Evolution of Genetic Programming Populations

W. B. Langdon
Genetic Programming Group,
Dept. of Computer Science,
University College, London,
Gower Street, London WC1E 6BT, UK
Email: W.Langdon@cs.ucl.ac.uk
www: http://www.cs.ucl.ac.uk/staff/W.Langdon/
Tel: +44 (0) 171 380 7214, Fax: +44 (0) 171 387 1397

Keywords: population variety, diversity, genetic programming, Price's theorem, Fisher's theorem.

Abstract

We investigate in detail what happens as genetic programming (GP) populations evolve. Since we shall use the populations which showed GP can evolve stack data structures as examples, we start in Section 1 by briefly describing the stack experiment [Langdon, 1995]. In Section 2 we show Price's Covariance and Selection Theorem can be applied to Genetic Algorithms (GAs) and GP to predict changes in gene frequencies. We follow the proof of the theorem with experimental justification using the GP runs from the stack problem. Section 3 briefly describes Fisher's Fundamental Theorem of Natural Selection and shows in its normal interpretation it does not apply to practical GAs.

An analysis of the stack populations, in Section 4, explains that the difficulty of the stack problem is due to the presence of "deceptive" high scoring partial solutions in the population. These cause a negative correlation between necessary primitives and fitness. As Price's Theorem predicts, the frequency of necessary primitives falls, eventually leading to their extinction and so to the impossibility of finding solutions like those that are evolved in successful runs.

Section 5 investigates the evolution of variety in GP populations. Detailed measurements of the evolution of variety in stack populations reveal loss of diversity causing crossover to produce offspring which are copies of their parents. Section 6 concludes with measurements that show in the stack population crossover readily produces improvements in performance initially but later no improvements at all are made by crossover.

Section 7 discusses the importance of these results to GP in general.

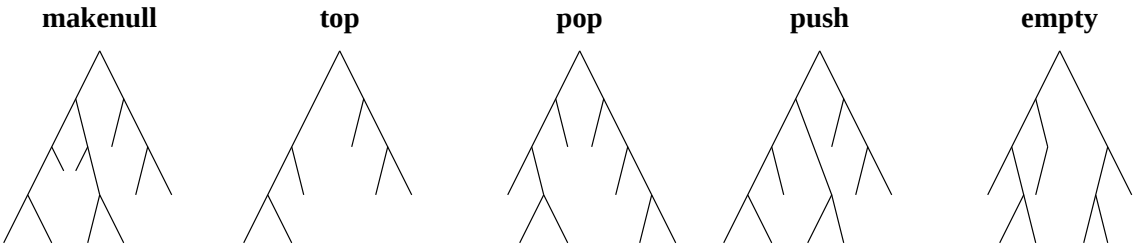

Figure 1: One Individual – One Program: Five Operations – Five Trees

1 Architecture of Stack Individuals

Before going into the details of the evolution of variety in the stack populations, this section re-caps the basic multi-tree architecture used in [Langdon, 1995] to evolve a single program which implements five actions required of a stack data structure, initialise (*makenull*), read *top* of the stack, *pop* the top of the stack and return its value, *push* an integer onto the stack and and test to see if the stack is *empty* or not.

Each evolved program must implement all five actions. This is represented in the chromosome by allocating an evolvable tree per action. When the program is used, e.g. during its fitness testing, then the tree corresponding to the desired action is called. I.e. each individual within the population is composed of five trees, see Figure 1.

This multiple tree architecture was chosen so that each tree contains code which has evolved for a single purpose. It was felt that this would ease the formation of "building blocks" of useful functionality and enable crossover, or other genetic operations, to assemble working implementations of the operations from them. Similarly complete programs could be formed whilst each of its trees improved.

The genetic operations, reproduction, crossover and mutation are redefined to cope with this multi-tree architecture. We define the genetic operations to act upon only one tree at a time. The other trees are unchanged and are copied directly from the first parent to the offspring. Genetic operations are limited to a single tree at a time in the expectation that this will reduce the extent to which they disrupts "building blocks" of useful code. Crossing like trees with like trees is similar to the crossover operator with "branch typing" used by Koza in most of his experiments involving ADFs in [Koza, 1994].

In the case of reproduction, the only action on the chosen tree is also to copy it, in other words each new individual is created by copying all trees of the parent program.

When crossing over, one type of tree is selected (at random, with equal probability, i.e. 1/5). This tree in the offspring is created by crossover between the trees in each parent of the chosen type in the normal GP way [Koza, 1992] (see Figure 2). In the stack

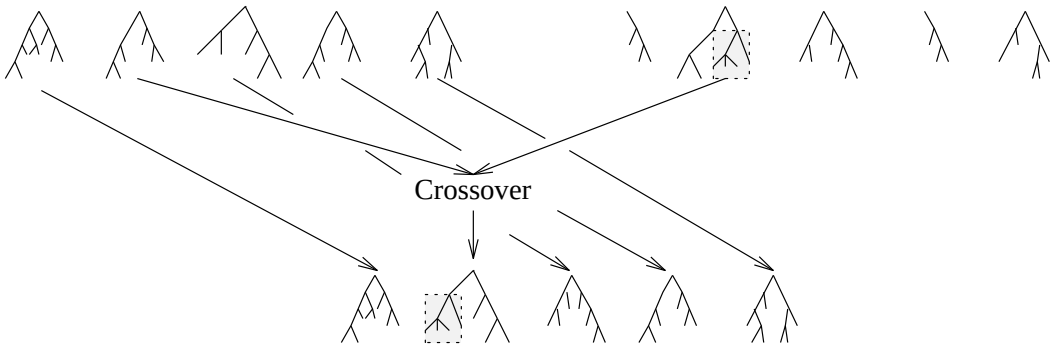

Figure 2: Crossover in One Tree at a time

Table 1: Tableau for Evolving a Stack

Objective	To evolve a pushdown stack	
Architecture	Five separate trees	
Primitives	+, −, 0, 1, max, arg1, aux, inc_aux, dec_aux, read, write, write_Aux	
Fitness Case	4 test sequences, each of 40 tests	
Fitness Scaling	1.0 for each test passed	
Selection	Scalar tournament of 4	
Hits	n/a	
Wrapper	makenull	result ignored
	top	no wrapper
	pop	no wrapper
	push	result ignored
	empty	result $> 0 \Rightarrow$ TRUE, otherwise FALSE
Parameters	Population = 1000, G=101, program size <= 250	
Success Predicate	Fitness >= 160.0	

experiments, all trees have identical primitives, c.f. Table 1.

