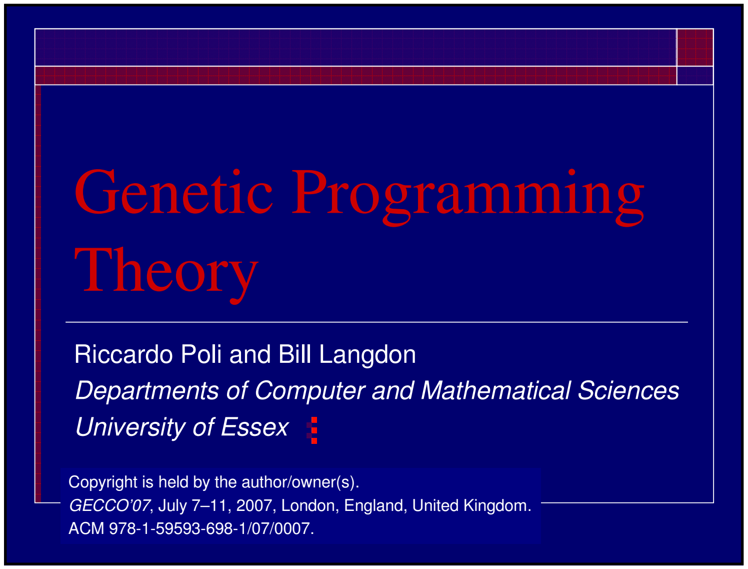


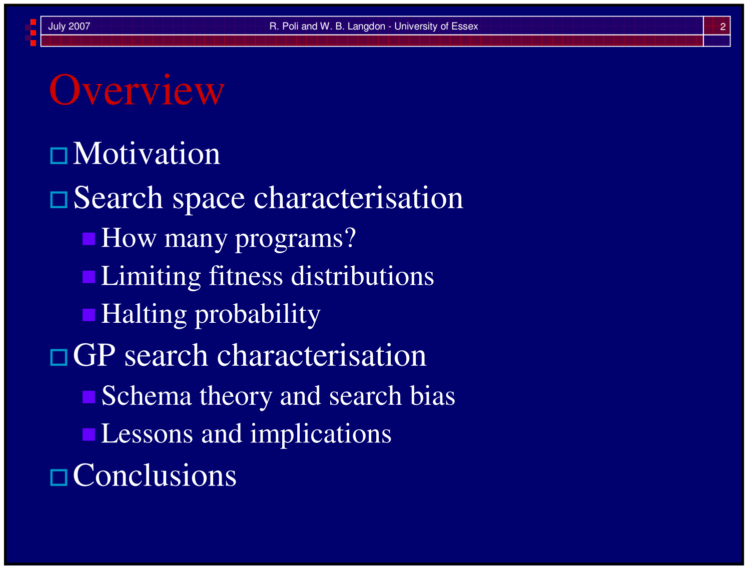
Genetic Programming Theory

Riccardo Poli and Bill Langdon

Departments of Computer and Mathematical Sciences

University of Essex

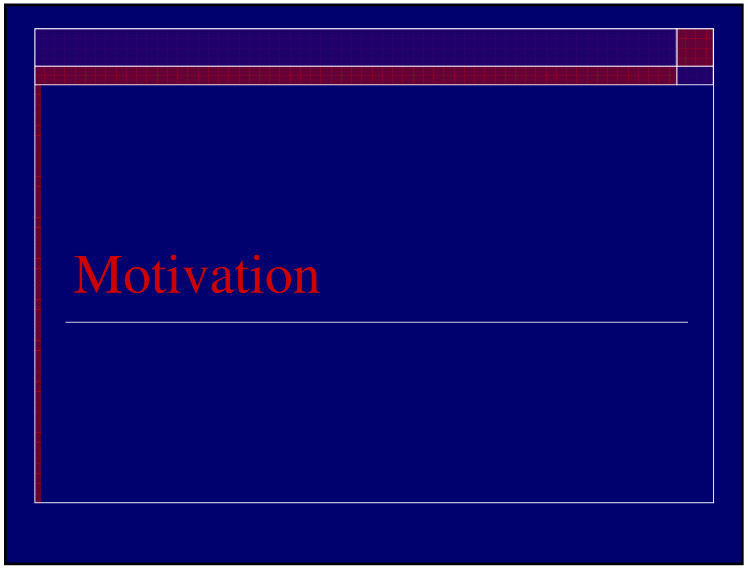
GECCO'07, July 7–11, 2007, London, England, United Kingdom.
ACM 978-1-59593-698-1/07/0007.



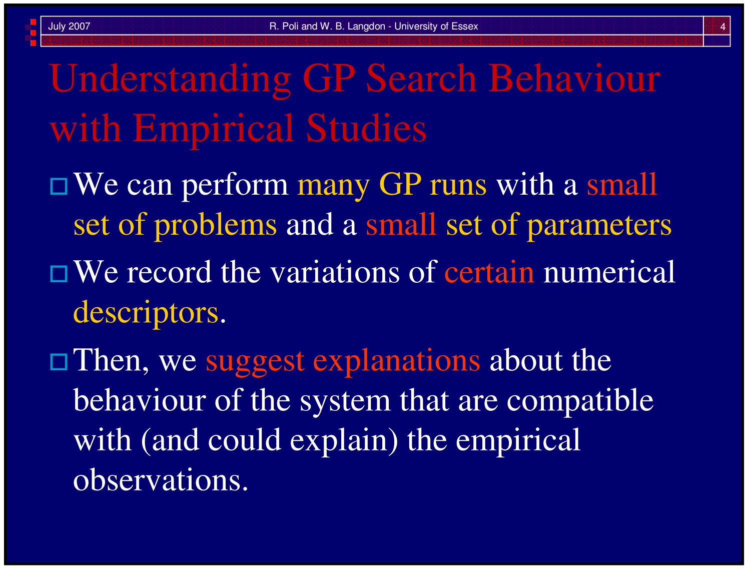
July 2007 R. Poli and W. B. Langdon - University of Essex 2

Overview

- □ Motivation
- □ Search space characterisation
  - How many programs?
  - Limiting fitness distributions
  - Halting probability
- □ GP search characterisation
  - Schema theory and search bias
  - Lessons and implications
- □ Conclusions



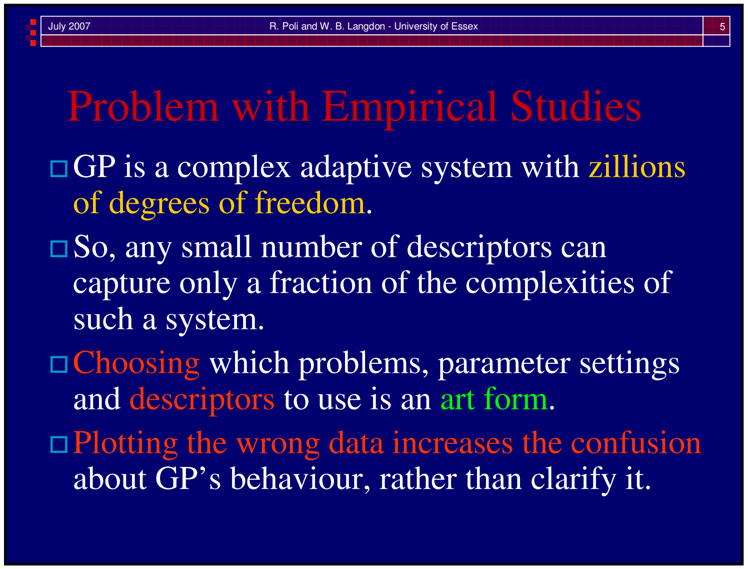
Motivation



July 2007 R. Poli and W. B. Langdon - University of Essex 4

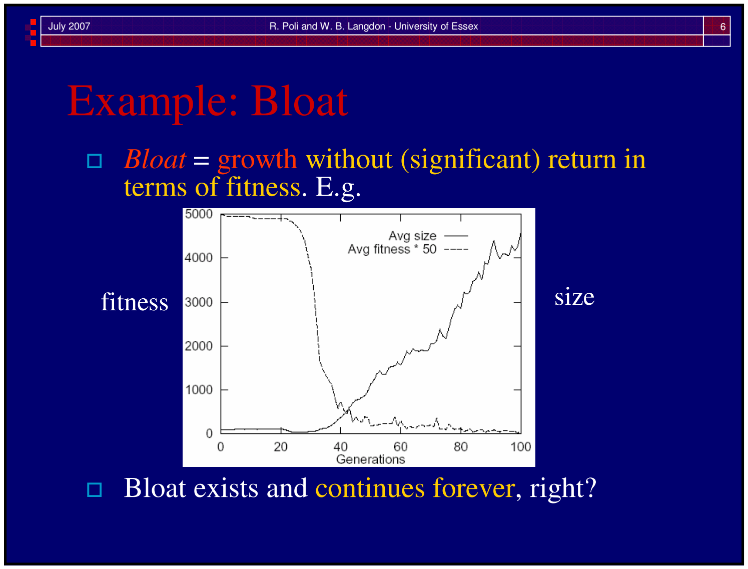
Understanding GP Search Behaviour with Empirical Studies

- □ 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

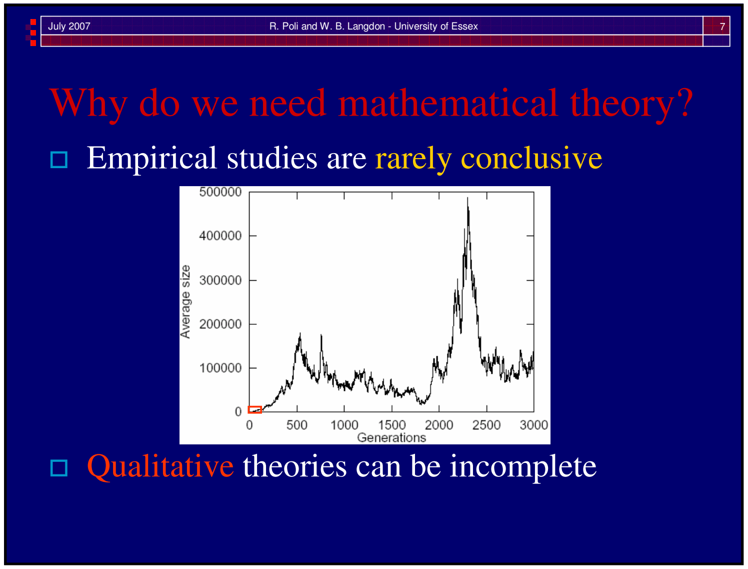
- □ GP is a complex adaptive system with zillions of degrees of freedom.
- □ So, any small number of descriptors can capture only a fraction of the complexities of such a system.
- □ Choosing which problems, parameter settings and descriptors to use is an art form.
- □ Plotting the wrong data increases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



Example: Bloat

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

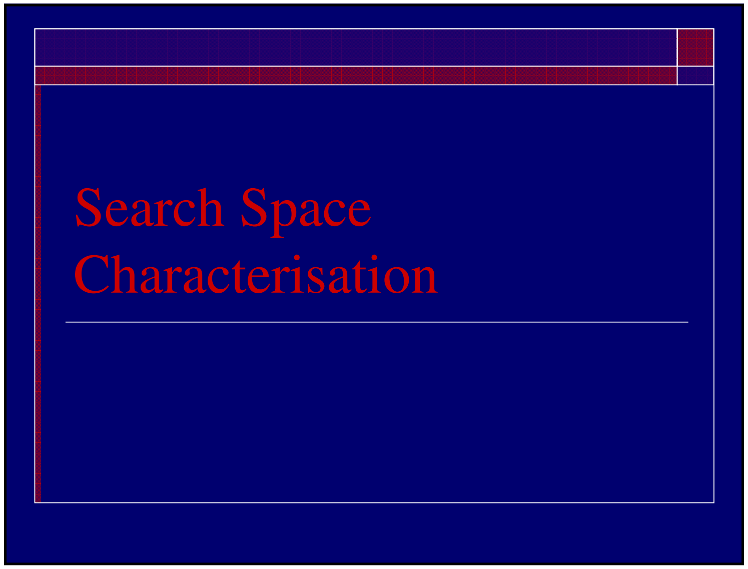
- □ Bloat exists and continues forever, right?



