a "don't care" symbols which stands for exactly one node, # terminal stands for any valid subtree, while the # function stands for exactly one node of arity not smaller than the number of subtrees connected to it.

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



Upper and lower building blocks

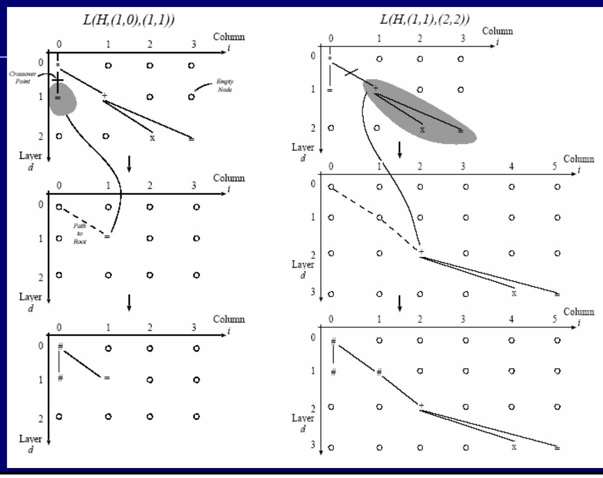
- VA hyperschemata can express which parents can produce instances of a schema of interest *H*



- If one crosses over at point *j* any individual matching *L(H,i,j)* and at point *i* any individual matching *U(H,i)*, the resulting offspring is always an instance of *H*

- *U(H, i)* is the VA hyperschema obtained from *H* by replacing the subtree below crossover point *i* with a # node.
- *L(H, I, j)* is the VA hyperschema obtained by:
  - rooting at coordinate *j* in an empty reference system the subschema of *H* below crossover point *i*,
  - then by labelling all the nodes on the path between node *j* and the root node with # function nodes, and
  - labelling the arguments of those nodes which are to the left of such a path with # terminal nodes

- For example,



Exact GP Schema Theorem for Subtree Crossover (2001)

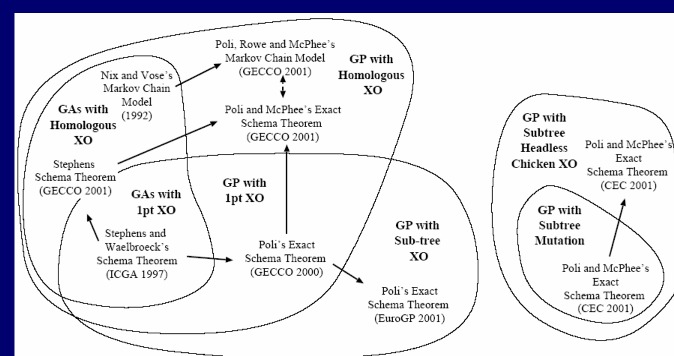
- Schema theorem for standard GP crossover

$$E[m(H,t+1)/M] = (1-p_{xo})p(H,t)+$$
$$p_{xo}\sum_{k,l}\frac{1}{N(G_k)N(G_l)}$$
$$\sum_{i\in H\cap G_k}\sum_{j\in G_l} p(U(H,i)\cap G_k,t)p(L(H,i,j)\cap G_l,t)$$

$$p(U(H,i)\cap G_k,t) = \Pr\left\{\begin{array}{l}\text{Selecting a root-donating parent}\\ \text{with shape } k \text{ such that its upper}\\ \text{part w.r.t. crossover point } i \text{ matches}\\ \text{the upper part of } H \text{ w.r.t. } i\end{array}\right\}$$

$$p(L(H,i,j)\cap G_l,t) = \Pr\left\{\begin{array}{l}\text{Selecting a subtree-donating parent}\\ \text{with shape } l \text{ such that its lower part}\\ \text{w.r.t. crossover point } j \text{ matches the}\\ \text{lower part of } H \text{ w.r.t. } i\end{array}\right\}$$

Exact Schema Theories for Different Operators and their Relations



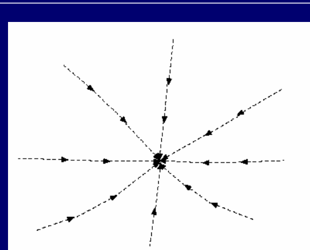
So what?