1.1 Stack Primitives

Primitives like those a human programmer might use, were chosen. Firstly this ensures a solution is possible, i.e. a program which solves the problem can be written using only these primitives. (The need for the available primitives to be powerful enough so that a solution to the problem can be express using them is called the *sufficiency* requirement [Koza, 1992, page 86]). Secondly as some constructs are useful to human programmers it was expected that corresponding primitives might be useful to the GP. For example primitives were included that aid maintenance of a stack pointer, although their functionality could in principle be evolved using combinations of the other primitives.

The following primitives were available to the GP:

- arg1, the value to be pushed on to the stack. When arg1 is used by any of the

operations except push it has the value zero. Evolving programs can read arg1 but they can not change it.

- arithmetic operators + and −.
- constants 0, 1 and the maximum depth of the stack, max (which has the value 10).
- indexed memory functions read and write.
- primitives to help maintain a stack pointer; aux, inc_aux, dec_aux and write_Aux.

1.2 Indexed Memory

63 integer memory cells (numbered $-31 \ldots 31$) were available.

1.3 Register

In addition to the indexed memory [Teller, 1994] a single auxiliary variable "aux" was provided which, like each addressable memory cell, is capable of storing a single 32-bit signed integer. The motivation for including it and the primitives that manipulate it was that it could be used as a stack pointer, holding addresses to be used with the index memory. However, as with all the other primitives, the GP is not forced to use it in any particular way or even use it at all.

There are four associated primitives:

1. aux, which evaluates to its current value.
2. inc_aux, which increases the current value by one and returns the new value.
3. dec_aux, which decreases the current value by one and returns the new value.
4. write_Aux, which evaluates its argument and sets aux to this value. It behaves like write in that it returns the original value of aux rather than the new one.

2 Price's Selection and Covariance Theorem

Price's Covariance and Selection Theorem [Price, 1970] from population genetics relates the change in frequency of a gene in a population from one generation to the next, to the covariance of the gene's frequency in the original population with the number of offspring produced by individuals in that population (see Equation 1). The theorem holds "for a single gene or for any linear combination of genes at any number of loci, holds for any sort of dominance or epistasis (non-linear interaction between genes), for sexual or asexual

reproduction, for random or non-random mating, for diploid, haploid or polyploid species, and even for imaginary species with more than two sexes" [Price, 1970]. In particular it applies to genetic algorithms (GAs) [Altenberg, 1994].

$$\Delta Q = \frac{\mathrm{Cov}(z, q)}{\overline{z}} \tag{1}$$

Q = Frequency of given gene (or linear combinations of genes) in the population
ΔQ = Change in Q from one generation to the next.
q_i = Frequency of gene in the individual i (more information is given in Section 2.1.
z_i = Number of offspring produced by individual i.
$\overline{z}$ = Mean number of children produced.
Cov = Covariance

2.1 Proof of Price's Theorem

In this section we follow the proof of Price's Theorem given in [Price, 1970] (which assumes sexual reproduction) and show it applies to Genetic Algorithms (GAs) [Holland, 1992] in general and to genetic programming (GP) [Koza, 1992] in particular. In the next section (2.2), we extend the proof to cover asexual reproduction. This more general proof also applies to Genetic Algorithms, including GAs with asexual reproduction (i.e. copying and mutation). Firstly we define the additional symbols we shall use.

P_1 = Initial population
P_2 = Population at next generation (for purposes of the proof generations are assumed to be separated)
N = Size of initial population.
n_z = "Zygotic ploidy of the species for the gene". E.g. in natural species n_z may be 2, i.e. the gene can exist on two chromosomes.
In traditional GAs chromosomes are not paired so n_z is 1. In GP there is still only one chromosome but the same gene (primitive) can occur multiple times within it. For GP we define n_z to be unity.
g_i = Number of copies of gene in individual i
q_i = Frequency of gene in the individual i. That is the number of times the gene appears in individual i divided by the "zygotic ploidy" of the species for the gene (i.e. 1 if haploid, 2 if diploid).

$$q_i = g_i / n_z$$

When n_z is unity (e.g. most GAs and GP) q_i becomes the number of copies of the gene in individual i (i.e. $q_i = g_i$). So gene frequencies are defined to be relative to number of individuals in the population rather than per available loci.
$\overline{q}$ = Arithmetic mean of q_i in population P_1

Q_1 = Frequency of given gene (or linear combinations of genes) in the population. I.e. number of copies of gene in population divided by the number of chromosomes it could occupy.

Q_2 = Frequency of gene in population P_2

n_G = "Gamete ploidy for the gene". In natural species n_G is typically 1, i.e. the gene can exist on one chromosome in the gamete (germ cell).
In traditional GAs there is no separate germ cell and whether the chromosome fragment can contain the gene depends upon whether the locus of the gene is present in the fragment or not.
In GP there is still only one chromosome but there are no fixed loci and the same gene (primitive) can occur multiple times within a crossover fragment.

z_i = Number of offspring produced by individual i. Note this is the same as the number of successful gametes it produces. (In GA terminology the number of chromosome fragments produced from i which occur in individuals in the next population).

$\overline{z}$ = Mean number of children produced.

$g_i^{'}$ = Number of copies of the gene in all the successful gametes produced by individual i.
In traditional linear chromosome GAs, $g_i^{'}$ is the number of chromosome fragments copied from individual i that are passed to the next generation which contain the gene's location and where the location contains the gene. (NB the value at the gene's location has not been changed by mutation).
If a traditional GA, with zero mutation rate, the expected value of $g_i^{'}$ is $z_i/2$. With mutation $g_i^{'}$ is reduced proportionately to the gene mutation rate.
In GP, $g_i^{'}$ is the number of copies of the gene that are copied from i and passed to the next generation.