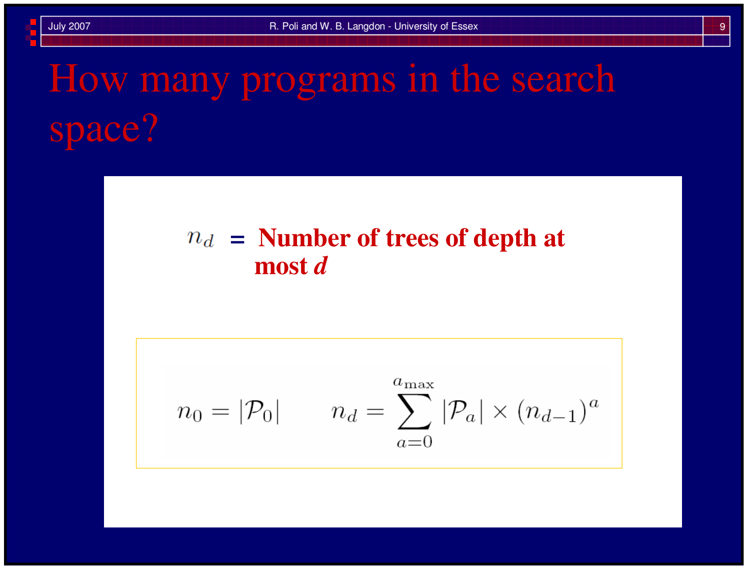
Why do we need mathematical theory?

- □ Empirical studies are rarely conclusive

- □ Qualitative theories can be incomplete



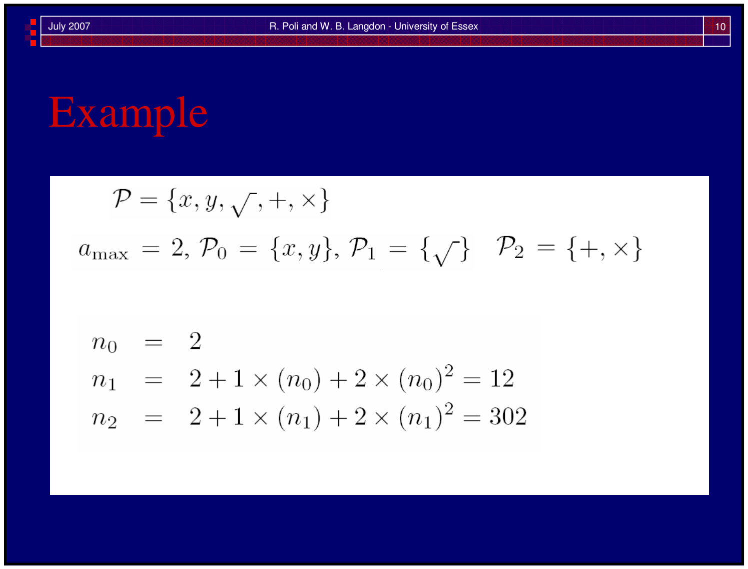
Search Space Characterisation



How many programs in the search space?

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

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

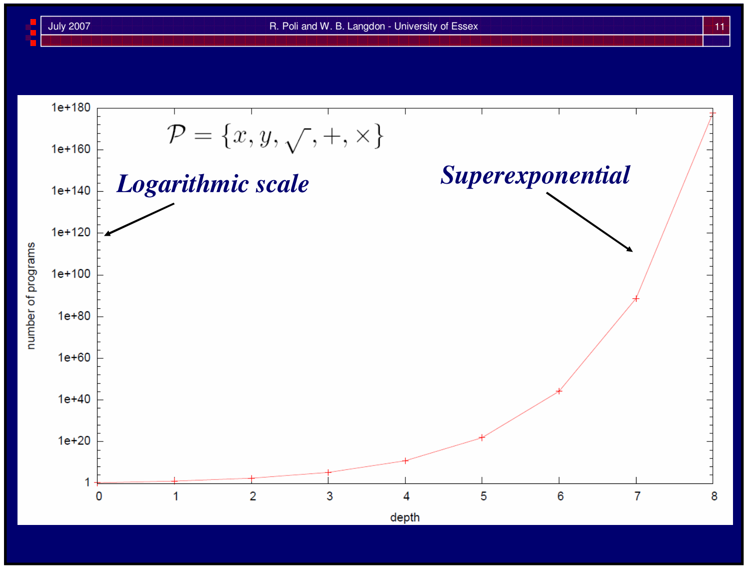


Example

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

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

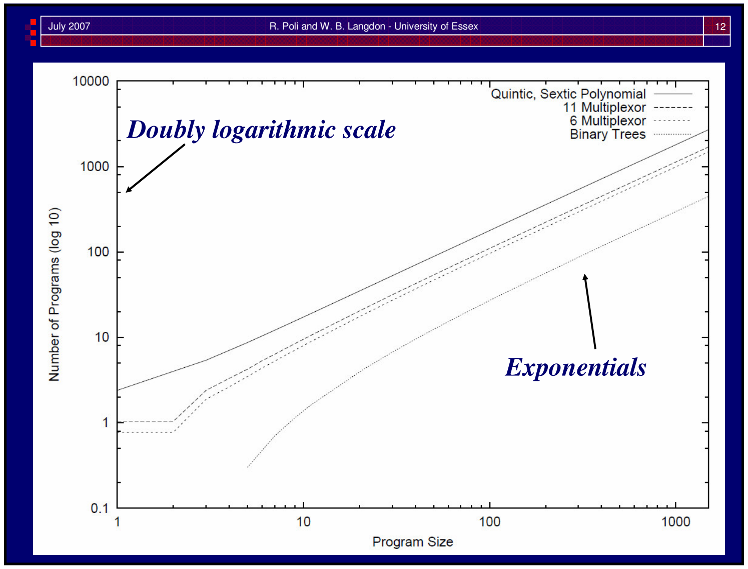
$$\begin{aligned} n_0 &= 2 \\ n_1 &= 2 + 1 \times (n_0) + 2 \times (n_0)^2 = 12 \\ n_2 &= 2 + 1 \times (n_1) + 2 \times (n_1)^2 = 302 \end{aligned}$$



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

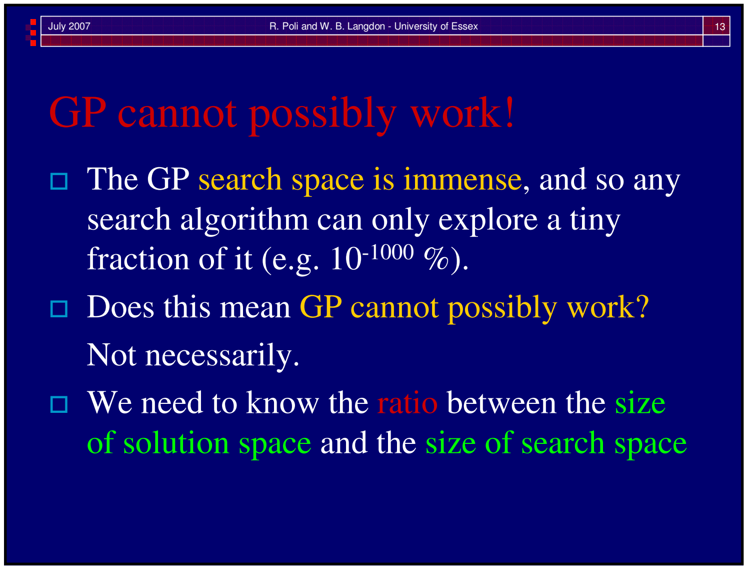
Logarithmic scale

Superexponential



Doubly logarithmic scale

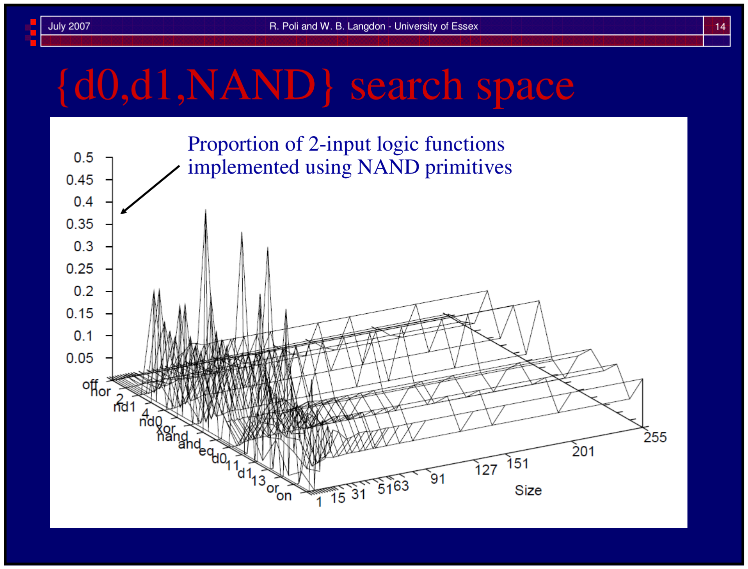
Exponentials



GP cannot possibly work!

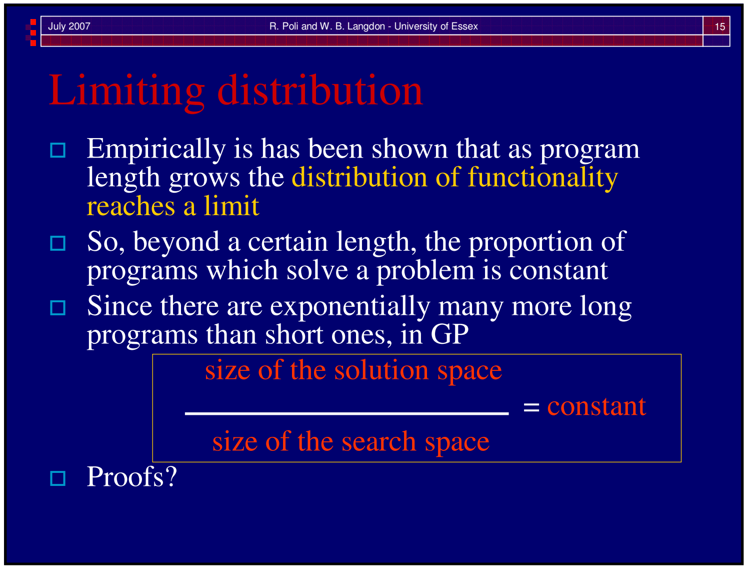
- The GP search space is immense, and so any search algorithm can only explore a tiny fraction of it (e.g. 10^{-1000} %).
- Does this mean GP cannot possibly work?

