

Genetic Programming Theory

Riccardo Poli

*School of Computer Science and Electronic Engineering
University of Essex*

Copyright is held by the author/owner(s).
GECCO'10, July 7–11, 2010, Portland, Oregon, USA.
ACM 978-1-4503-0073-5/10/07.

Overview

- Motivation
- Search space characterisation
 - How many programs?
 - Limiting fitness distributions
 - Implications
- GP search characterisation
 - Schema theory
 - Search bias
- Bloat
 - What's bloat?
 - Reasons
 - How to avoid it
- Conclusions

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

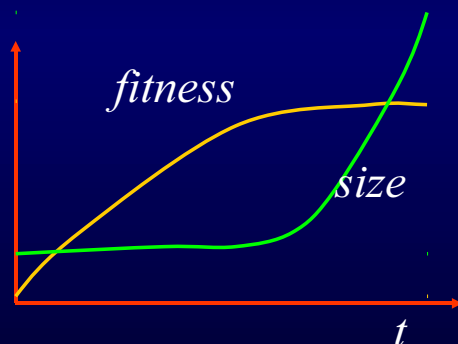
- GP is a complex adaptive system with **zillions of degrees of freedom**.
- So, any small number of descriptors can capture only a fraction of the complexities of such a system.
- **Choosing** which problems, parameter settings and **descriptors** to use is an **art form**.
- **Plotting the wrong data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

Example: Bloat

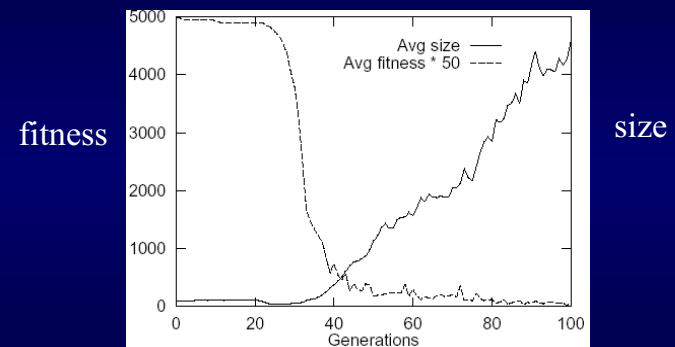
- What's bloat?
- Run demo

Bloat

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



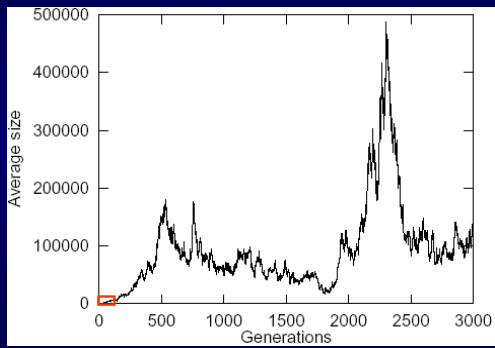
Fitness vs size



- Bloat exists and **continues forever**, right?

Why do we need 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**

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

Example

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

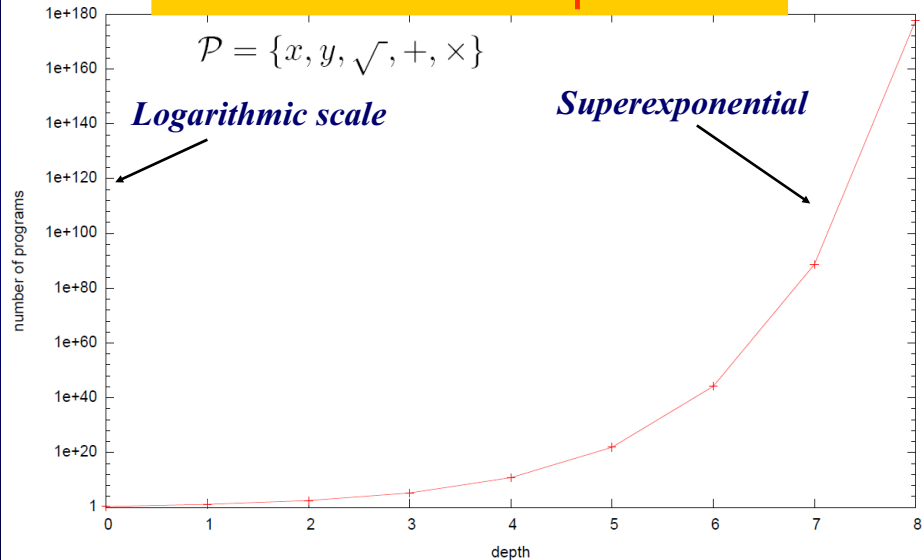
$$a_{\max} = 2, \mathcal{P}_0 = \{x, y\}, \mathcal{P}_1 = \{\sqrt{\quad}\} \quad \mathcal{P}_2 = \{+, \times\}$$

$$n_0 = 2$$

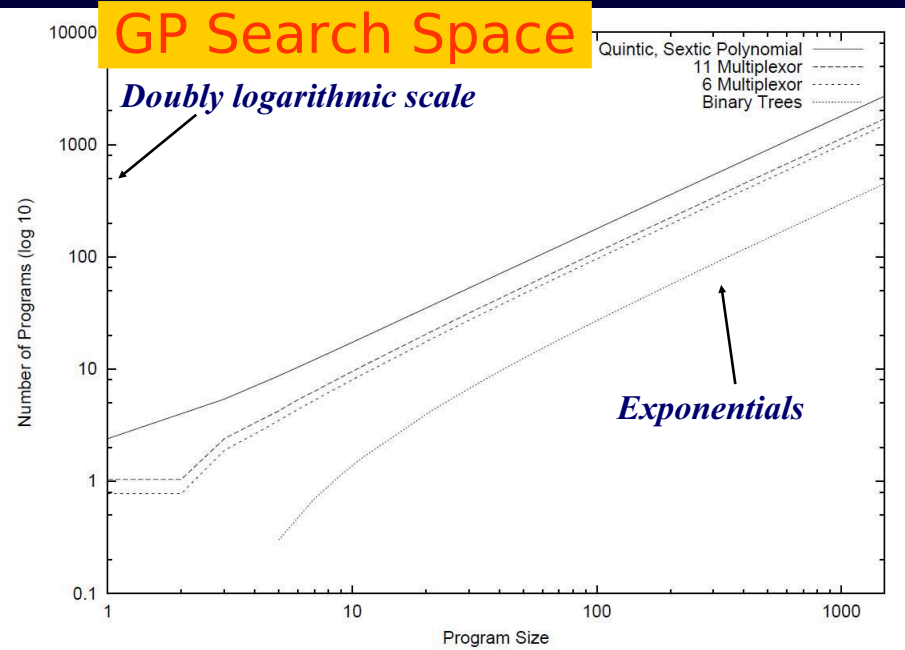
$$n_1 = 2 + 1 \times (n_0) + 2 \times (n_0)^2 = 12$$