$q_i^{'}$ = Frequency of gene in the offspring produced by individual i. Defined by

$$\begin{aligned} q_i^{'} &= \frac{g_i^{'}}{z_i n_G} \quad \text{, if } z_i \neq 0 \\ &= q_i \quad \text{, otherwise} \end{aligned}$$

$\Delta q_i = q_i^{'} - q_i$

Proof of Price's Theorem with Sexual Reproduction

We shall start with the frequency of the gene in the current population, Q_1. Then find the frequency in the subsequent generation, Q_2. Subtracting them yields the change in frequency, which we shall simplify to give Price's Theorem.

$$\begin{aligned} Q_1 &= \frac{\sum g_i}{n_z N} \\ &= \frac{\sum n_z q_i}{n_z N} \\ &= \overline{q} \end{aligned}$$

Each individual in the new population is created by joining one or more "gametes" (in GAs and GP by joining crossover fragments) and the number of each gene in the individual is the sum of the number in each of the gametes from which it was formed. Thus the number of genes in the new population is equal to the number in the successful gametes produced by the previous generation.

Similarly the number of chromosomes in an individual is the sum of the number in each of the gametes which formed it, n_G. Thus if n_G is the same in all cases:

$$
\begin{aligned}
Q_2 &= \frac{\sum g_i^{'}}{\sum z_i n_G} && (2)\\
&= \frac{\sum z_i n_G q_i^{'}}{\sum z_i n_G} \\
&= \frac{\sum z_i q_i^{'}}{N\overline{z}} && (3)\\
&= \frac{\sum z_i q_i}{N\overline{z}} + \frac{\sum z_i \Delta q_i}{N\overline{z}} \\
&= \frac{\sum ((z_i - \overline{z})(q_i - \overline{q}) + \overline{z}\, q_i + z_i \overline{q} - \overline{z}\,\overline{q})}{N\overline{z}} + \frac{\sum z_i \Delta q_i}{N\overline{z}} \\
&= \frac{\frac{1}{N}\sum (z_i - \overline{z})(q_i - \overline{q}) + \overline{z}\frac{1}{N}\sum q_i + \overline{q}\frac{1}{N}\sum z_i - \frac{1}{N}\sum \overline{z}\,\overline{q}}{\overline{z}} + \frac{\sum z_i \Delta q_i}{N\overline{z}} \\
&= \frac{\frac{1}{N}\sum (z_i - \overline{z})(q_i - \overline{q}) + \overline{z}\,\overline{q} + \overline{q}\,\overline{z} - \overline{z}\,\overline{q}}{\overline{z}} + \frac{\sum z_i \Delta q_i}{N\overline{z}} \\
&= \frac{\frac{1}{N}\sum (z_i - \overline{z})(q_i - \overline{q}) + \overline{q}\,\overline{z}}{\overline{z}} + \frac{\sum z_i \Delta q_i}{N\overline{z}} \\
&= \frac{\mathrm{Cov}(z,q)}{\overline{z}} + \overline{q} + \frac{\sum z_i \Delta q_i}{N\overline{z}} \\
\Delta Q &= \frac{\mathrm{Cov}(z,q)}{\overline{z}} + \frac{\sum z_i \Delta q_i}{N\overline{z}}
\end{aligned}
$$

"If meiosis and fertilization are random with respect to the gene, the summation term at the right will be zero except for statistical sampling effects ('random drift'), and these will tend to average out to give equation 1." I.e. the expected value of $\sum z_i \Delta q_i$ is zero.

So while survival of an individual and the number of children it has may be related to whether it carries the gene, it is assumed that the production of gametes (crossover fragments) and their fusing to form offspring is random. In GA terms selection for reproduction is dependent upon fitness and in general dependent on the presence of specific genes but selection of crossover points is random and so independent of genes (Section 2.4 discusses this further for GPs).

2.2 Proof of Price's Theorem with Asexual Reproduction

The proof of Price's theorem given in [Price, 1970] (reproduced above) assumes sexual reproduction. For it to be applied to GAs and GP it needs to be extended to cover asexual reproduction (i.e. copying and mutation). Before doing so, we define further symbols we shall use.

$g'_{a\,i}$ = Number of copies of the gene in the offspring created asexually by individual i.

$g'_{x\,i}$ = Number of copies of the gene in all the successful gametes (n.b. sexual reproduction) produced by individual i.

a_i = Proportion of offspring of individual i created asexually (in GAs mutation or direct copying).

$$a_i = g'_{a\,i}/g'_i$$

$\overline{a}$ = $\sum a_i z_i / N\,\overline{z}$

x_i = Proportion of offspring of individual i created sexually, i.e. by crossover.

$$x_i = g'_{x\,i}/g'_i$$

$\overline{x}$ = $\sum x_i z_i / N\,\overline{z}$

$q'_{a\,i}$ = Frequency of gene in the offspring produced asexually by individual i. Defined by

$$\begin{aligned} q'_{a\,i} &= \frac{g'_{a\,i}}{a_i z_i n_z} \quad \text{, if } a_i z_i \neq 0 \\ &= q_{a\,i} \quad \text{, otherwise} \end{aligned}$$

$q'_{x\,i}$ = Frequency of gene in the offspring produced sexually by individual i. Defined by

$$\begin{aligned} q'_{x\,i} &= \frac{g'_{x\,i}}{x_i z_i n_G} \quad \text{, if } x_i z_i \neq 0 \\ &= q_{x\,i} \quad \text{, otherwise} \end{aligned}$$

So Equation 2 becomes

$$\begin{aligned} Q_2 &= \frac{\sum g'_{a\,i} + g'_{x\,i}}{\sum a_i z_i n_z + x_i z_i n_G} \\ &= \frac{\sum a_i z_i n_z q'_{a\,i} + x_i z_i n_G q'_{x\,i}}{\sum a_i z_i n_z + x_i z_i n_G} \end{aligned}$$

If reproduction type (sexual or asexual) is independent of the gene then the expected values of the gene frequencies, $q'_{a\,i}$ and $q'_{x\,i}$ will be equal (and equal to q'_i) and so in large populations

$$\begin{aligned} Q_2 &= \frac{\sum a_i z_i n_z q'_i + x_i z_i n_G q'_i}{\sum a_i z_i n_z + x_i z_i n_G} \\ &= \frac{\sum a_i z_i n_z q'_i + x_i z_i n_G q'_i}{N\,\overline{a}\,\overline{z}\,n_z + N\,\overline{x}\,\overline{z}\,n_G} \end{aligned}$$

If reproduction type is independent of the gene then in large populations

$$\begin{aligned} Q_2 &= \frac{\sum \overline{a} z_i n_z q'_i + \overline{x} z_i n_G q'_i}{N\,\overline{z}(\overline{a}\,n_z + \overline{x}\,n_G)} \\ &= \frac{\sum z_i q'_i}{N\,\overline{z}} \end{aligned}$$

The rest of the proof (i.e. from Equation 3 onwards) follows.