  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

Proportion of 2-input logic functions implemented using NAND primitives

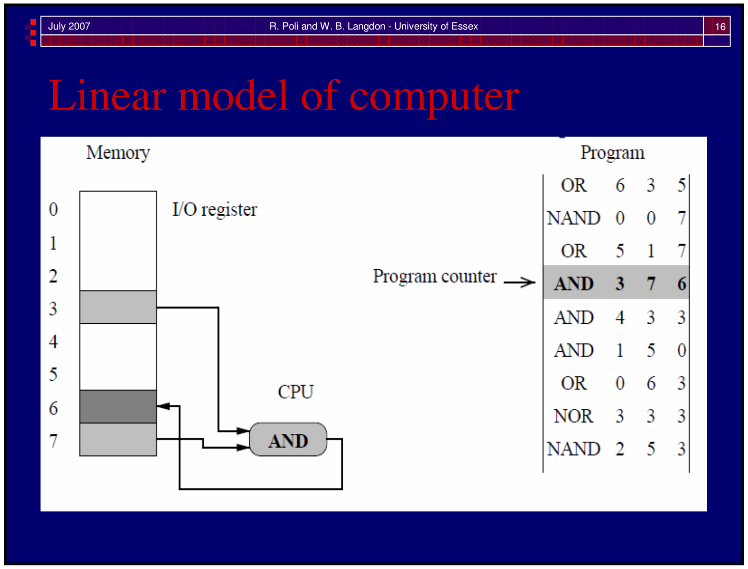


Limiting distribution

- 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

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

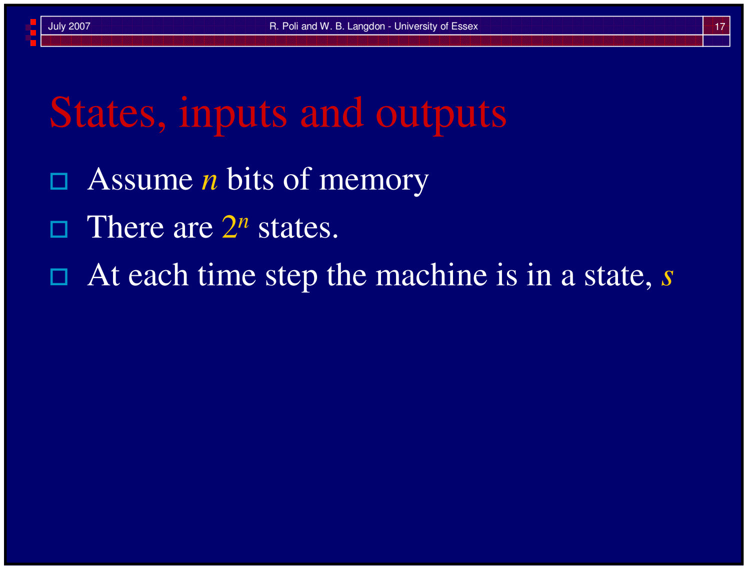
- Proofs?



Linear model of computer

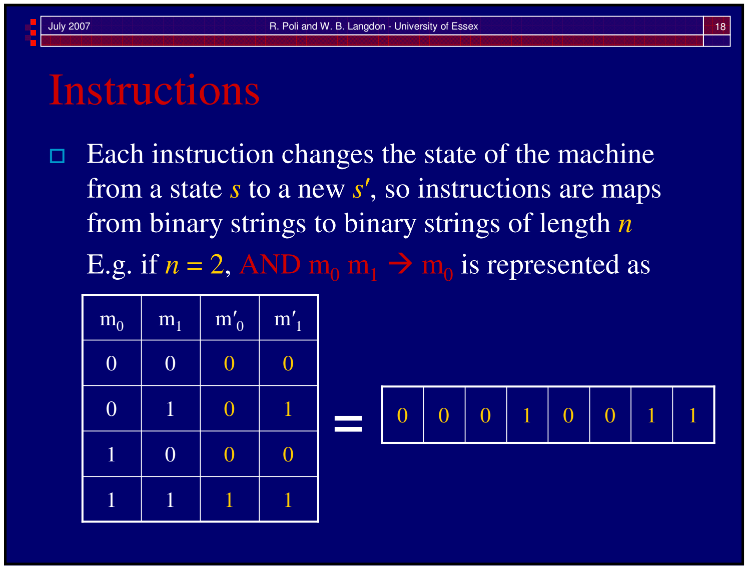
Program

OR	6	3	5
NAND	0	0	7
OR	5	1	7
AND	**3**	**7**	**6**
AND	4	3	3
AND	1	5	0
OR	0	6	3
NOR	3	3	3
NAND	2	5	3



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

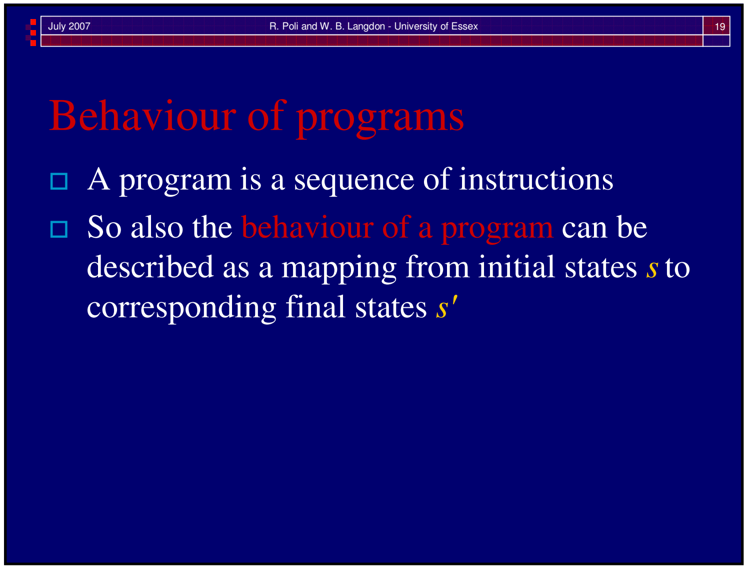
- Each instruction changes the state of the machine from a state s to a new s', so instructions are maps from binary strings to binary strings of length n