$$n_2 = 2 + 1 \times (n_1) + 2 \times (n_1)^2 = 302$$

GP Search Space



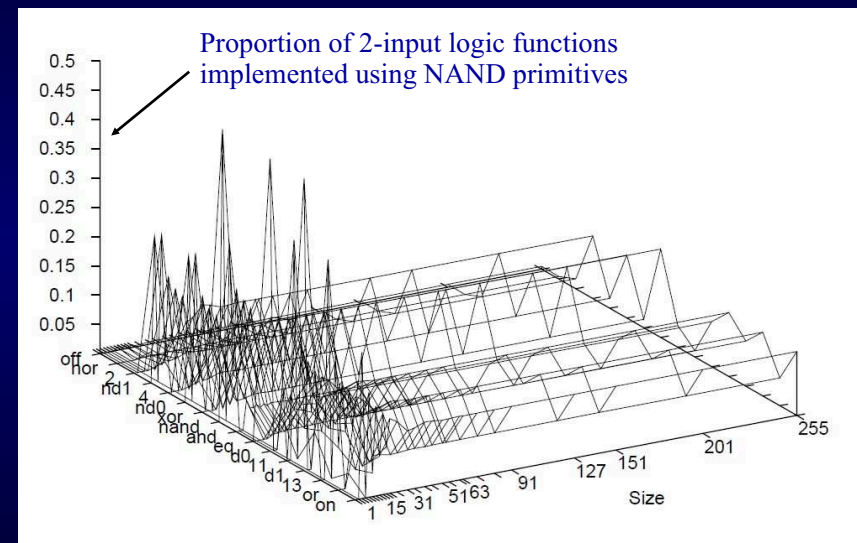
GP Search Space



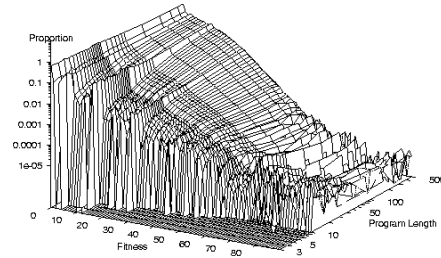
GP cannot possibly work!

- 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

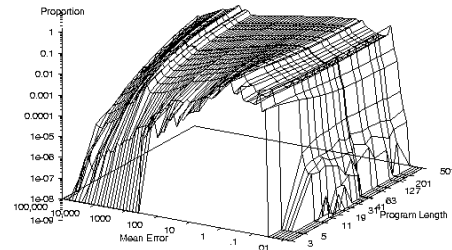
{d0,d1,NAND} search space



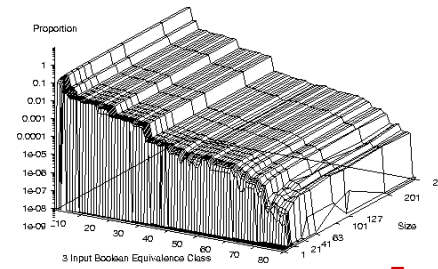
Proportion of Ant programs with each score



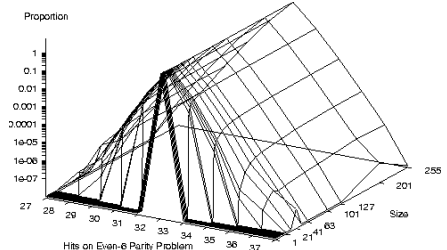
Error Distribution Sextic Polynomial Problem



Distribution of 3 bit Boolean Functions



Even-6 parity program space



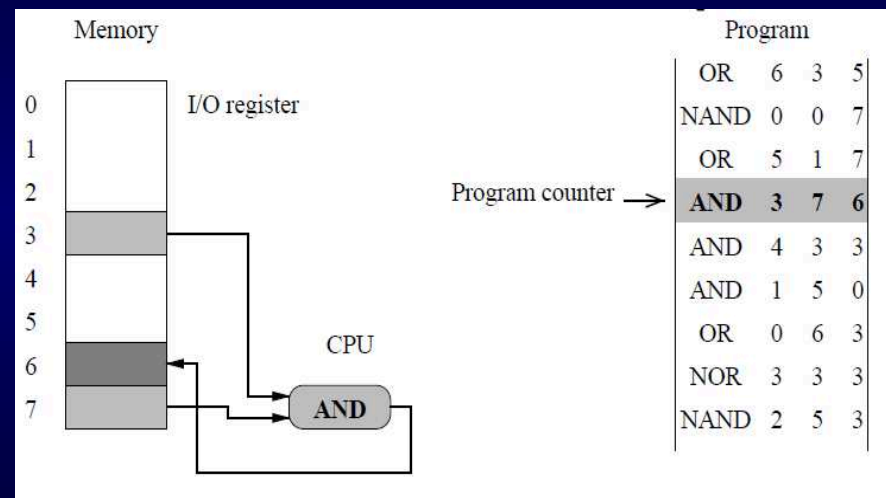
Limiting distribution

- Empirically it 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



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 $m_0 m_1 \rightarrow m_0$ 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 $m_0 m_1 \rightarrow m_0$

NOP

OR $m_0 m_1 \rightarrow m_0$

AND $m_0 m_1 \rightarrow m_0$

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?

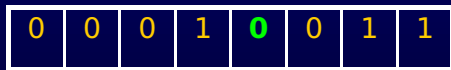
Two primitives: AND $m_0 m_1 \rightarrow m_0$ OR $m_0 m_1 \rightarrow m_0$



*Identity function
(no instruction
executed yet)*

AND $m_0 m_1 \rightarrow m_0$ 1/2

1/2 OR $m_0 m_1 \rightarrow m_0$



A



B



A

AND $m_0 m_1 \rightarrow m_0$ 1/2

1/2 OR $m_0 m_1 \rightarrow m_0$



A



C



B

AND $m_0 m_1 \rightarrow m_0$ 1/2

1/2 OR $m_0 m_1 \rightarrow m_0$



C



B



C

AND $m_0 m_1 \rightarrow m_0$ 1/2

1/2 OR $m_0 m_1 \rightarrow m_0$

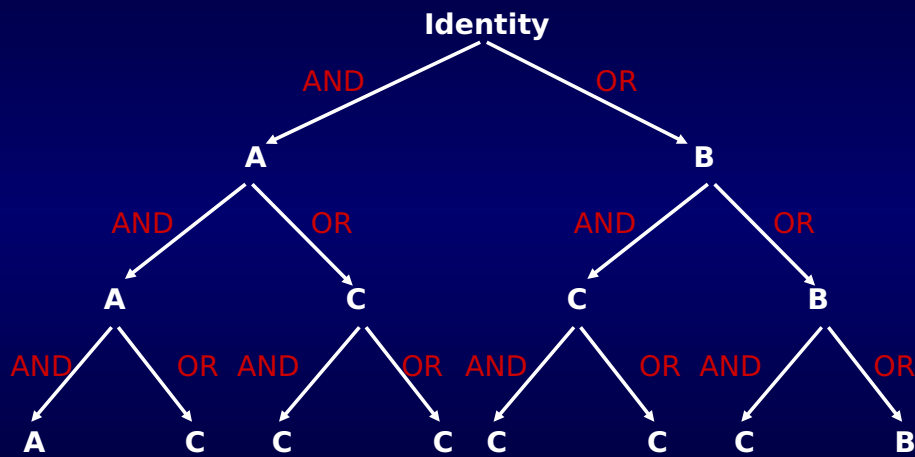


C



C

Probability tree



Distribution of behaviours

Program length	Behaviour A	Behaviour B	Behaviour C	Identity
0	0	0	0	1
1	$\frac{1}{2}$	$\frac{1}{2}$	0	0
2	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{2}$	0
3	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{3}{4}$	0
4	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{7}{8}$	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 m_1 into m_0 .

Markov chain proofs of limiting distribution

- Using Markov chain theory Bill Langdon 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.

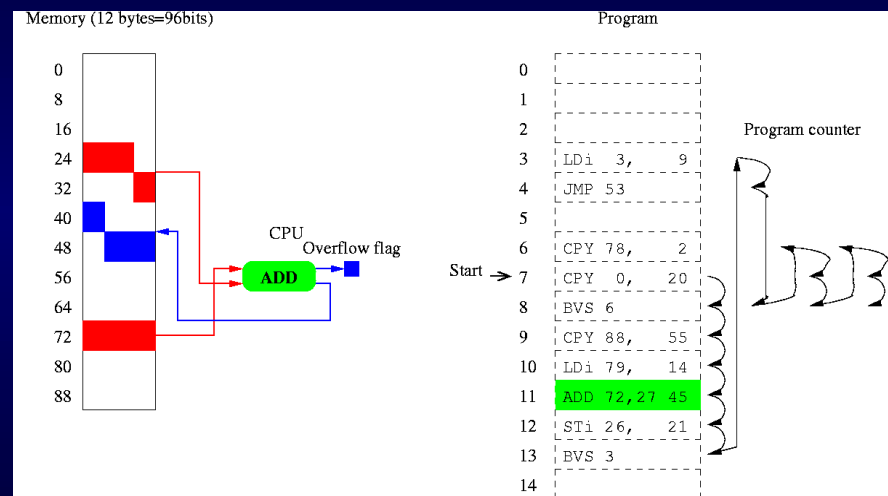
Implication of $|\text{solution space}|/|\text{search space}|=\text{constant}$

- 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.