2.3 Price's Theorem for Genetic Algorithms

Where the population size is unchanged, as is usually the case in GAs and GP (and two parents are required for each individual created by crossover), $\overline{z} = p_r + p_m + 2p_c$ (where p_r = copy rate, p_m = mutation rate and p_c is the crossover rate. Since $p_r + p_m + p_c = 1$, the mean number of children $\overline{z} = 1 + p_c$ and Equation 1 becomes:

$$\Delta Q = \frac{\mathrm{Cov}(z, q)}{1 + p_c} \tag{4}$$

2.4 Applicability of Price's Theorem to GAs and GPs

The simplicity and wide scope of Price's Theorem has lead Altenberg to suggest that covariance between parental fitness and offspring fitness distribution is fundamental to the power of evolutionary algorithms. Indeed [Altenberg, 1995] shows Holland's schema theorem [Holland, 1973; Holland, 1992] can be derived from Price's Theorem. This and other analysis, leads [Altenberg, 1995, page 43] to conclude "the Schema Theorem has no implications for how well a GA is performing".

While the proof in [Price, 1970] assumes discrete generations the result "can be applied to species with overlapping, inter-breeding generations". Thus the theorem can be applied to steady state GAs [Syswerda, 1989; Syswerda, 1991] such as used in [Langdon, 1995].

For the theorem to hold the genetic operations (crossover and mutation in GA terms) must be independent of the gene. That is on average there must be no relationship between them and the gene. In large populations random effects will be near zero on average but in smaller populations their effect may not be negligible. In GAs selection of crossover and mutation points is usually done independently of the contents of the chromosome and so Price's theorem will hold (except in small GA populations where random fluctuations may be significant). In GP populations are normally bigger (and the number of generations similar) so random effects, "genetic drift", are less important.

In standard GP it is intended that the genetic operators should also be independent, however in order to ensure the resultant offspring are syntactically correct and not too big, genetic operators must consider the chromosome's contents. This is normally limited to just its structure in terms of tree branching factor (i.e. the number of arguments a function has) and tree depth or size limits. That is, they ignore the actual meaning of a node in the tree (e.g. whether it is MUL or ADD) but do consider how many arguments it has. Thus a function with two arguments (e.g. MUL) and a terminal (e.g. max) may be treated differently.

It is common to bias the choice of crossover points in favour of internal nodes (e.g. in these GP experiments internal points in program trees are deliberately chosen 30% of the time, the other 70% are randomly chosen through the whole tree. [Koza, 1992, page 114] weights internal nodes to external nodes 9:1, while [Angeline, 1996, page 27] argues "that no one constant value for leaf frequency is optimal for every problem"). This reduces the proportion of crossover fragments which contain only a single terminal. Once again the genetic operators ignore the meaning of nodes within the tree.

In a large diverse population these factors should have little effect and Price's Theorem should hold. However when many programs are near the maximum allowed size a function which has many arguments could be at a disadvantage since the potential offspring containing it have a higher chance of exceeding size limits. Therefore restrictions on program size may on average reduce the number of such functions in the next generation compared to the number predicted by considering only fitness (i.e. by Price's Theorem). [Altenberg, 1994, page 47] argues Price's theorem can be applied to genetic programming and we shall show experimental evidence for it based on genes composed of a single GP primitive.

2.5 Application of Price's Theorem to the GP Stack Problem

In this section we experimentally test Price's Theorem by comparing its predictions with what actually happened using GP populations from the 60 runs of the stack problem described in [Langdon, 1995]. Firstly we consider the change in numbers of a single primitive and then we examine the change in frequency versus fitness for all primitives in a typical and in a successful run.

In GAs the expected number of children each individual has is determined by its fitness. On average the expected number is equal to the actual number of offspring z (as used in Price's theorem, i.e. in Equations 1 and 4). For example when using roulette wheel selection the expected number of children is directly proportional to the parent's fitness. When using tournament selection (as in [Langdon, 1995]) the expected number of children is determined by the parent's rank within the population and the tournament size. The remainder of this section uses the expected number of offspring as predicted by the parents fitness ranking within the current population in place of z.

Price's theorem predicts the properties of the next generation. In a steady state population it can be used to predict the average rate of change. However in general subsequent changes to the population will change the predicted rate of change. For simplicity we assume that during one generation equivalent (i.e. the time taken to create as many new

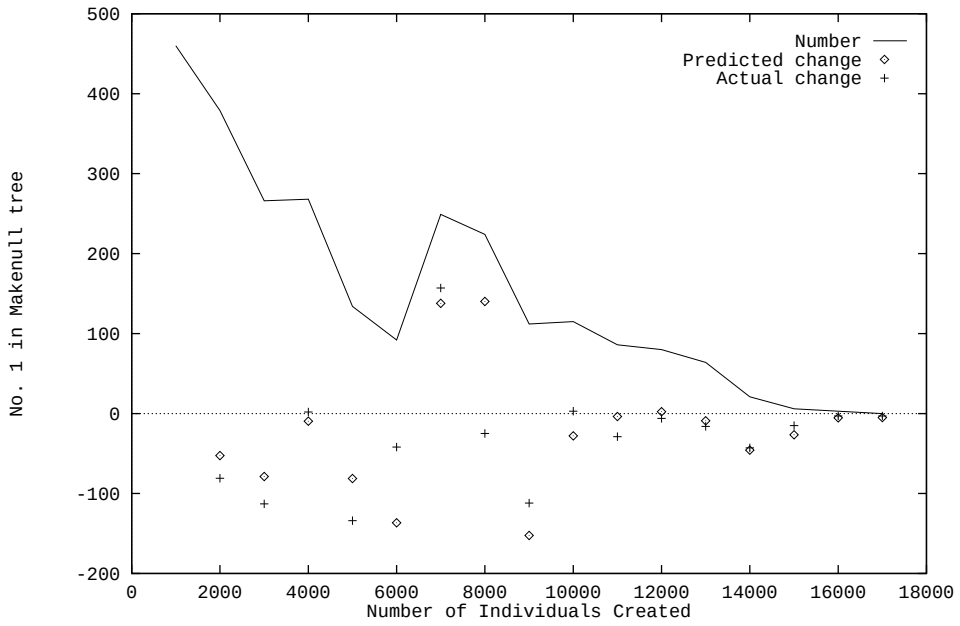

Figure 3: Evolution of the number of the terminal "1" in the makenull tree plus predicted change and actual change in next generation, in typical stack (51) run.

individuals as there are in the population) such effects are small and base the predicted properties of the new population on linear extrapolation using the predicted rate of change.