  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

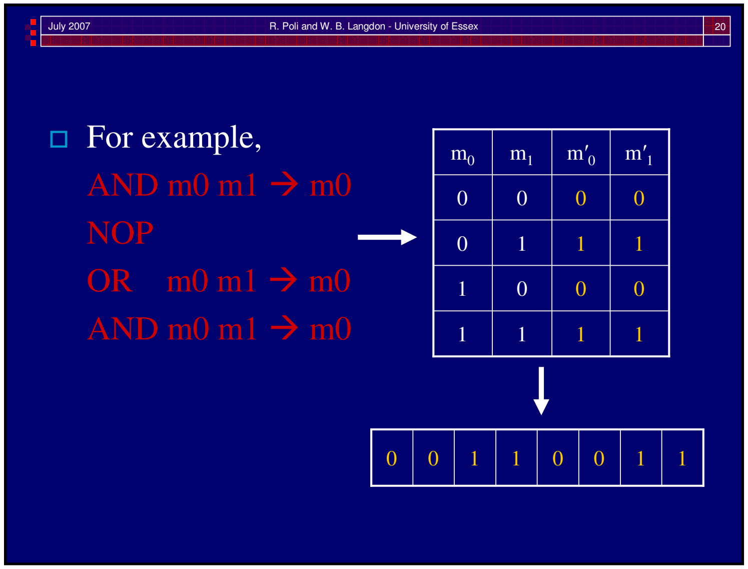
=

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

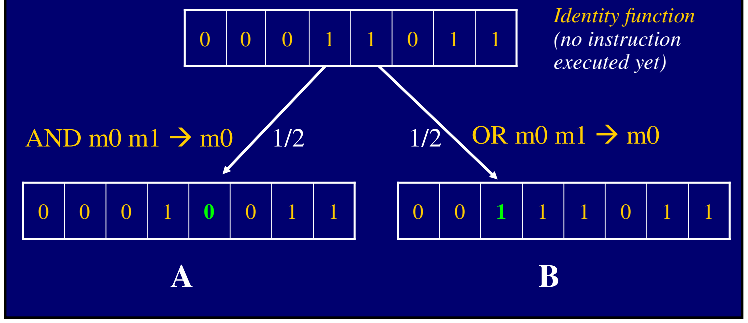
  AND m0 m1 → m0

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

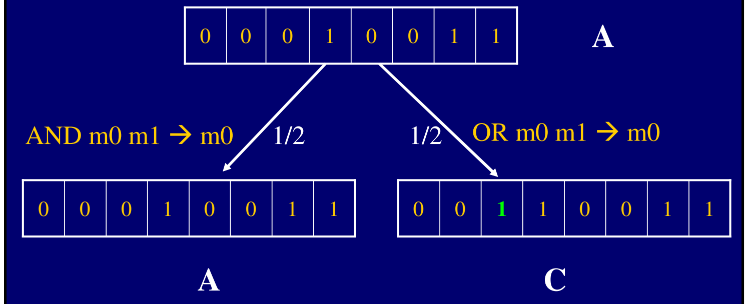
1/2 OR m0 m1 → m0

0	0	0	1	0	0	1	1

A

0	0	1	1	1	0	1	1

B



0	0	0	1	0	0	1	1

A

AND m0 m1 → m0 1/2

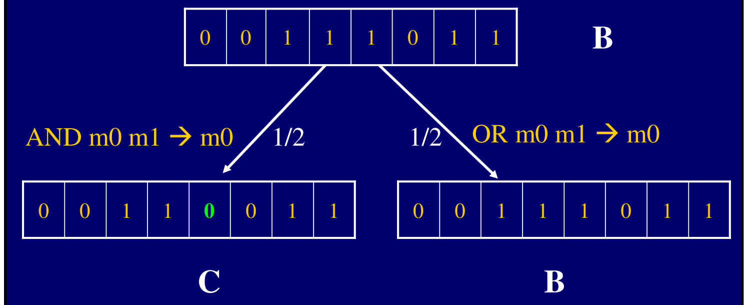
1/2 OR m0 m1 → m0

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

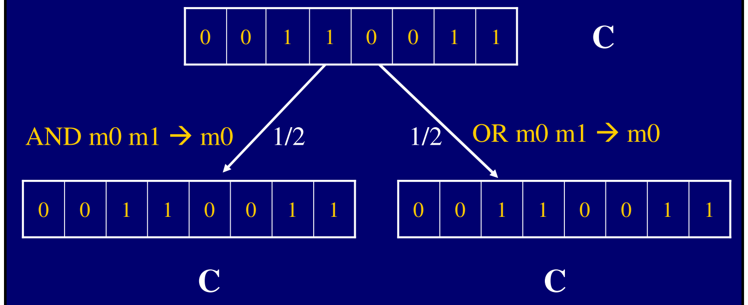
1/2 OR m0 m1 → m0

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

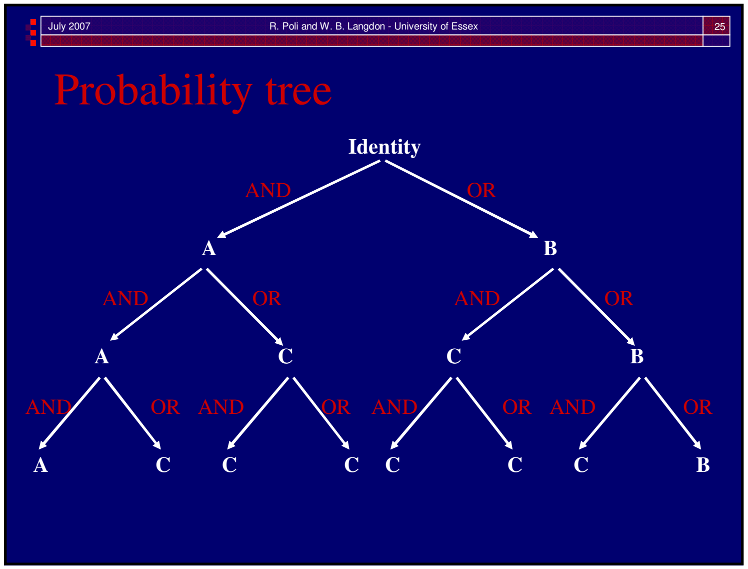
1/2 OR m0 m1 → m0

0	0	1	1	0	0	1	1

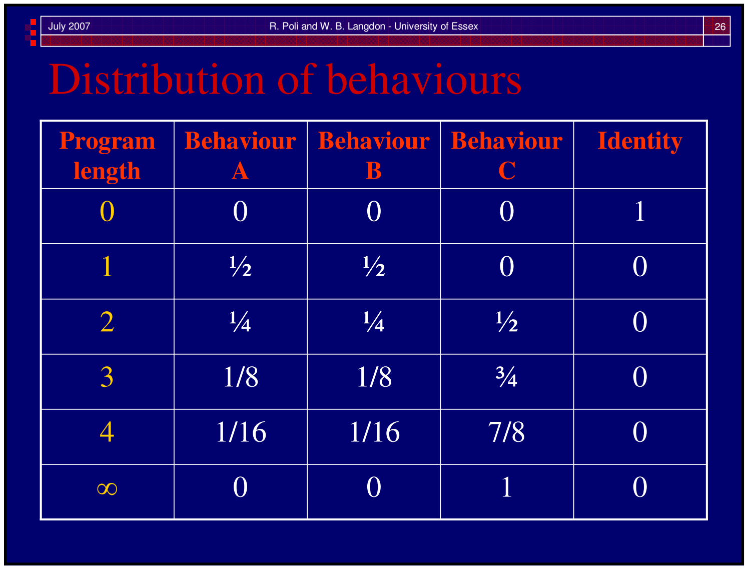
C

0	0	1	1	0	0	1	1

C

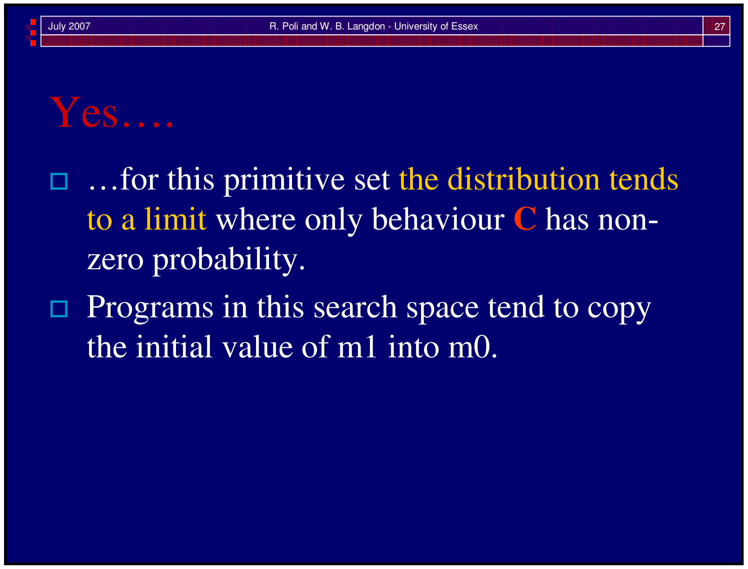


Probability tree



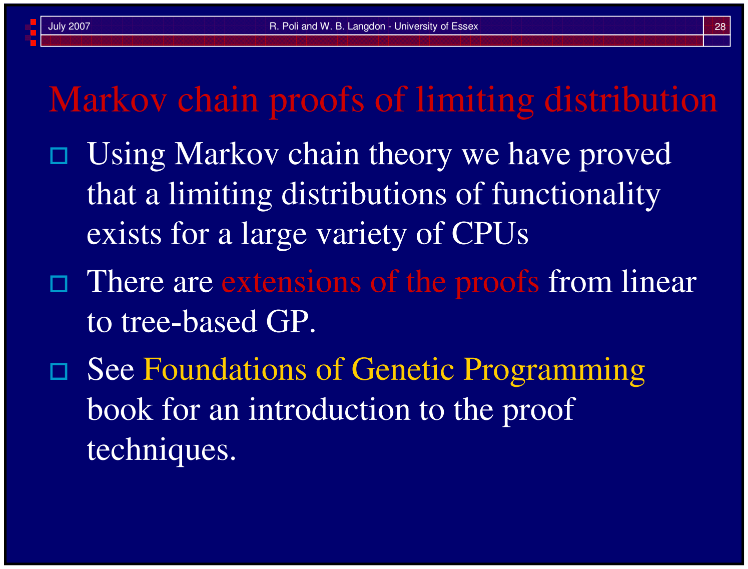
Distribution of behaviours

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



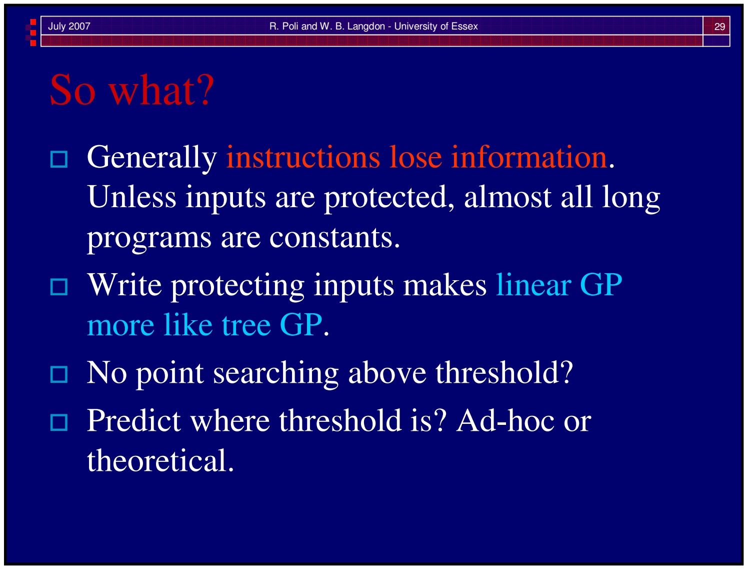
Yes….

- …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.



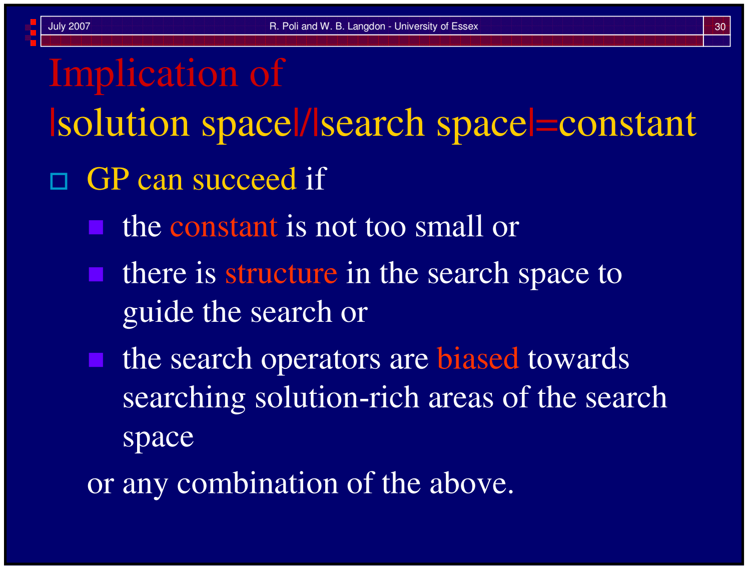
Markov chain proofs of limiting distribution

- 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?

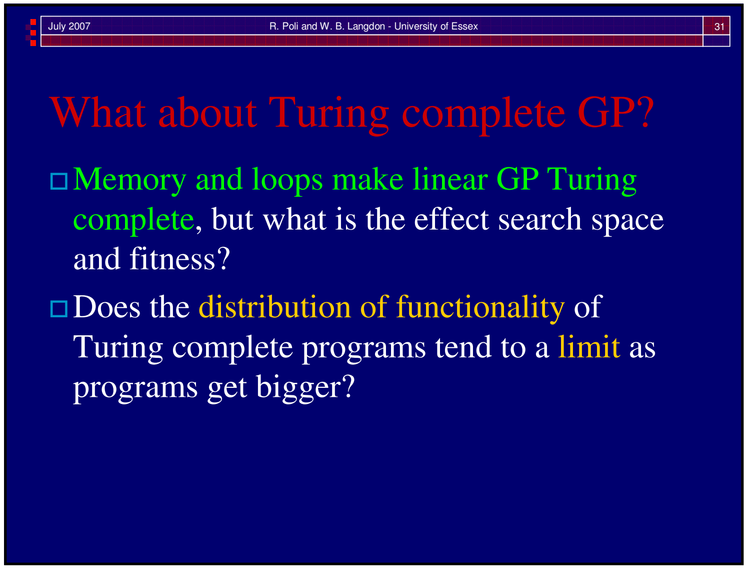
- □ Generally instructions lose information. Unless inputs are protected, almost all long programs are constants.
- □ Write protecting inputs makes linear GP more like tree GP.
- □ No point searching above threshold?
- □ Predict where threshold is? Ad-hoc or theoretical.



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

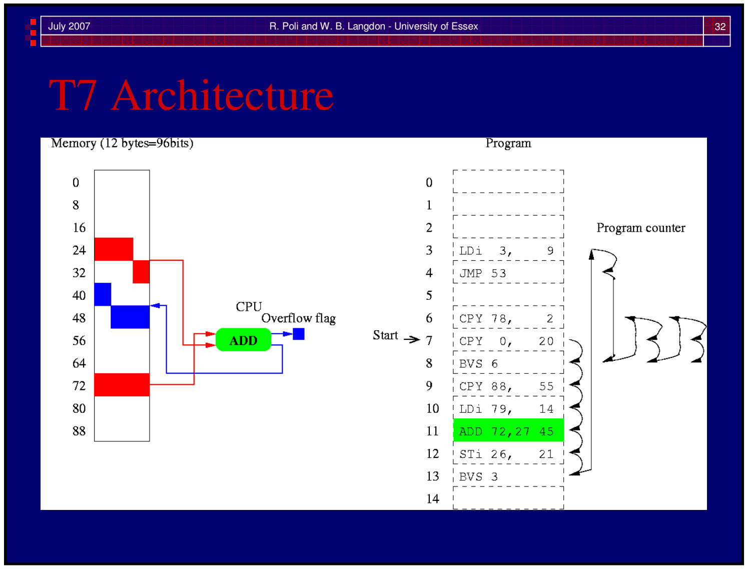
- □ GP can succeed if
  - the constant is not too small or
  - there is structure in the search space to guide the search or
  - the search operators are biased towards searching solution-rich areas of the search space

or any combination of the above.



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?



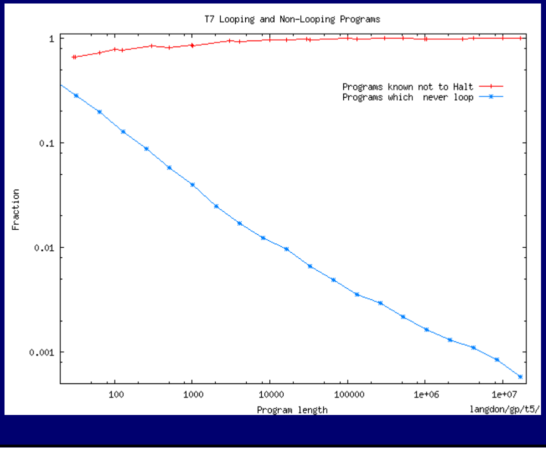
T7 Architecture

```
0
1
2
3   LDi  3,    9
4   JMP 53
5
6   CPY 78,    2
7   CPY  0,   20
8   BVS 6
9   CPY 88,   55
10  LDi 79,   14
11  ADD 72,27 45
12  STi 26,   21
13  BVS 3
14
```

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?