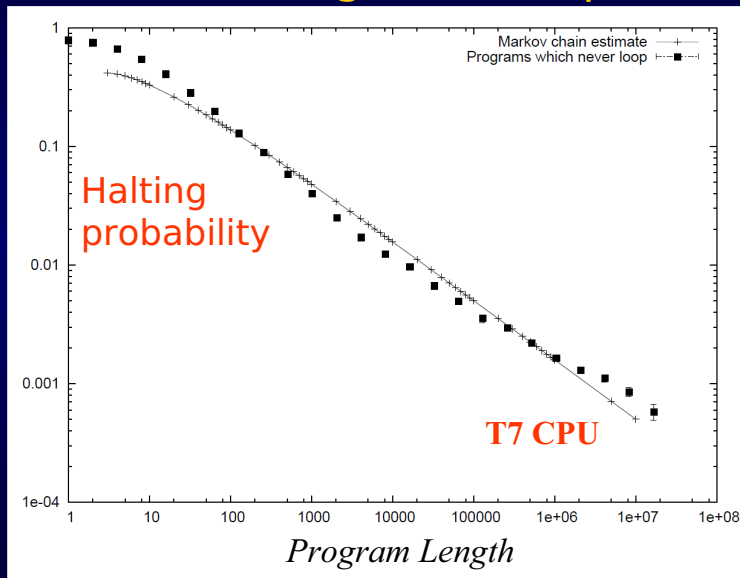
What about Turing complete GP?

- **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?

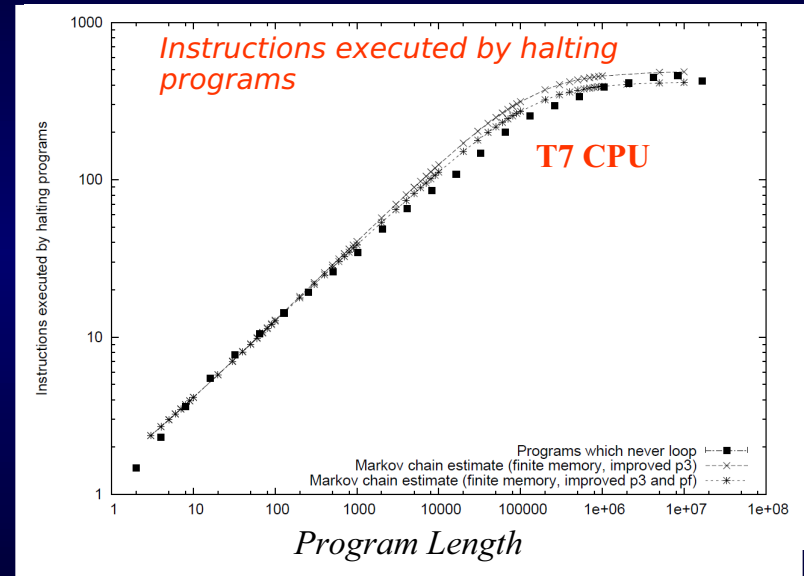
T7 Architecture



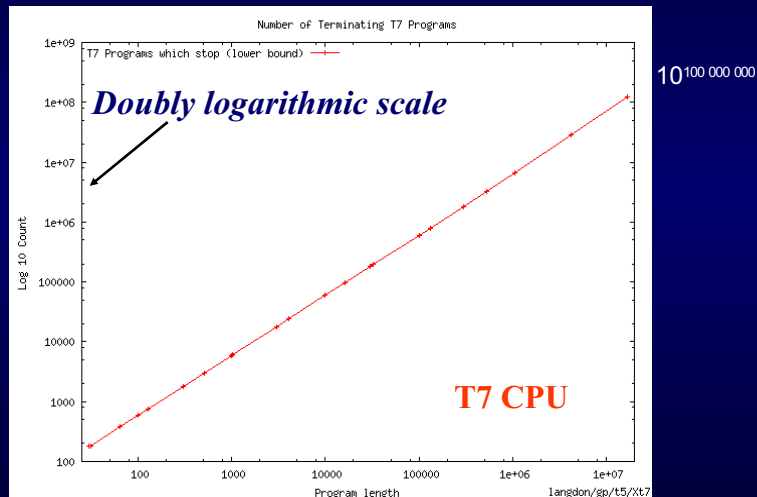
Almost all T7 Programs Loop



Halting programs have limited active code



Number of halting programs rises exponentially with length



T7 Turing complete GP cannot possibly work?

- If only halting programs can be solutions to problems, then $|\text{solution space}|/|\text{search space}| < p(\text{halt})$
- In T7, $p(\text{halt}) \rightarrow 0$, so, $|\text{solution space}|/|\text{search space}| \rightarrow 0$
- Since the search space is immense, GP with T7 seems to have no hope of finding solutions.

What can we do?

- Control $p(\text{halt})$
- 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

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?

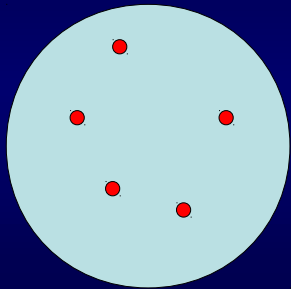
- So, as the number of instructions executed grows, the distribution of functionality of non-looping programs approaches a limit.
- Number of instructions executed, not program length, tells us how close the distribution is to the limit

No Free Lunch

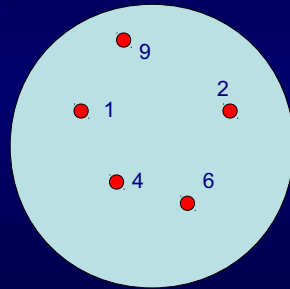
- **Wolpert & Macready:** when evaluated over all possible problems, all algorithms are equally good (or bad) irrespective of our evaluation criteria.
- **Schumacher, Vose, Whitley:** if one selects a set of fitness functions that is closed under permutation then the expected performance of any search algorithm over that set is constant (and vice versa)

Closed under permutation?

A search space

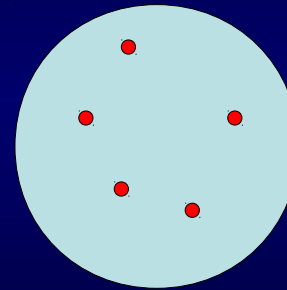


A fitness function

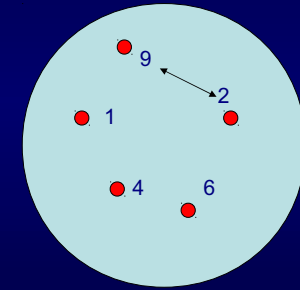


Closed under permutation?

A search space



A permutation of the fitness function



Closed under permutation?

- A set of fitness functions is closed under permutation if for every function in the set all possible permutations of that function are also in the set

Does No-free Lunch Apply to GP?