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

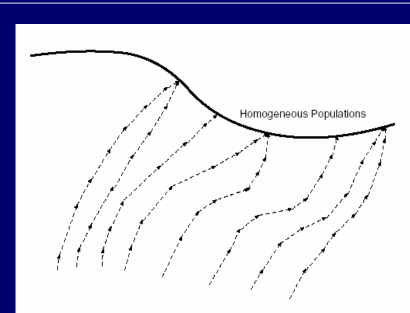
Lessons

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

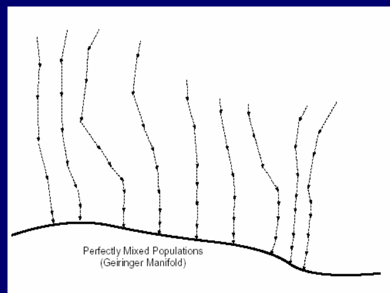
Mutation Bias



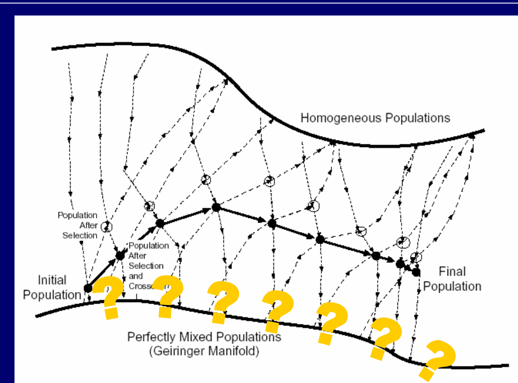
Selection Bias



Crossover Bias



So where is evolution going?



Geringer's manifold for linear GP

- A fixed-point distribution for the proportion of a linear, variable-length schema $h_1 h_2 \ldots h_N$ under a **GP homologous crossover** for an infinite population on a flat fitness landscape is

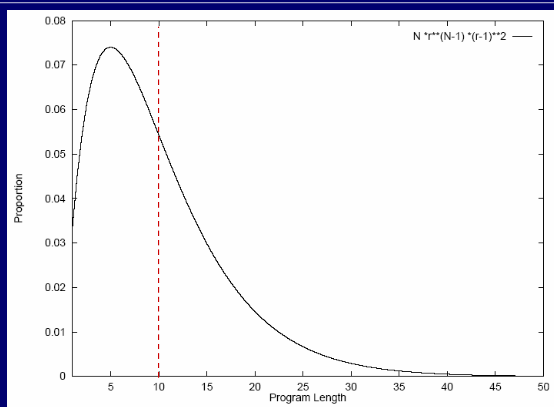
$$\lim_{t\to\infty} \Phi(h_1 h_2 \cdots h_N, t) = \Phi(*^{N-1} h_N, 0) \prod_{i=1}^{N-1} \frac{\Phi(*^{i-1} h_i \#, 0)}{\Phi(*^i \#, 0)}$$

- This is a unique global attractor for the system.

- A fixed-point distribution for program length under **GP subtree crossover** for an infinite population on a flat fitness landscape is a Gamma distribution:

$$\Phi((=)^N, t) = N r^{N-1} (r-1)^2$$

- For example the fixed-point length distribution for $\mu = 10$ is



Unequal Search Space Sampling

- The average probability that each program of length x will be sampled by standard crossover is

$$p_{\text{sample}}(x) = costant \times x r^{x-1} / |\mathcal{F}|^{x-1}$$

- For a flat landscape, standard GP will sample a particular short program much more often than it will sample a particular long one.

Allele Diffusion

- A fixed-point distribution for the proportion of a linear, variable-length schema $h_1 h_2 \dots h_N$ under GP subtree crossover for an infinite population initialised at the fixed-point length distribution is

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

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

- Crossover attempts to push the population towards distributions of primitives where each primitive 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 fully 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

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

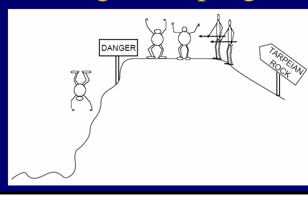
- Or equivalently

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

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

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



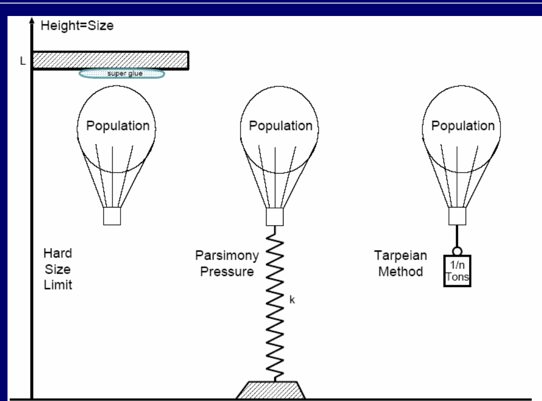
The Tarpeian method to control bloat

- Tarpeian fitness-wrapper

```
IF size(program) > average_pop_size AND random_int MOD n = 0
THEN
    return( very_low_fitness );
ELSE
    return( fitness(program) );
```

- The Tarpeian method drastically decreases the selection probability of longer-than-average programs creating a sort of fitness hole that discourages growth

Hot Air Balloon Metaphor



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.

- Different *operators* lead to different schema theorems, but we have started integrating them into a single coherent theory.
- The theory of GP is more general than the corresponding GA theory → unification by inclusion

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