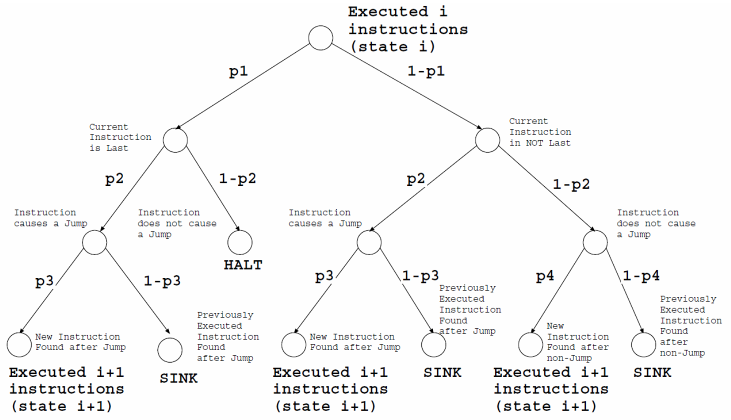
Almost all T7 Programs Loop

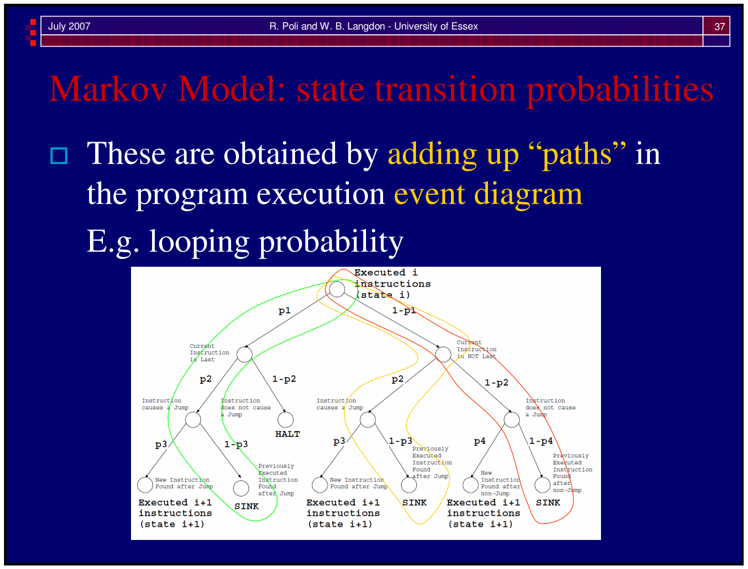


Markov model: States

- 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


Executed i instructions (state i)
p1
1-p1
Current Instruction is Last
Current Instruction in NOT Last
p2
1-p2
p2
1-p2
Instruction causes a Jump
Instruction does not cause a Jump
HALT
Instruction causes a Jump
Instruction does not cause a Jump
p3
1-p3
p3
1-p3
p4
1-p4
New Instruction Found after Jump
Previously Executed Instruction Found after Jump
New Instruction Found after Jump
Previously Executed Instruction Found after Jump
New Instruction Found after non-Jump
Previously Executed Instruction Found after non-Jump
Executed i+1 instructions (state i+1)
SINK
Executed i+1 instructions (state i+1)
SINK
Executed i+1 instructions (state i+1)
SINK



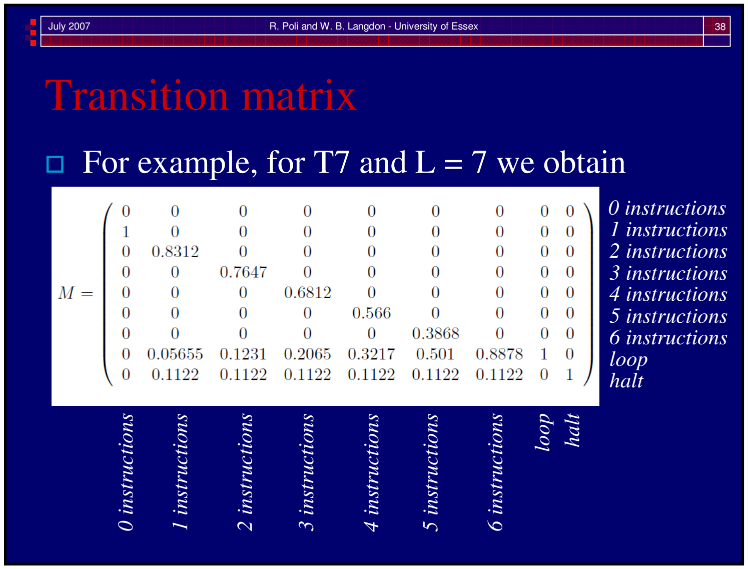
July 2007 R. Poli and W. B. Langdon - University of Essex 37

Markov Model: state transition probabilities

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

  E.g. looping probability

Executed i instructions (state i)
p1
1-p1
Current Instruction is Last
Current Instruction in NOT Last
p2
1-p2
p2
1-p2
Instruction causes a Jump
Instruction does not cause a Jump
HALT
Instruction causes a Jump
Instruction does not cause a Jump
p3
1-p3
p3
1-p3
p4
1-p4
New Instruction Found after Jump
Previously Executed Instruction Found after Jump
New Instruction Found after Jump
Previously Executed Instruction Found after Jump
New Instruction Found after non-Jump
Previously Executed Instruction Found after non-Jump
Executed i+1 instructions (state i+1)
SINK
Executed i+1 instructions (state i+1)
SINK
Executed i+1 instructions (state i+1)
SINK

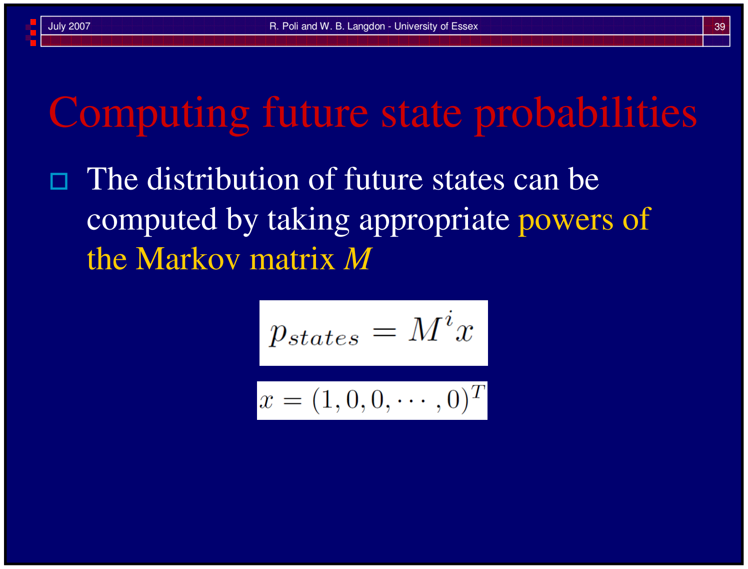


July 2007 R. Poli and W. B. Langdon - University of Essex 38

Transition matrix

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

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



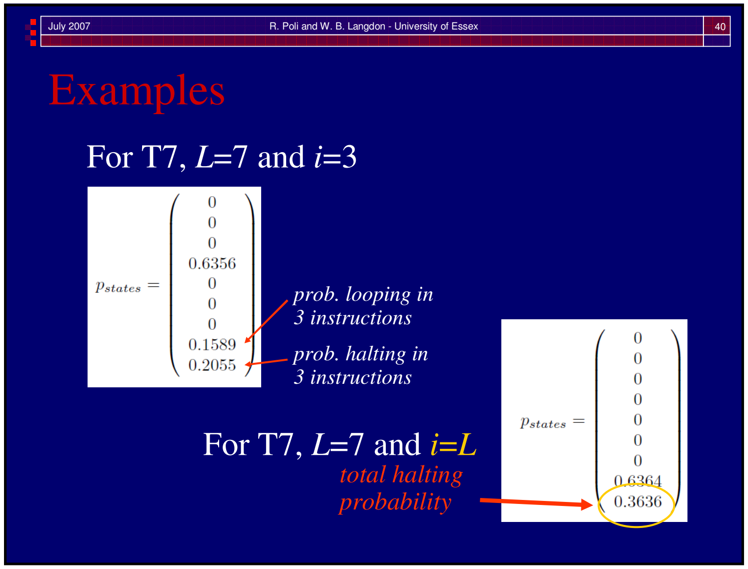
July 2007 R. Poli and W. B. Langdon - University of Essex 39

Computing future state probabilities

- The distribution of future states can be computed by taking appropriate powers of the Markov matrix *M*