- The fitness of program p is obtained by evaluating p 's behaviour on the fitness cases and adding up the errors

$$f(p) = \sum_{i=1}^n g(p(x_i), t(x_i))$$

with

$$g(a, b) = |a - b|^k$$

Geometric interpretation of fitness

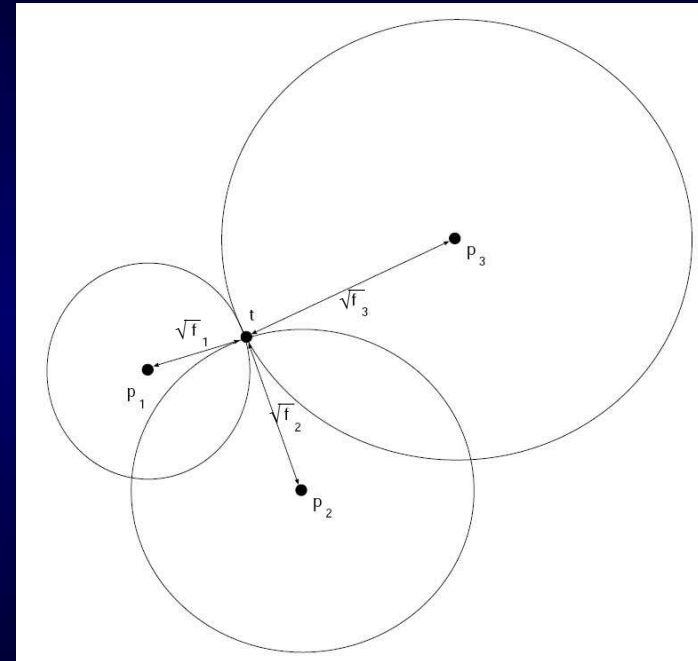
- $k=1 \rightarrow f(p)$ is a distance

$$f(\mathbf{p}) = d(\mathbf{p}, \mathbf{t})$$

$$\mathbf{p} = (p(x_1), p(x_2), \dots, p(x_n))$$

$$\mathbf{t} = (t_1, t_2, \dots, t_n)$$

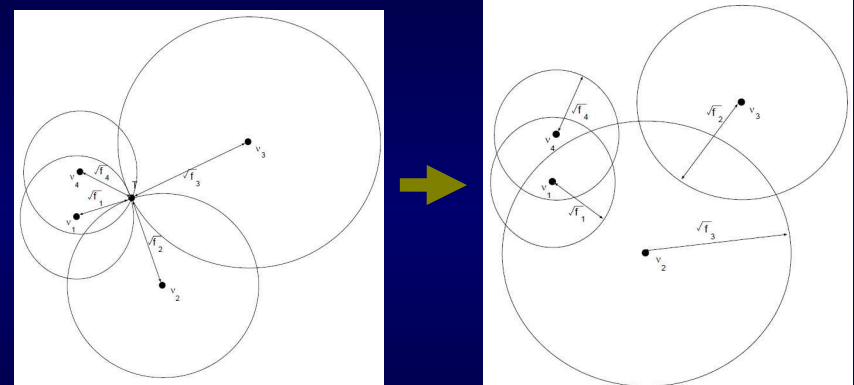
- $k=2 \rightarrow \text{sqrt}(f(p))$ is a distance



NFL does not apply to GP

- Under a permutation, fitnesses (the radii of our circles) are rearranged
- Often, the new circles do not meet at one point, i.e., a set of fitness cases that produce the new fitness function does not exist
- So, the set of problems is not closed under permutation and NFL does not hold.

Example



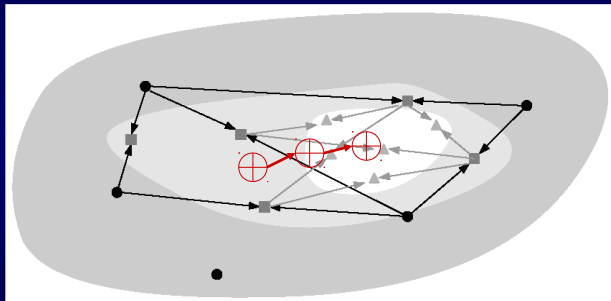
So what?

- There exists at least one performance measure under which not all GP systems are equally good
- Trying to find a super-GP algorithm is not necessarily a waste of time (as NFL would imply)
- Can we use geometry to guide us? (constructive proof)

GP Search Characterisation

GA and GP search

- GAs and GP search like this:



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

Detailed Markov Models of GP

- GP is Markovian: the next generation depends only on the current population
- So, we can model GP by computing the probability of the population changing into an other population at the next generation for all possible populations
- We store such probabilities in a matrix M

Detailed Markov Models of GP

- M contains all we know about GP
- M^k gives us the behaviour of GP over k generations
- M is huge and, thus, Markov models are hard to use

GP is guaranteed to find solutions to problems if...

- ...its Markov matrix M is ergodic
- Ergodicity means that there for some k all elements of M^k are non-zero, i.e., in k generations there is a non-zero probability to go from any population to any other population...
- ...including one containing a solution!

Guaranteeing Ergodicity

- In fixed-length EAs ergodicity is guaranteed if there is non-zero point mutation
- In GP it is harder to prove formally (variable size representation, infinite search space)

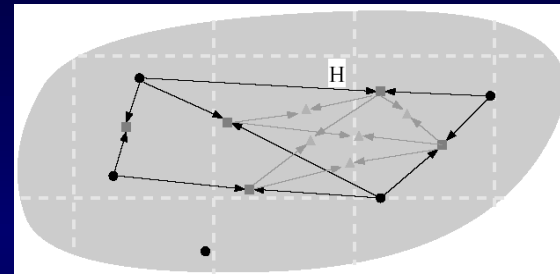
Informal ergodicity argument for GP

- **If** we limit the search space
- **And** we use a mutation operator that can generate any point in it
- **And** the root node can be selected as a mutation point
- **Then** M is ergodic and, so, *GP is guaranteed to find solutions to all problems, given enough time!*
- The use of additional operators (XO) may speed up the 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

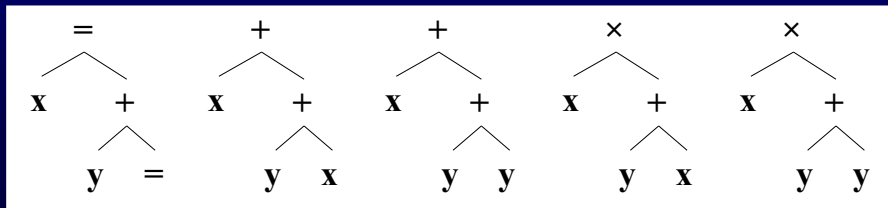
- The **selection/crossover/mutation** process is a random **coin flip** (**Bernoulli trial**). New individuals are either in schema H or not.
- So, $m(H,t+1)$ is a **binomial** stochastic variable.
- Given the **success probability** of each trial $\alpha(H,t)$, an **exact schema theorem** is

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

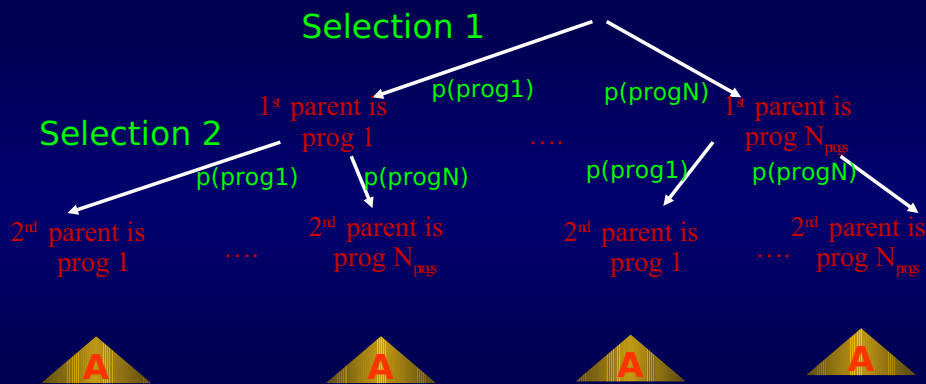
Exact Schema Theory for GP with Subtree Crossover

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.