The 60 runs of the stack problem use identical parameters and differ only in the initial seed used by the [Park and Miller, 1988] pseudo random number generator. For convenience individual runs are numbered (1) to (60).

The solid line in Figure 3 plots the evolution of the number of a particular primitive in a particular tree in the population for a typical run. (As there is no crossover between trees of different types, primitives of the same type but in different trees are genetically isolated from each other and so Equation 4 can be applied independently to each tree). The change from one generation equivalent to the next is plotted by crosses which show good agreement with the change predicted by linearly extrapolating the rate of change predicted by Price's theorem. Some discrepancy between the actual change and the predicted change is expected due to "noise". That is the number of children an individual has is a stochastic function of its fitness (see Figure 8). However non-random deviations from the prediction are to be expected as linear extrapolation assumes the rate of change will not change appreciably in the course of one generation equivalent (such as happens at generations 6 and 8).

Figures 4 to 7 plot the covariance of primitive frequency with normalised fitness against the change in the primitives frequency in the subsequent generation (equivalent). While

these plots show significant differences from the straight line predicted by Equation 4, least squares regression yields best fit lines which pass very close to the origin but (depending upon run and primitive) have slopes significantly less than $1 + p_c = 1.9$ (they lie in the range 1.18 to 1.79, see Table 2).

Random deviations from the theory are expected but should have negligible effect when averaged by fitting the regression lines. The fact that regression coefficients differ from 1.9 is explained by the fact that we are recording changes over a generation, during this time it is possible for the population to change significantly. We would expect this effect to be most noticeable for primitives with a high rate of change since these effect the population! A high rate of change may not be sustainable for a whole generation and so the actual change will be less than predicted by extrapolating from its initial rate of change. However large changes have a large effect on least squares estimates so these outliers can be expected to reduce the slope of the regression line.

Regression coefficients can be calculated after excluding large values leaving only the smaller changes. However this makes the calculation dependent on small values with high noise. This may be exacerbated if the primitive quickly became extinct as there are few data points left. (When considering a typical run (51) of the stack problem and excluding covariances outside the range $-0.1 \ldots +0.1$ regression coefficients were often effected by this noise and lie in the range $-0.96 \ldots 6.28$ for the twelve primitives in the empty tree).

In conclusion Price's Theorem gives quantitative predictions of the short term evolution of practical GP populations, however such predictions are effected by sampling noise in finite populations and may be biased if predictions are extrapolated too far in rapidly evolving populations. The theorem can also be used to explain the effects of fitness selection on GP populations.

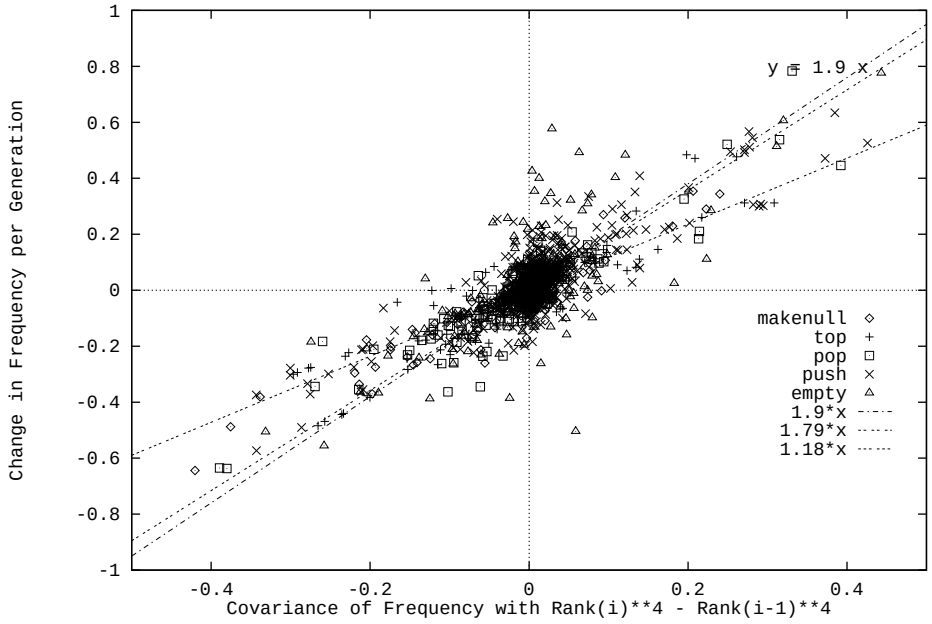

Figure 4: Covariance of Primitive frequency and $\left(\frac{R_i}{N}\right)^4 - \left(\frac{R_{i-1}}{N}\right)^4$ v. change in frequency in next generation, in typical stack (51) run. Data collected every generation equivalent.