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

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



July 2007 R. Poli and W. B. Langdon - University of Essex 40

Examples

For T7, *L*=7 and *i*=3

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

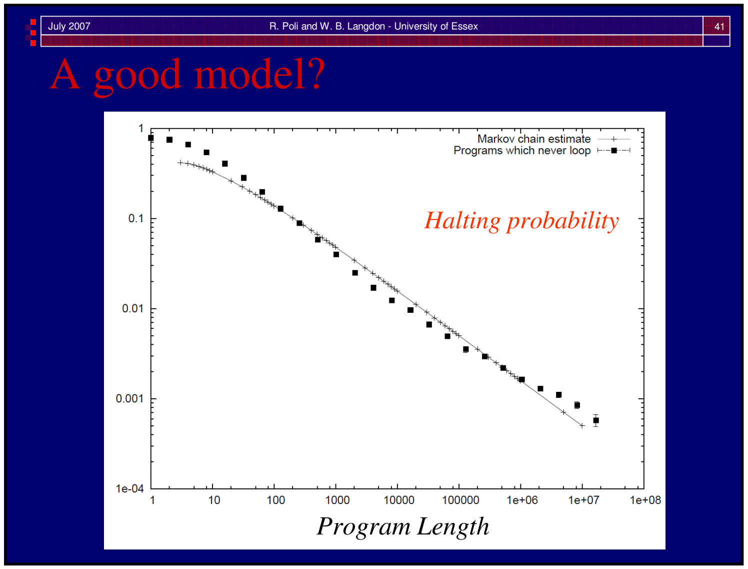
prob. looping in 3 instructions

prob. halting in 3 instructions

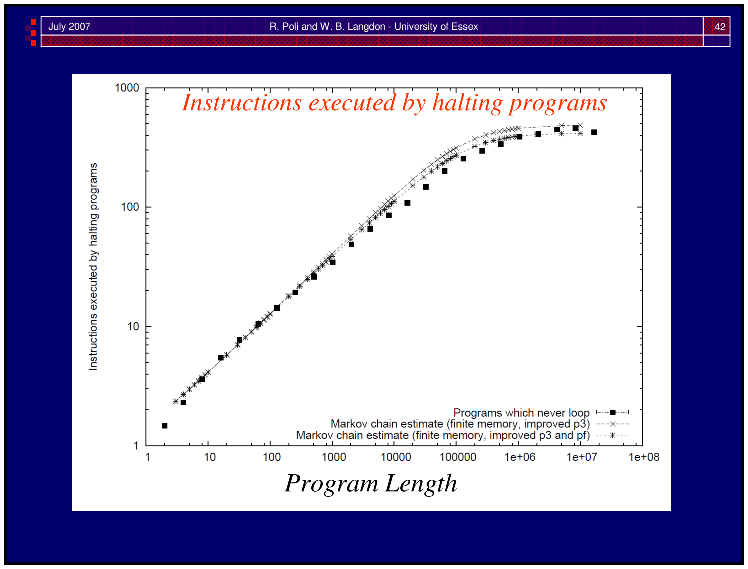
For T7, *L*=7 and *i*=*L*

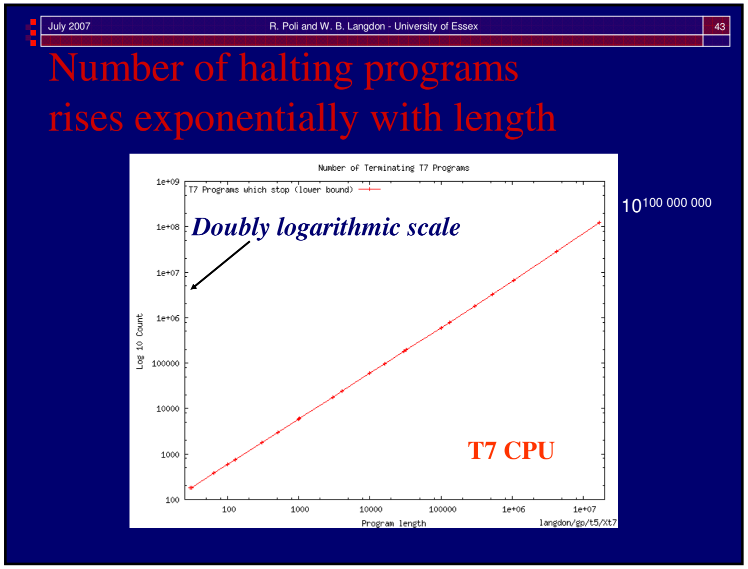
total halting probability

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



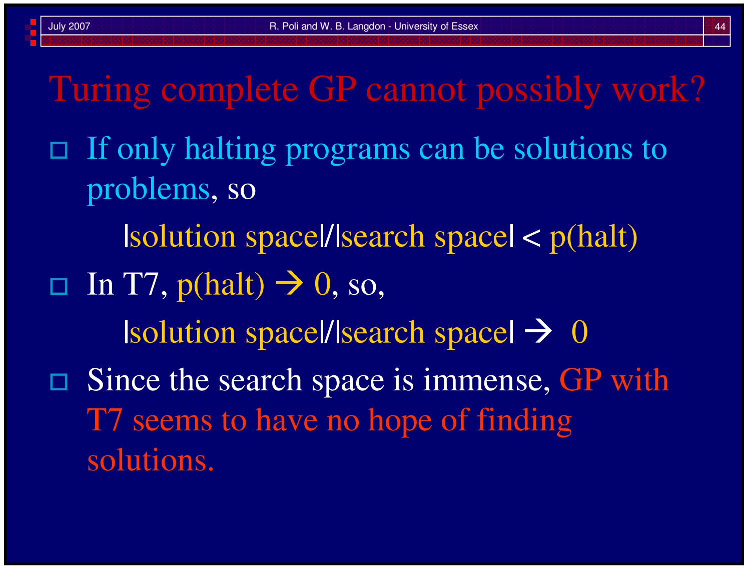
A good model?





Number of halting programs rises exponentially with length

$10^{100\ 000\ 000}$



Turing complete GP cannot possibly work?

- □ If 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

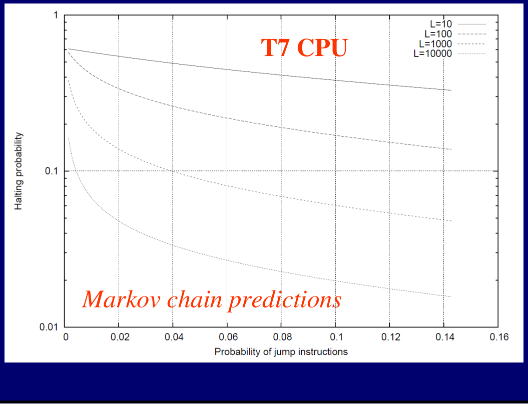
- □ Since the search space is immense, GP with T7 seems to have no hope of finding solutions.

What can we do?

- Control p(halt)
- Size population appropriately
- Design fitness functions which promote termination
- Repair
- Use result of program even if it is still running
- ....
- Any mix of the above

Controlling *p(halt)*

- Modify the probability of using jumps



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

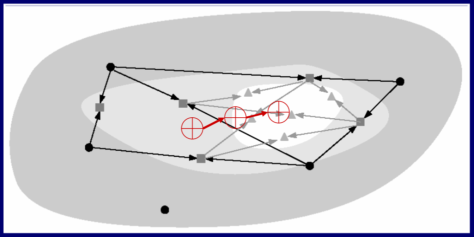
Limiting distribution of functionality for halting programs?

- 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 $L = 10^6$).

GP Search Characterisation

GA and GP search

- □ GAs and GP search like this:

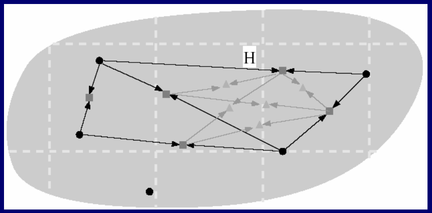


- □ How can we understand (characterise, study and predict) this search?

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

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

Exact Schema Theory for GP with Subtree Crossover

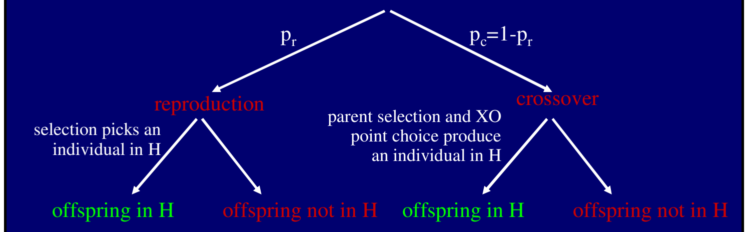
GP Schemata

- *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.
- 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?

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



p_r

p_c=1-p_r

reproduction

crossover

selection picks an individual in H

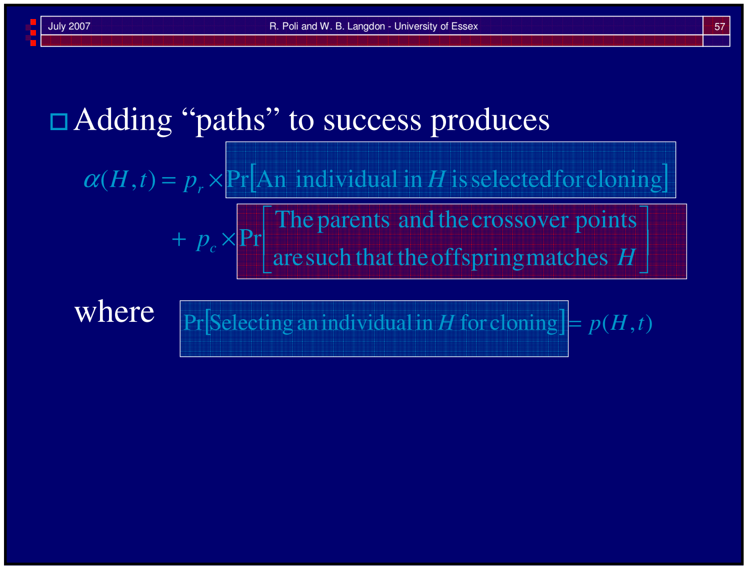
parent selection and XO point choice produce an individual in H

offspring in H

offspring not in H

offspring in H

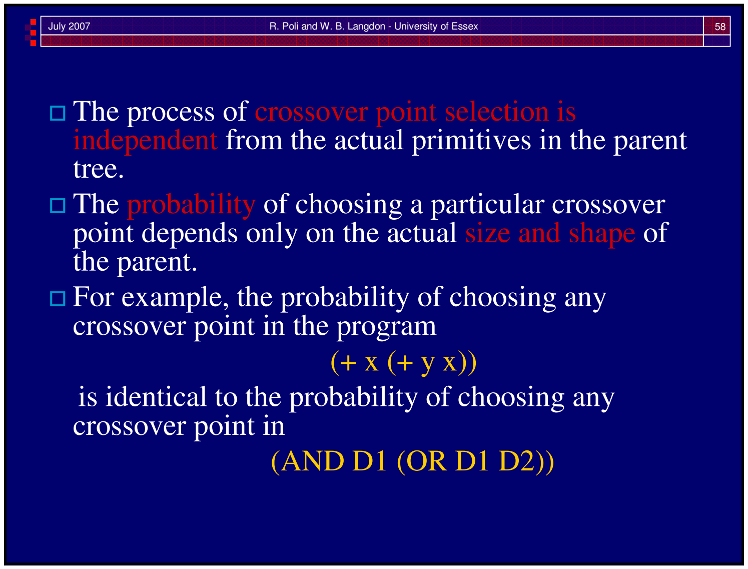
offspring not in H



- Adding "paths" to success produces

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

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

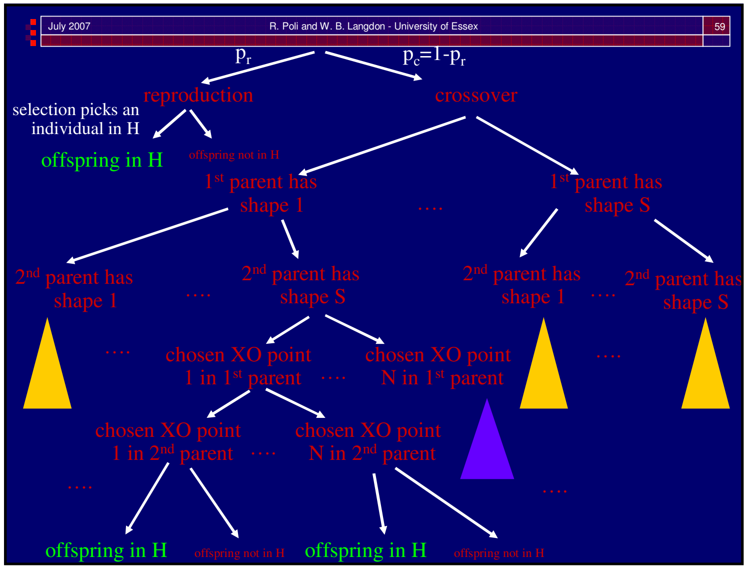


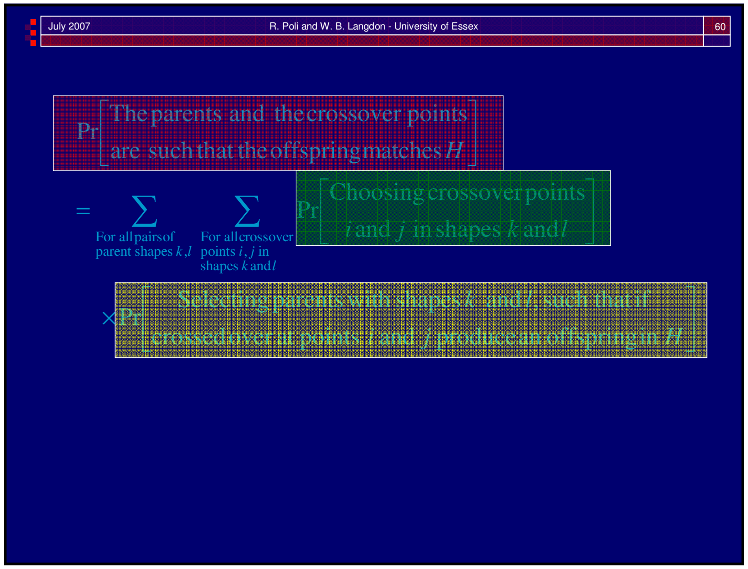
- The process of crossover point selection is independent from the actual primitives in the parent tree.
- The probability of choosing a particular crossover point depends only on the actual size and shape of the parent.
- For example, the probability of choosing any crossover point in the program