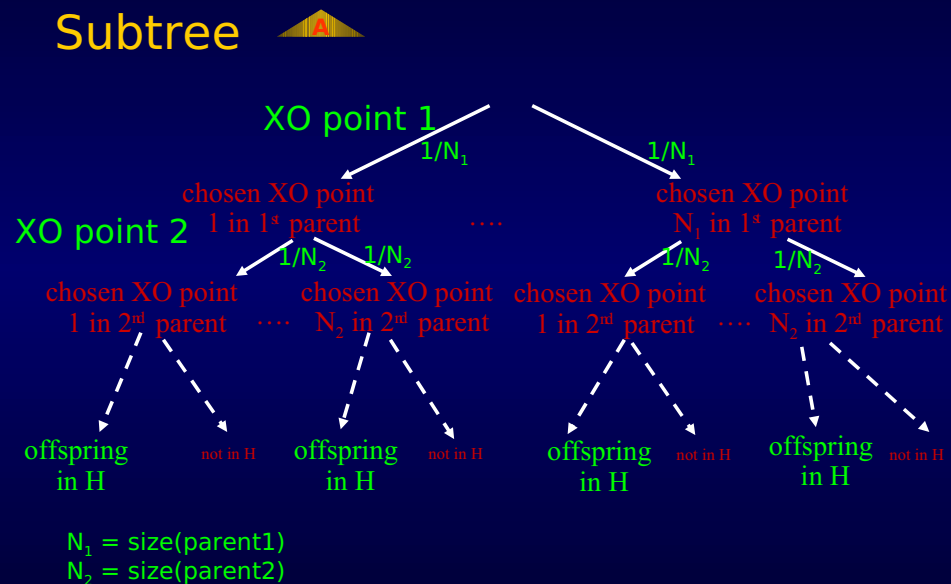
Selection-Crossover Probability tree



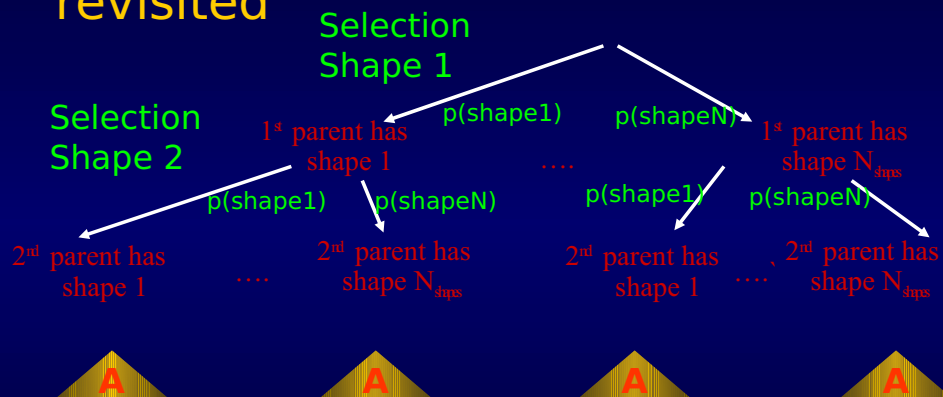
Creation of individuals via crossover is a compound event

{create individual} =
 {select parent 1,
 select parent 2,
 choose crossover point 1,
 choose crossover point 2 }

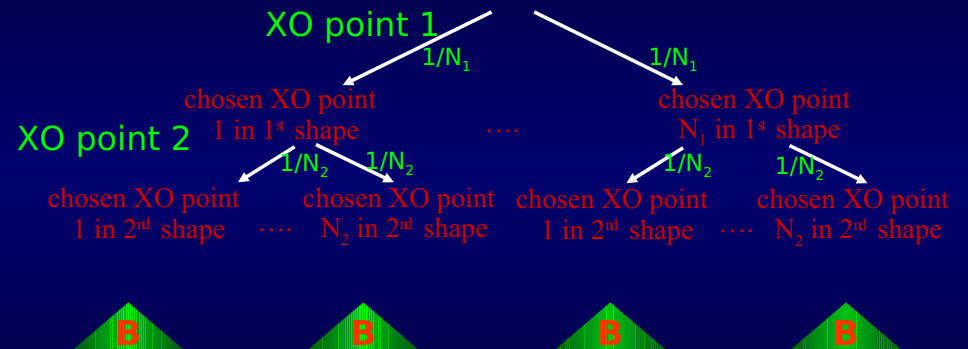
Subtree



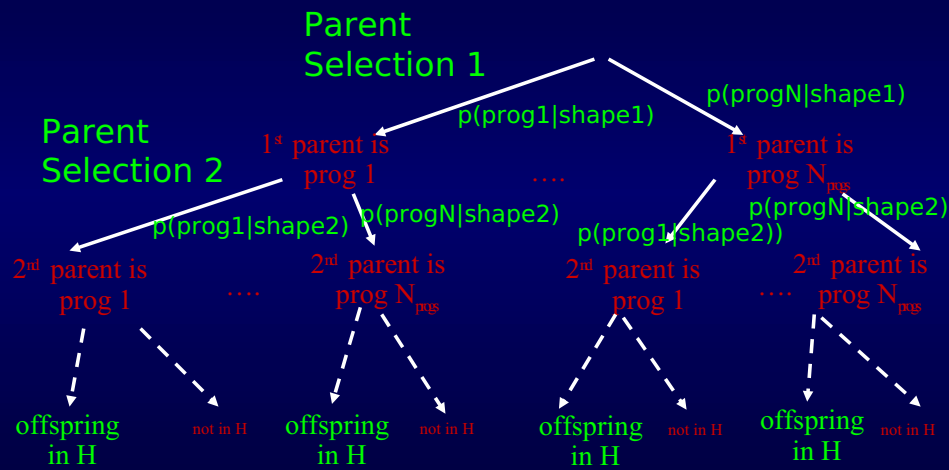
Selection-XO Probability Tree revisited



Subtree **A** revisited



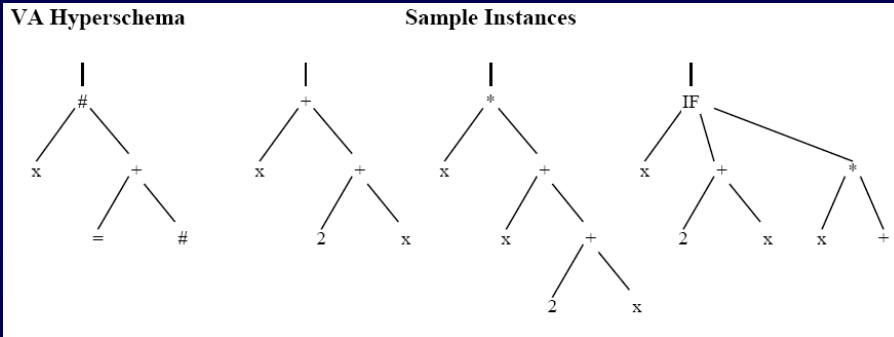
Subtree **B** (take 1)



Variable Arity Hyperschemata

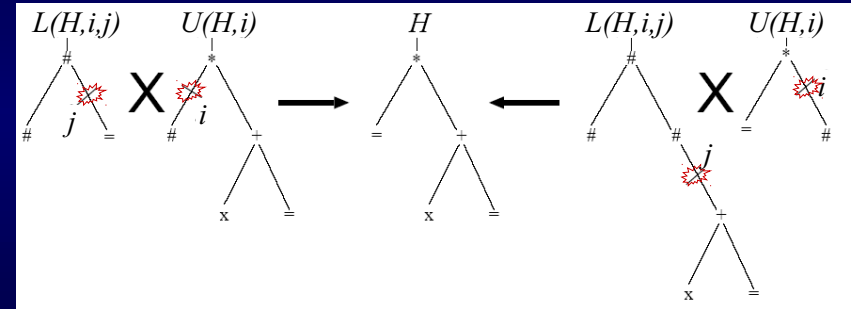
- A GP variable arity hyperschema is a tree with internal nodes from $F \cup \{=, \#\}$ and leaves from $T \cup \{=, \#\}$.
 - = is a “don't care” symbols which stands for exactly one node
 - # is a more general “don't care” that represents either a valid subtree or a tree fragment depending on its arity