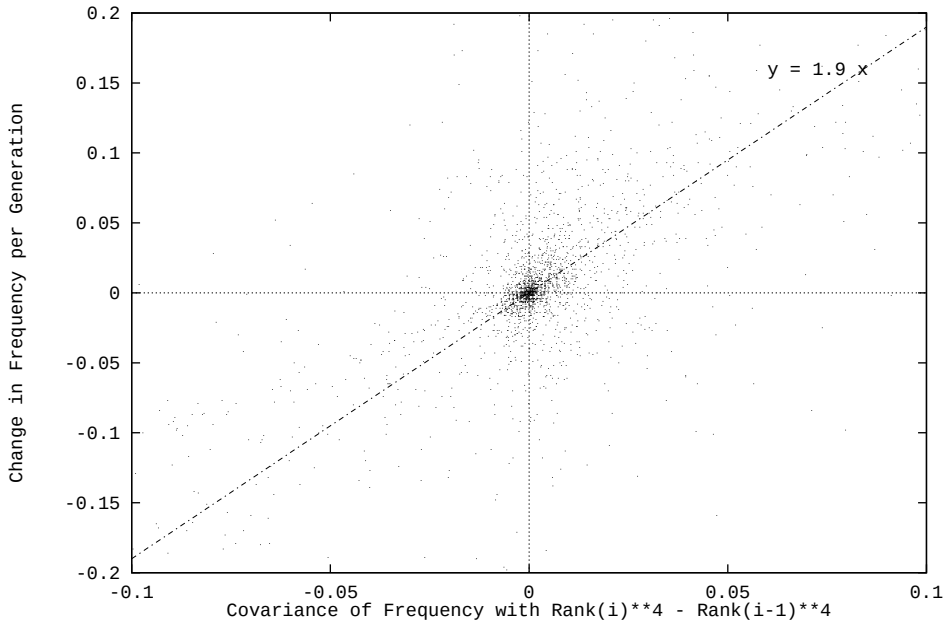

Figure 5: Covariance of Primitive frequency and $\left(\frac{R_i}{N}\right)^4 - \left(\frac{R_{i-1}}{N}\right)^4$ v. change in frequency in next generation, in typical stack (51) run. Only data near the origin shown.

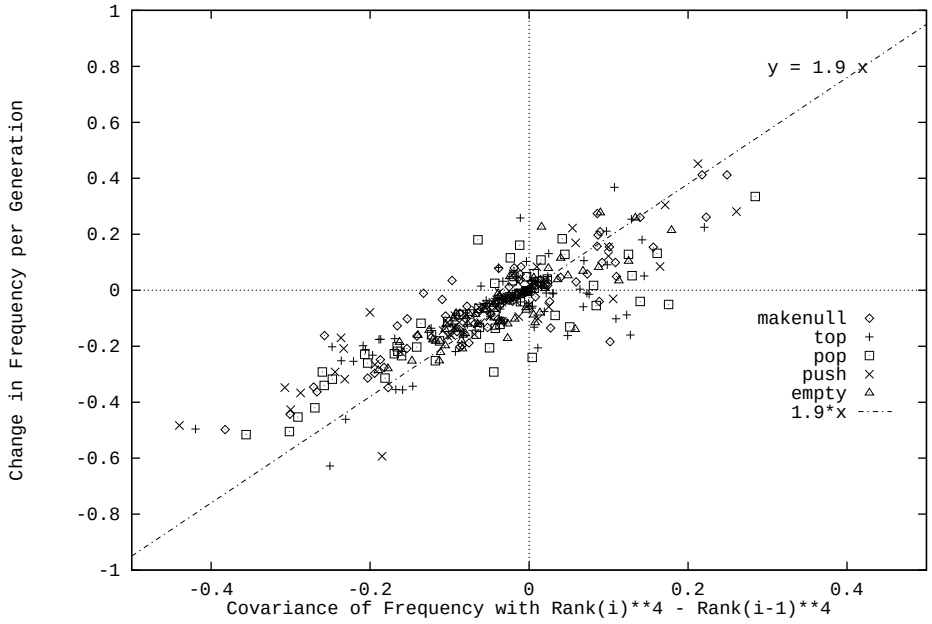

Figure 6: Covariance of Primitive frequency and $\left(\frac{R_i}{N}\right)^4 - \left(\frac{R_{i-1}}{N}\right)^4$ v. change in frequency in next generation, in successful stack (2) run. Data collected every generation equivalent.

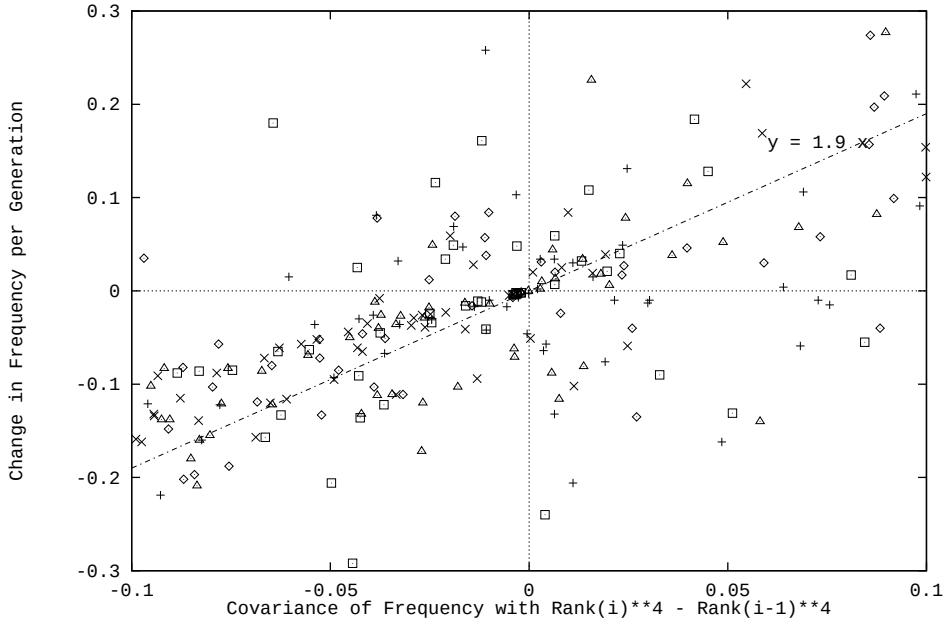

Figure 7: Covariance of Primitive frequency and $\left(\frac{R_i}{N}\right)^4 - \left(\frac{R_{i-1}}{N}\right)^4$ v. change in frequency in next generation, in successful stack (2) run. Data near origin.

Table 2: Least squares regression coefficients of covariance of primitive frequency and $\left(\frac{R_i}{N}\right)^4 - \left(\frac{R_{i-1}}{N}\right)^4$ with change in frequency in the next generation for a typical (51) stack run.