  (+ x (+ y x))

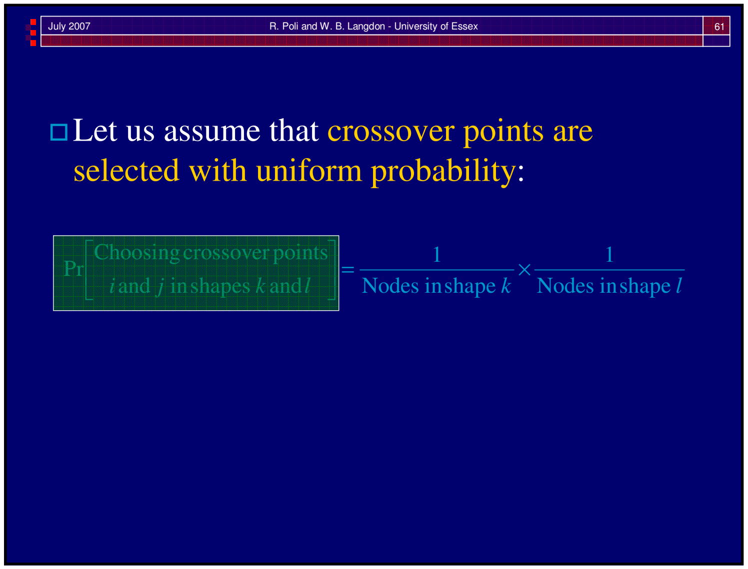
  is identical to the probability of choosing any crossover point in

  (AND D1 (OR D1 D2))



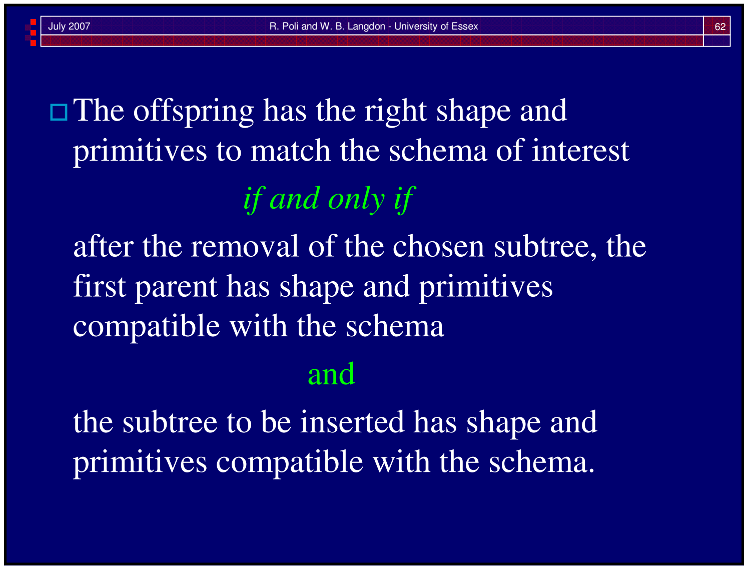


$$\Pr\left[\begin{array}{c}\text{The parents and the crossover points}\\ \text{are such that the offspring matches } H\end{array}\right] = \sum_{\substack{\text{For all pairs of}\\ \text{parent shapes } k,l}} \; \sum_{\substack{\text{For all crossover}\\ \text{points } i,j \text{ in}\\ \text{shapes }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



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

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



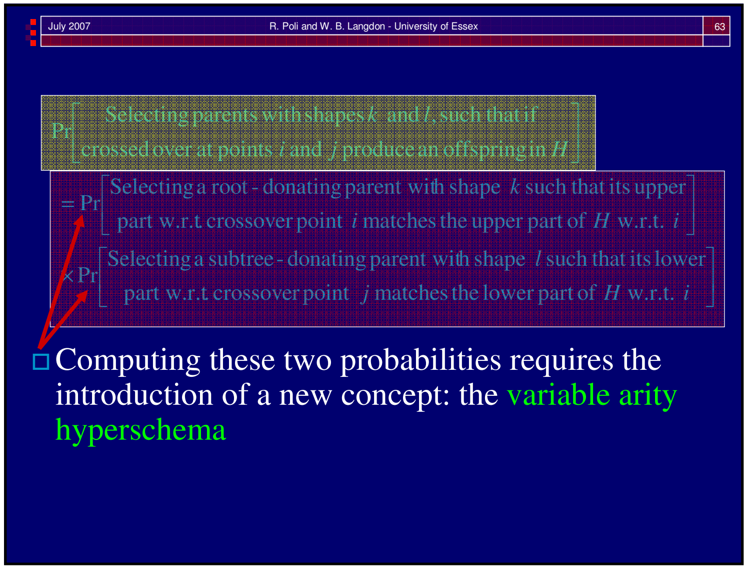
- The offspring has the right shape and primitives to match the schema of interest

  if and only if

  after the removal 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.

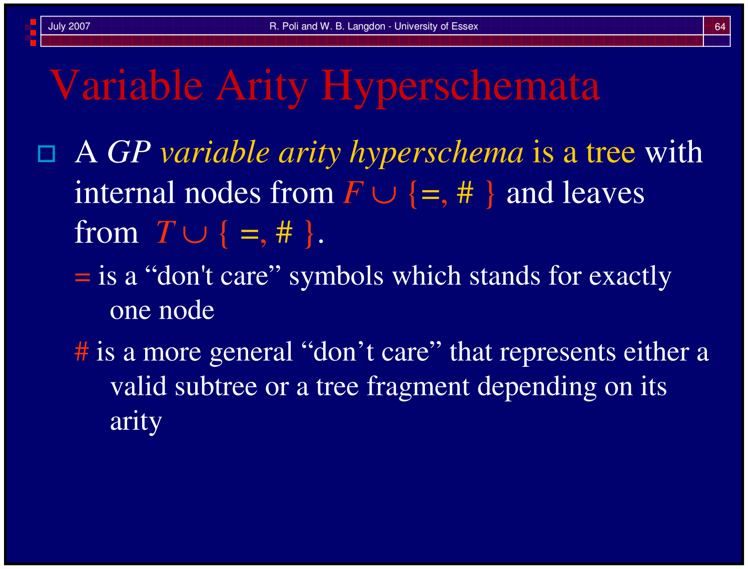


$$\Pr\left[\begin{array}{c}\text{Selecting parents with shapes } k \text{ and } l \text{ such that if}\\ \text{crossed over at points } i \text{ and } j \text{ produce an offspring in } H\end{array}\right]$$

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

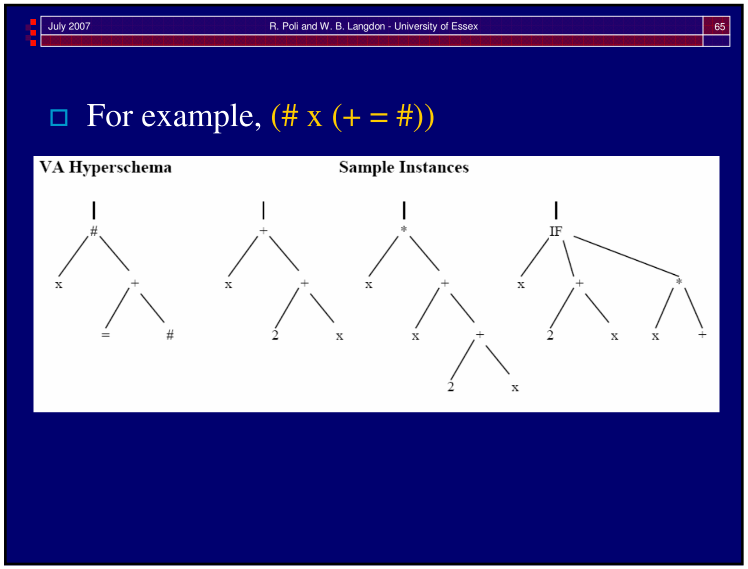
$$\times \Pr\left[\begin{array}{c}\text{Selecting a subtree-donating parent with shape } l \text{ such that its lower}\\ \text{part w.r.t. crossover point } j \text{ matches the lower part of } H \text{ w.r.t. } i\end{array}\right]$$

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

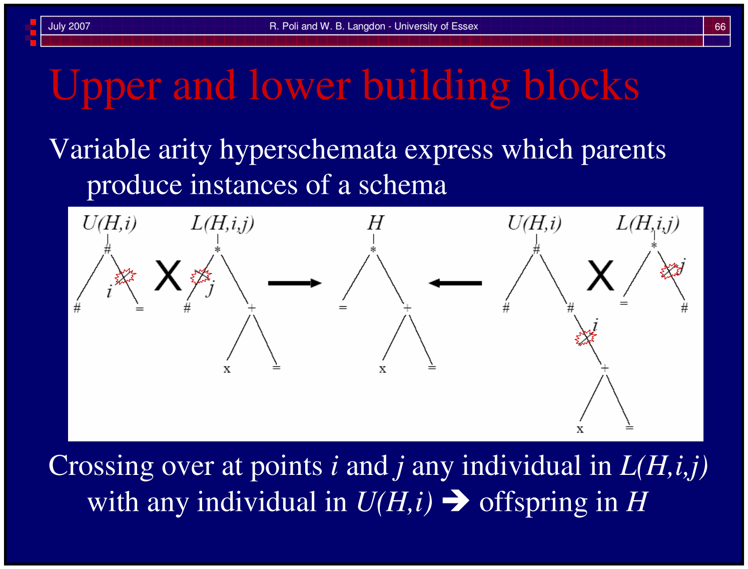


Variable Arity Hyperschemata

- 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
  - # is a more general “don’t care” that represents either a valid subtree or a tree fragment depending on its arity



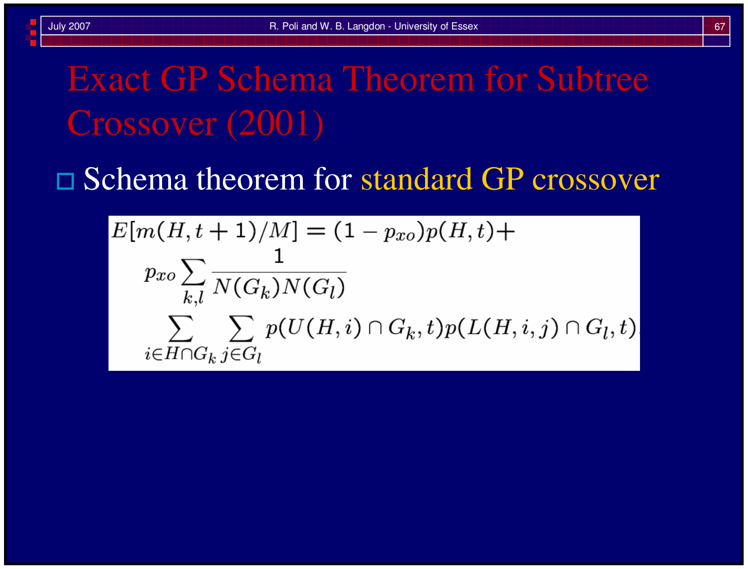
- For example, (# x (+ = #))