□ For example, $(\# \times (+ = \#))$



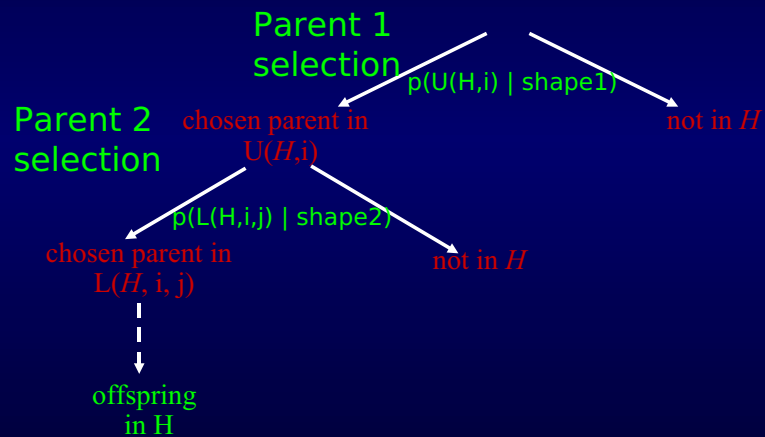
Upper and lower building blocks

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 $U(H,i) \rightarrow$ offspring in H

Subtree (take 2)



Bayes' Theorem

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

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

Exact GP Schema Theorem for Subtree Crossover

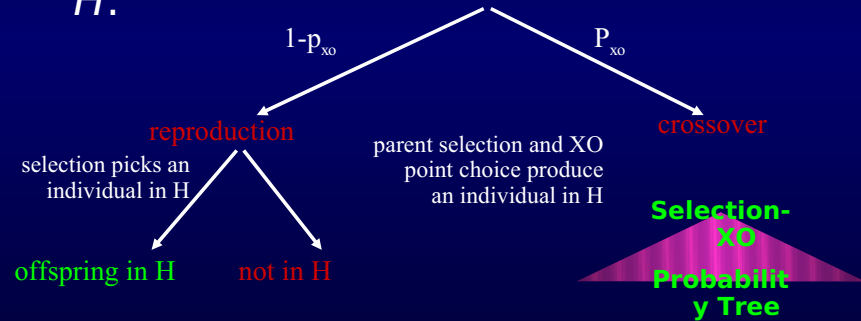
- Schema theorem for selection + 100% standard GP crossover

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

size(shape1)=N₁ size(shape2)=N₂
 XO points in shape2
 XO points in shape1

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

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

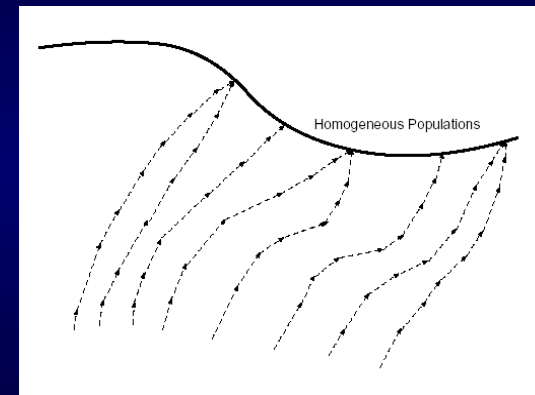
So what?

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

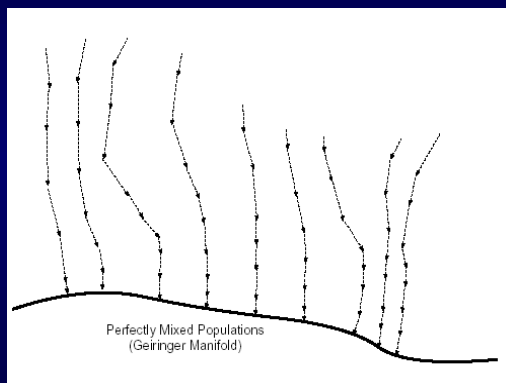
Lessons

- GP's convergence
- Operator biases
- Size evolution equation
- Bloat control
- Optimal parameter setting
- Optimal initialisation
- ...

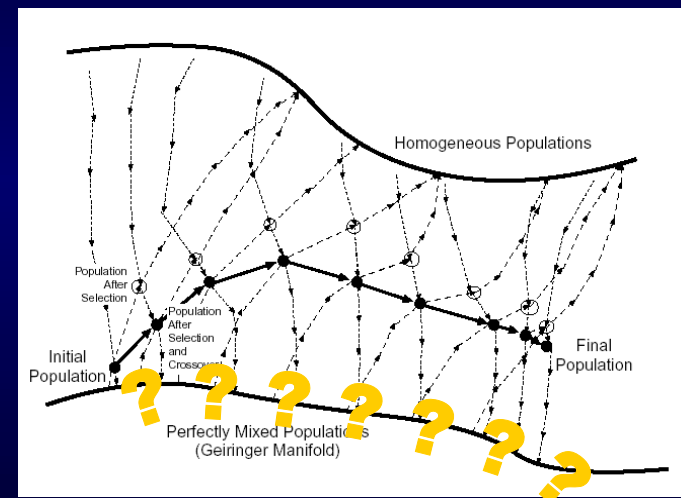
Selection Bias



Crossover Bias



So where is evolution going?



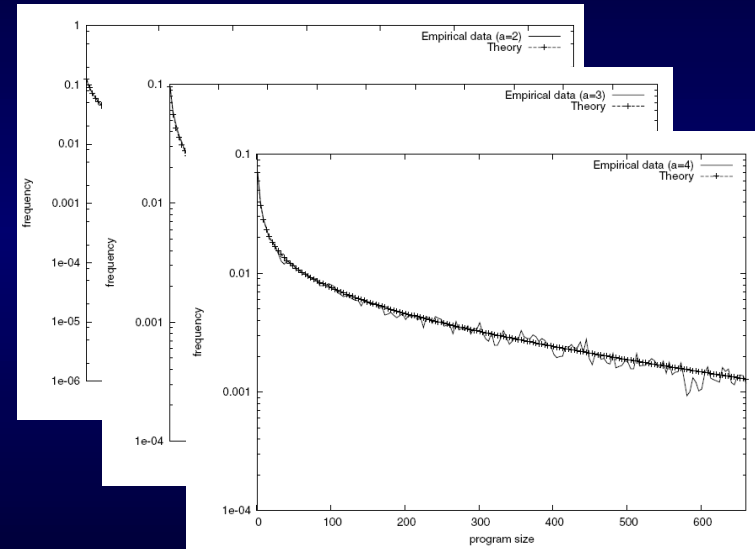
GP with subtree XO pushes the population towards a Lagrange distribution of the 2nd kind

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

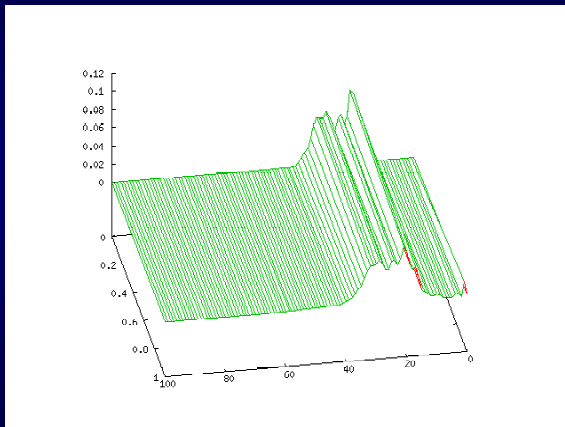
Proportion of programs with n internal nodes