Primitive	makenull	top	pop	push	empty
	Δ Frequency – Intercept : Gradient				
ADD	-0.026 : 1.26	-0.007 : 1.39	-0.007 : 1.33	0.035 : 1.34	0.010 : 1.78
SUB	-0.017 : 1.21	-0.016 : 1.44	0.006 : 1.75	-0.017 : 1.30	0.006 : 1.83
0	-0.001 : 1.35	0.002 : 1.46	0.011 : 1.50	0.031 : 1.34	-0.002 : 1.41
1	-0.015 : 1.18	-0.001 : 1.34	-0.018 : 1.17	-0.003 : 1.76	-0.002 : 1.24
max	0.001 : 1.52	-0.017 : 1.34	-0.008 : 1.44	0.007 : 1.73	-0.009 : 1.79
arg1	0.000 : 1.50	-0.008 : 1.60	0.018 : 1.74	0.012 : 1.39	0.001 : 1.17
aux	-0.025 : 1.20	0.003 : 1.61	-0.004 : 1.31	0.004 : 1.37	-0.024 : 1.29
inc_aux	0.006 : 1.38	0.004 : 1.67	-0.002 : 1.49	-0.011 : 1.50	-0.012 : 1.19
dec_aux	-0.002 : 1.51	-0.001 : 1.40	-0.005 : 1.72	0.004 : 1.26	-0.001 : 1.40
read	-0.020 : 1.21	-0.002 : 1.40	-0.015 : 1.71	0.009 : 1.38	-0.037 : 1.54
write	-0.003 : 1.42	-0.002 : 1.30	0.008 : 1.46	0.015 : 1.30	-0.038 : 1.54
write_Aux	-0.001 : 1.30	-0.011 : 1.20	-0.011 : 1.58	0.011 : 1.39	0.049 : 1.33

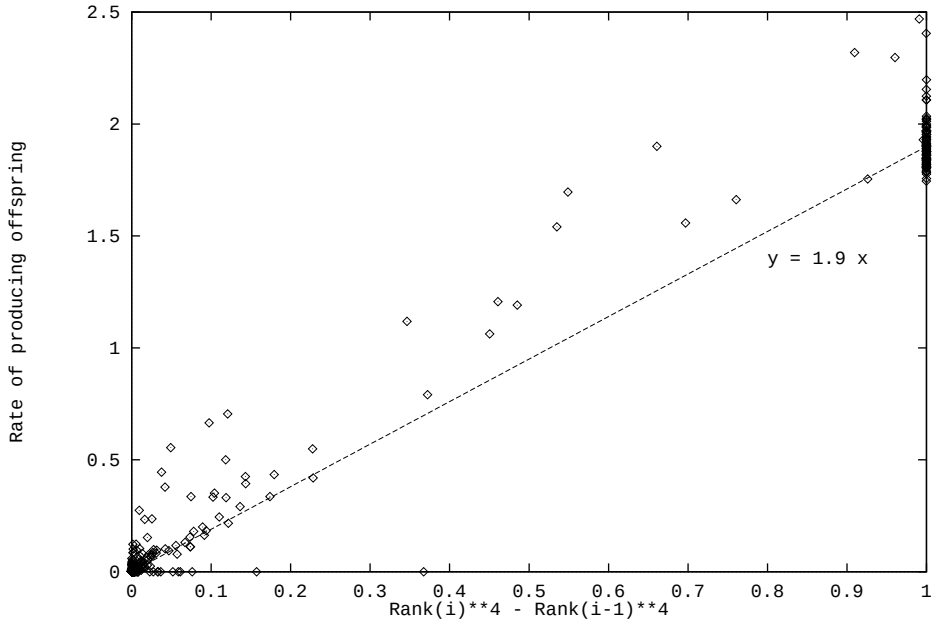

Figure 8: Rate of producing offspring v. $\left(\frac{R_i}{N}\right)^4 - \left(\frac{R_{i-1}}{N}\right)^4$ in typical stack (51) run. Data collected every generation equivalent.

3 Fisher's Fundamental Theorem of Natural Selection

Fisher's fundamental theorem of natural selection states "The rate of increase in fitness of any organism at any time is equal to its genetic variance in fitness at that time" [Fisher, 1958, page 37]. "Under the usual interpretation the theorem is believed to say that the rate of increase in the mean fitness of a population is equal to the population's additive variance for fitness". Since the variance can never be negative "natural selection causes a continual increase in mean fitness of a population. This interpretation of the theorem is only true when the population mates randomly and there is no dominance or epistasis" [Frank, 1995, page 382].

An example of this usage is given in [Tackett, 1995, page 289] which claims "According to Fisher's fundamental theory of natural selection the ability of a population to increase in fitness is proportional to the variance in fitness of the population members."

We would certainly expect epistasis (non-linear interaction between genes) to occur in most GAs and so would not expect this interpretation of the theorem to hold. Figure 9 shows the evolution of a stack population's fitness for one run. The error bars indicate a standard deviation either side of the mean population fitness. From Figure 9 we can see the standard deviation through out the bulk of the run is consistently close to 20, i.e. the variance of the population's fitness is near 400 (20×20). The usual interpretation of Fisher's theorem predicts the mean fitness will continually increase but obviously this is not the case as it remains fairly constant throughout the run and even falls occasionally.

We conclude that under the usual interpretation Fisher's theorem does not normally apply to GAs. This is important because this interpretation of Fisher's theorem has been used as an argument in favour of GA selection schemes which produce a high variance in population fitness [Tackett, 1995, pages 272 and 290]. (There may be other reasons for preferring these selection methods. A high fitness variance may indicate a high degree of variation in the population, which might be beneficial).