Upper and lower building blocks

Variable arity hyperschemata express which parents produce instances of a schema

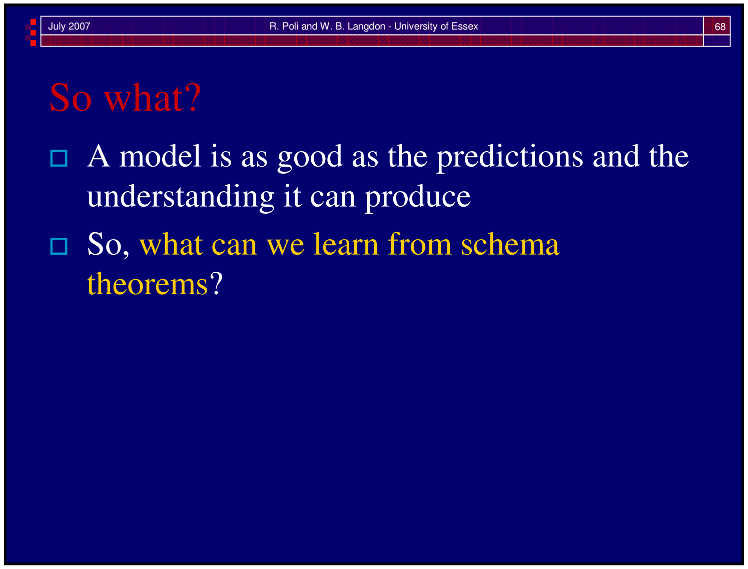
Crossing over at points *i* and *j* any individual in *L(H,i,j)* with any individual in *U(H,i)* ➔ offspring in *H*



Exact GP Schema Theorem for Subtree Crossover (2001)

- Schema theorem for standard GP crossover

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



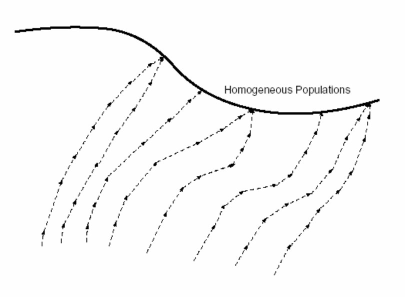
So what?

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

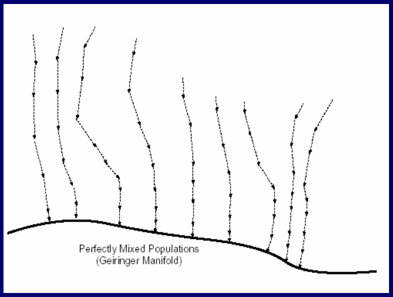
Lessons

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

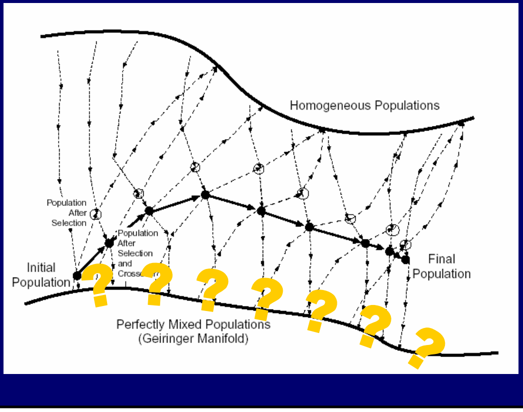
Selection Bias

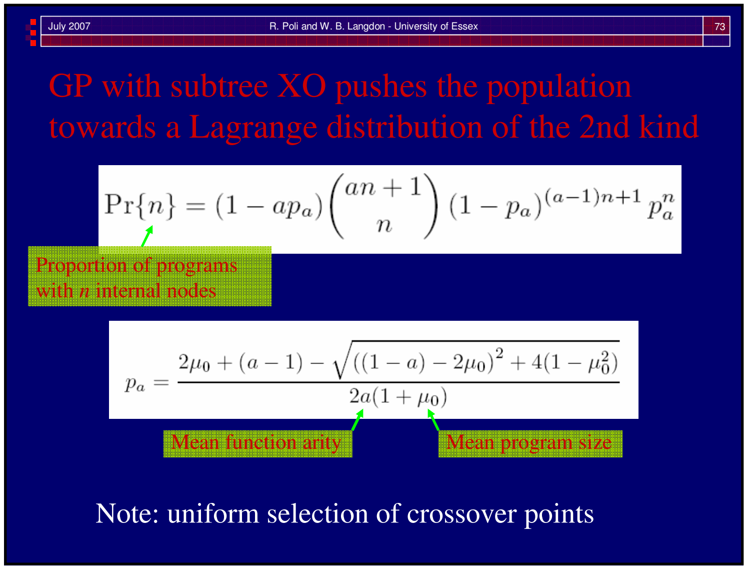


Crossover Bias



So where is evolution going?





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

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

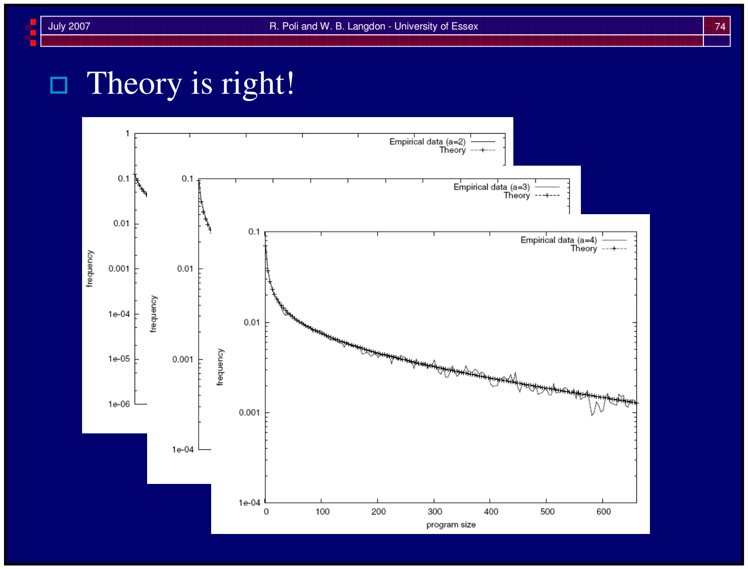
Proportion of programs with n internal nodes

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

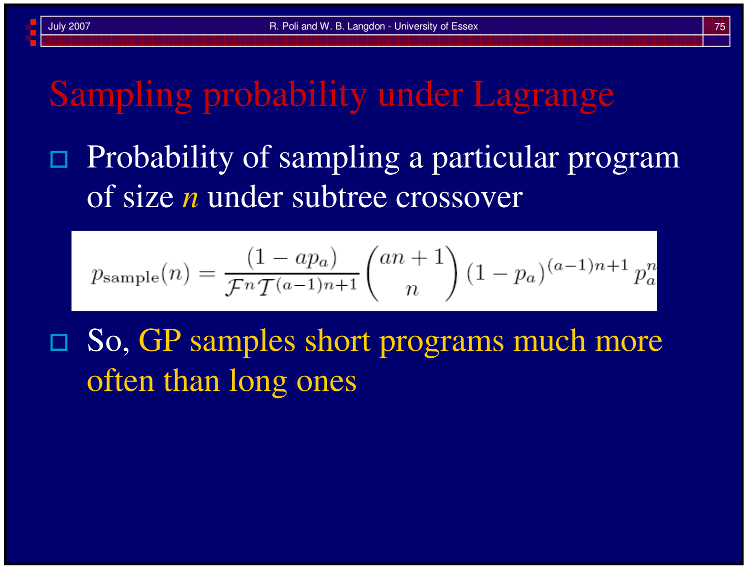
Mean function arity

Mean program size

Note: uniform selection of crossover points



- □ Theory is right!

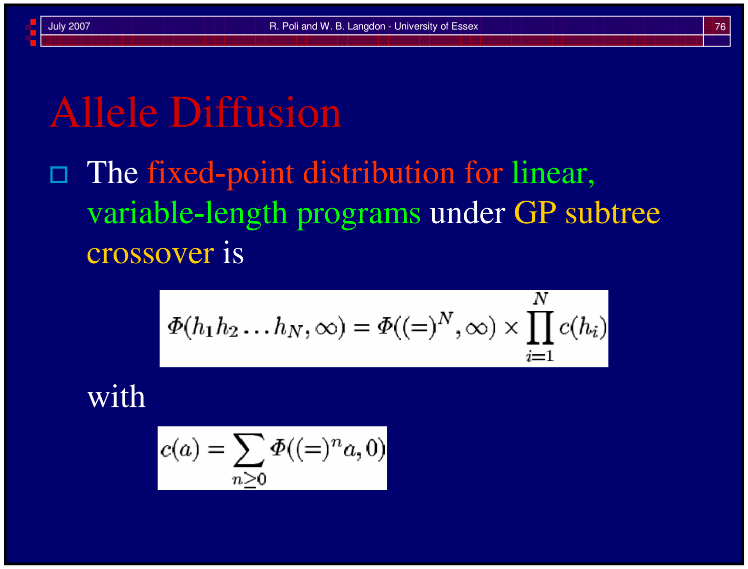


Sampling probability under Lagrange

- □ 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

$$\Phi(h_1h_2\ldots h_N,\infty) = \Phi((=)^N,\infty)\times\prod_{i=1}^{N}c(h_i)$$

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

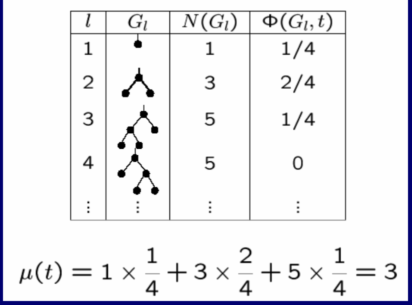
G_l = set of programs with shape l

$N(G_l)$ = 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

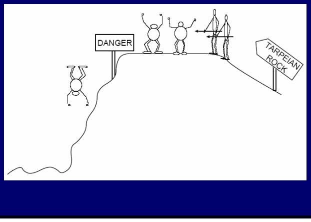
$$\sum_l N(G_l)\,[p(G_l,t) - \Phi(G_l,t)] > 0$$

↓

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

Tarpeian Bloat Prevention

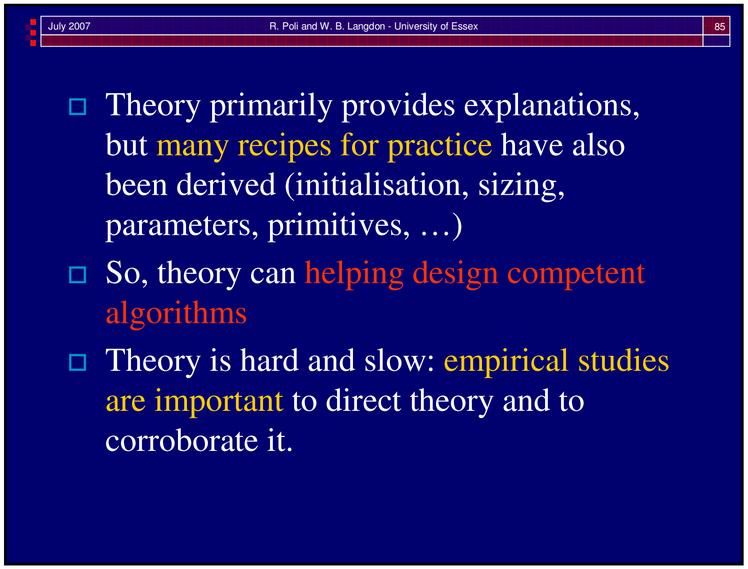
- □ To prevent growth one needs
  - To increase the selection probability for below-average-size programs
  - To decrease the selection probability for above-average-size programs



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, …)
- So, theory can helping design competent algorithms
- Theory is hard and slow: empirical studies are important to direct theory and to corroborate it.