Note: uniform selection of crossover points

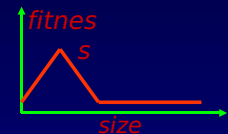
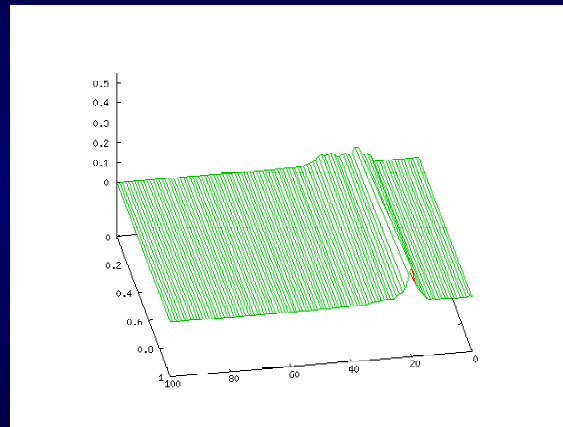
□ Theory is right!



How strong is the XO bias?



Can fitness override the bias?



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

Primitive Diffusion

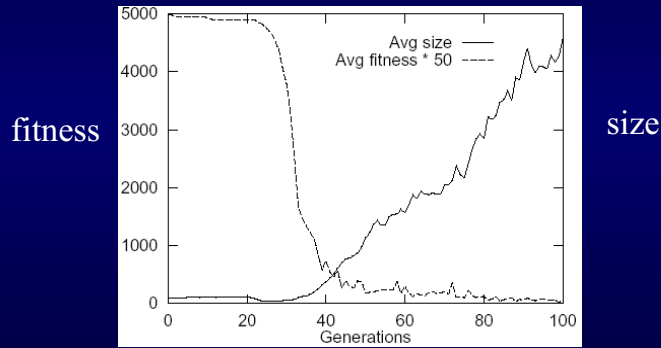
- In linear GP, XO pushes the population towards distributions where **each primitive is equally likely to be at any position in any individual**.
- The primitives in an individual are not just swapped with those of other individuals, they also **diffuse** within the representation of each individual.

Diffusion in tree-based GP

- For trees the situation is slightly more complex (see S. Dignum & R. Poli EuroGP 2010)
- **Arity histogram** = plot of # of primitives of each arity
- **All programs with the same arity histogram are equally likely** (so perfect diffusion within arity classes)
- But **arity histograms are not equally likely**, so slightly biased diffusion (with higher frequency of low arities near the root)

Bloat

Bloat



Ant Problem: Fitness

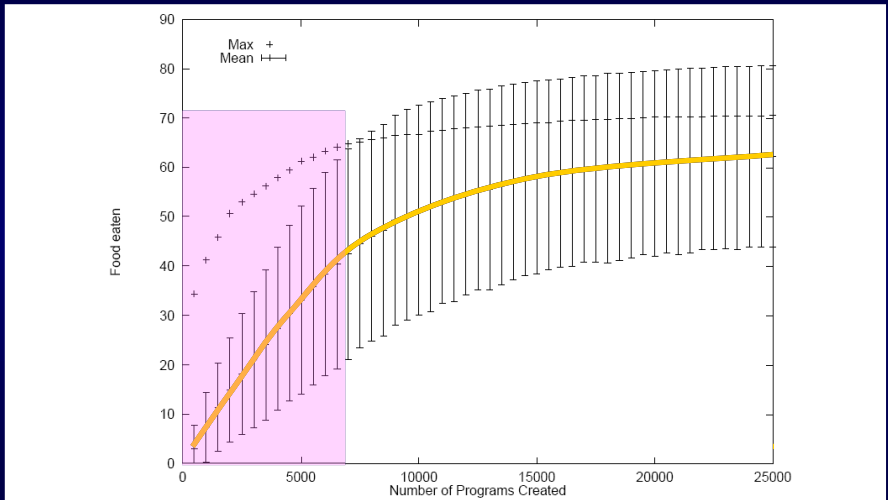


Figure 1: Evolution of maximum and population mean of food eaten. Error bars indicate one standard deviation. Means of 50 runs.

Ant Problem: Size

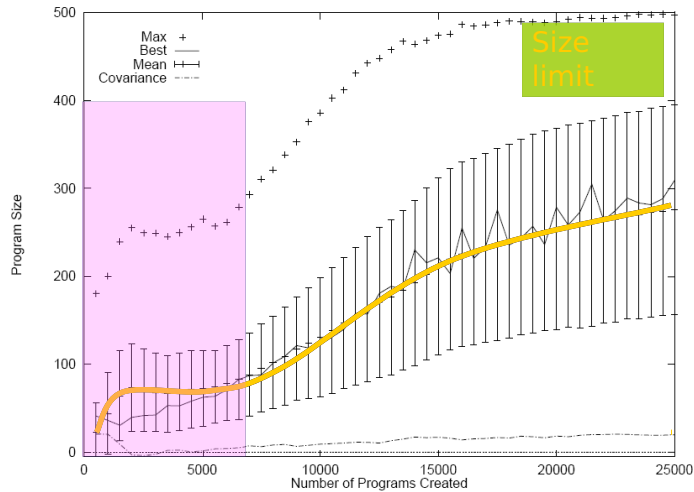


Figure 2: Evolution of maximum and population mean program length. Error bars indicate one standard deviation. Solid line is the length of the "best" program in the population, covariance of length and normalised rank based fitness shown dotted. Means of 50 runs.

More evidence: No Fitness = No Bloat

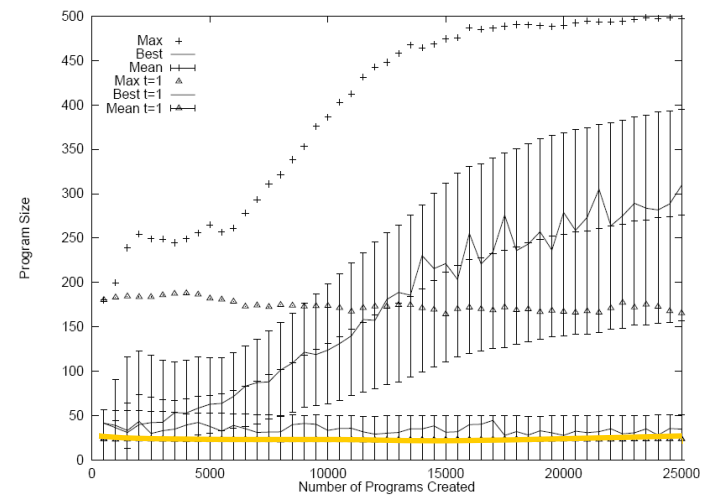


Figure 5: Evolution of maximum and population mean program length. Error bars indicate one standard deviation. Solid line is the length of the "best" program in the population. Means of 50 runs comparing tournament sizes of 7 and 1.

Early Theories

- Replication accuracy/protection against crossover
- Removal bias
- Drift in program spaces

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

$$\sum_i N(G_i) [p(G_i, t) - \Phi(G_i, t)] > 0$$

- Or

$$\sum_{\ell} \ell (p(\ell, t) - \Phi(\ell, t)) > 0$$

- Size evolution eq. = Price's theorem

$$E[\Delta\mu] = \frac{\text{Cov}(\ell, f)}{\bar{f}(t)}$$

Crossover Bias Theory of Bloat

1. Crossover does not change the mean program size, on average, but...
2. It creates a population of individuals with a large proportion of small programs.
3. In most problems, very small programs are less fit than longer programs (upon sufficient search), so they are ignored by selection. Thus...
4. Only larger programs will be picked as parents, hence increasing mean program size.

Are short programs unfit? Random Search

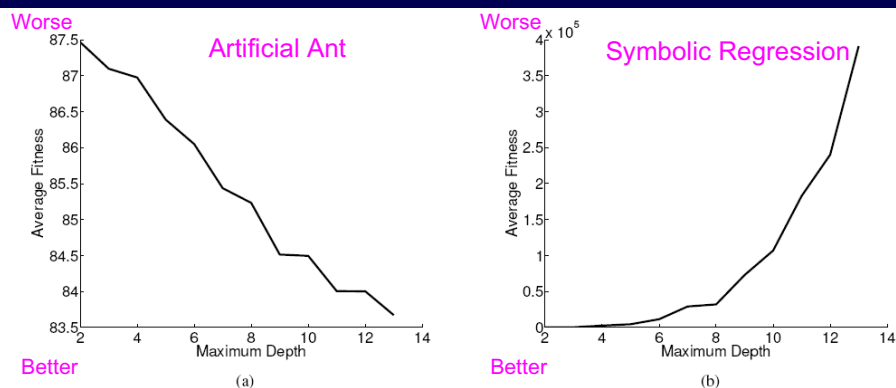


Figure 1: Average fitness vs. maximum depth over a sample of 4000 individuals generated with the Ramped Half and Half algorithm. All problems use minimization (the smaller the fitness, the better). (a): Artificial Ant on the Santa Fe trail. (b): Symbolic regression.

Are short programs unfit? Metropolis-Hastings

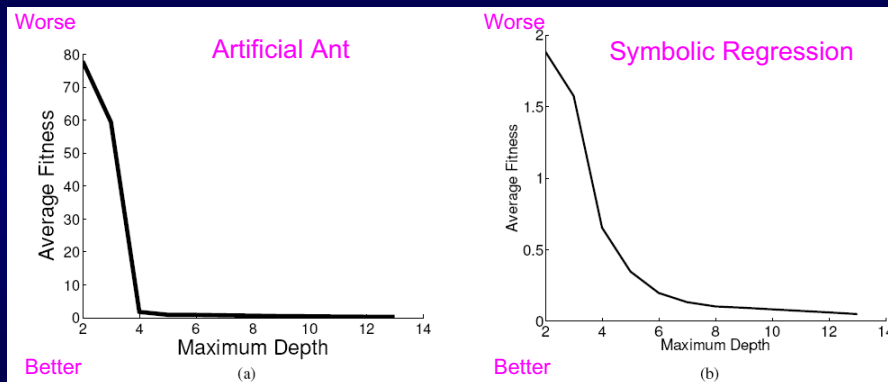


Figure 2: Average fitness vs. maximum depth over a sample of 4000 individuals generated with the Metropolis-Hastings algorithm (see the text for a definition of the acceptance criterium used). All problems use minimization (the smaller the fitness, the better). (a): Artificial Ant on the Santa Fe trail. (b): Symbolic regression.

Main techniques for limiting code bloat

- **Fixed size or depth limit:** Programs exceeding the limit are discarded and a **parent is kept instead**.
- This is **very dangerous** as it gives a fitness advantage to programs that tend to violate the constraint.

- **Parsimony pressure:** a term is added to the fitness function that penalises larger programs.
- Typically:

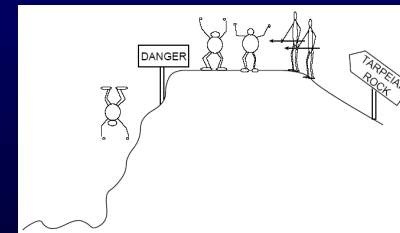
$$f_{\text{parsimony}}(\text{prog}) = f_{\text{raw}}(\text{prog}) - c * \text{size}(\text{prog})$$

where c is a constant.

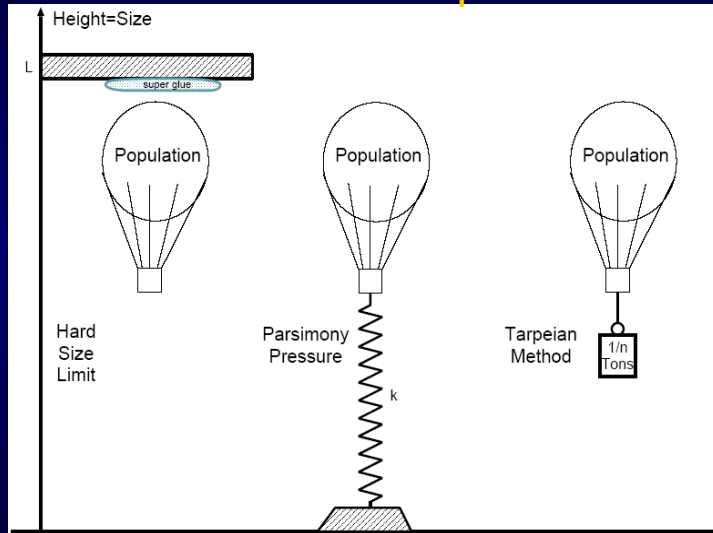
- **Modification of operators:** variation of the selection probability of crossover points by using explicitly defined introns, rejection of destructive crossover events, size-fair operators, MOO techniques, etc.
- For example, **point mutation** (applied with a fixed probability **per node**) **has an anti bloat effect**.
- Crossover equalisation

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**



Hot Air Balloon Metaphor



Covariant parsimony pressure

- Parsimonious fitness

$$f_p(x, t) = f(x) - g(\ell(x), t)$$

- Price:

$$E[\Delta\mu] = \frac{\text{Cov}(\ell, f_p)}{\bar{f}_p} = \frac{\text{Cov}(\ell, f) - \text{Cov}(\ell, g)}{\bar{f} - \bar{g}}$$

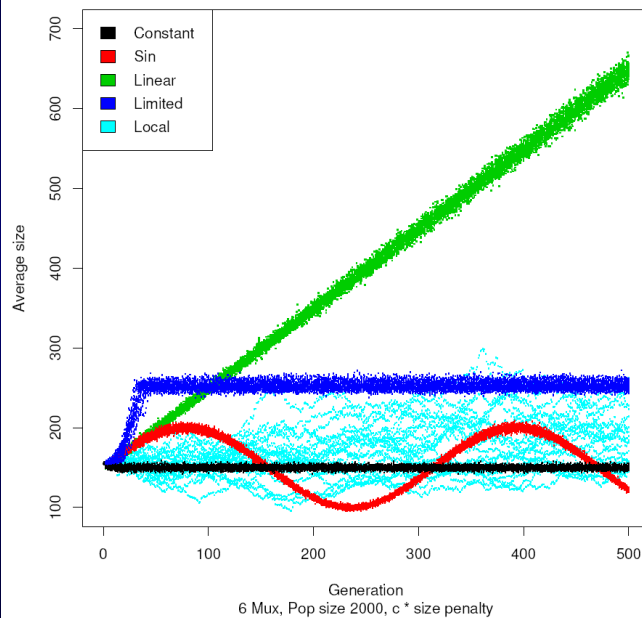
- Assume

$$g(\ell(x), t) = c(t)\ell(x)$$

- No bloat if

$$c(t) = \frac{\text{Cov}(\ell, f)}{\text{Var}(\ell)}$$

Avg size vs. time, different target size functions



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, **formal theories explain and predict** the behaviour of 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
- So, theory can **help design competent algorithms**

Take-home messages

- Bloat (explained and tamed!)
- Halting problem (solved!)
- NFL (beaten!)
- Convergence (guaranteed!)



Many thanks

Don't forget to get a look at...