[Price, 1972] makes the point that Fisher's publications on his fundamental theorem of natural selection "contains the most confusing published scientific writing I know of" [page 134] leading to "forty years of bewilderment about what he meant" [page 132]. [Price, 1972] and [Ewens, 1989; Ewens, 1992b; Ewens, 1992a] argue that the usual interpretation of Fisher's theorem is incorrect and his "fitness" should be considered as just the component of fitness which varies linearly with gene frequency. All other effects, such as "dominance, epistasis, population pressure, climate, and interactions with other species – he regarded as a matter of the environment" [Price, 1972, page 130]. Price and Ewens both give proofs

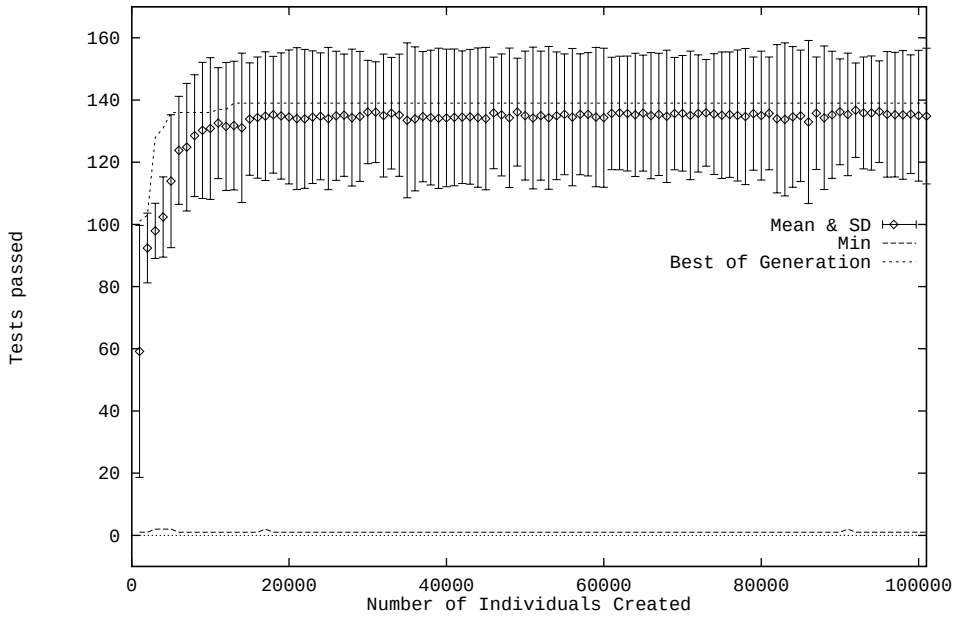

Figure 9: Evolution of Fitness in a typical stack run (51)

for this interpretation of Fisher's theorem but conclude that it is "mathematically correct but less important than he thought it to be" [Price, 1972, page 140].

4 Evolution of Stack Problem Populations

In this section we return to the stack problem of [Langdon, 1995] and investigate why most runs failed to find a solution. Investigation of the evolved solutions shows which primitives are essential to the correct operation of all the evolved solutions and in most runs one or more of these becomes extinct, thus preventing the evolution of a solution like those found. The loss of these primitive is explained using Price's Theorem by the negative covariance of their frequency with their fitness. Similar covariances are found in successful runs and we conclude success requires a solution to be found quickly, before extinction of critical primitives occurs.

Table 3 contains an entry for each of the five program trees (which each trial stack data structure comprises) and the primitives that the tree can use (see Section 1 and Table 1). Where the primitive is essential to the operation of one of the four stack solutions found, the entry contains the number(s) of the solutions. If the primitive is not essential to the correct operation of any of the four evolved solutions (in the particular tree) the entry is blank. Primitives ADD and max are omitted as they are always blank. (The essential primitives are shown within shaded boxes in Figures 10, 11, 12 and 13. NB in the stack

Table 3: Primitives Essential to the Operation of Evolved Stack Programs

Tree/Primitive Essential to Evolved Stack Solutions										
Tree	SUB	0	1	arg1	aux	inc _aux	dec _aux	read	write	write _Aux
makenull	4	4	1 2 3 4							1 2 3 4
top					1 2 3			1 2 3 4		1 4
pop					1 2 4	1 2 3	4	3	1 2 4	3
push				1 2 3 4		4	1 2 3		1 2 3 4	
empty	4	4			1 3 4					2

Table 4: Stack Primitives Essential to All Evolved Solutions

Tree	Primitive	Lost
makenul	1	14
makenul	write_Aux	7
top	read	21
push	arg1	6
push	write	29

Tree	Alternative Primitives	Both Lost
top	aux or write_Aux	12
pop	inc_aux or dec_aux	27
pop	read or write	15
push	inc_aux or dec_aux	40
empty	aux or write_Aux	9

problem each tree can use all of the primitives).

From Table 3 we can identify five primitives which are essential to the operation of all four evolved solutions and five pairs of primitives where one or other is required. These are shown in the two halves of Table 4 together with the number of runs where they were removed from the population by 21 generation equivalents (i.e. by the point where all four solutions had evolved).

After the equivalent of 21 generations in 43 of 60 runs, the number of one or more of the tree-primitives shown in the left had side of Table 4 had fallen to zero. That is the population no longer contained one or more primitives required to evolve a solution (like the solutions that have been found). In 12 of the remaining 17 populations both of one or more of the pairs of primitives shown on the right hand side of Table 4 had been removed from the population. Thus by generation 21 in all but 5 of 60 runs, the population no longer contained primitives required to evolve solutions like those found. In four of these five cases solutions were evolved (in the remaining case one of the essential primitives was already at a low concentration, which fell to zero by the end of the run at generation 101).

Figure 14 shows the evolution of six typical stack populations (runs 00, 10, 20, 30, 40

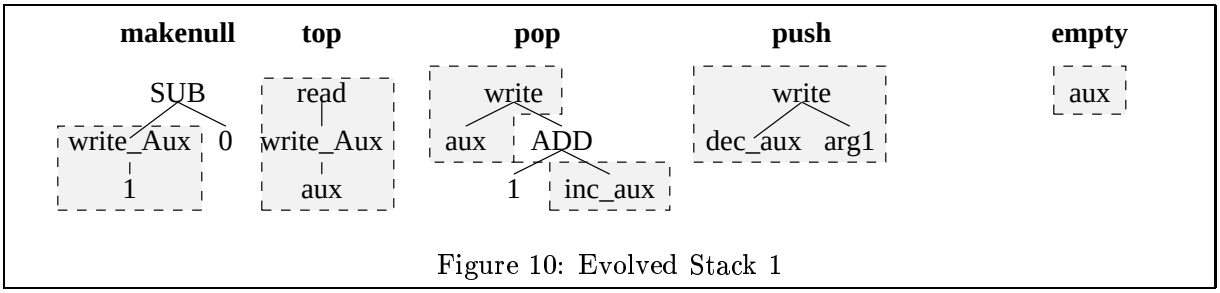

Figure 10: Evolved Stack 1

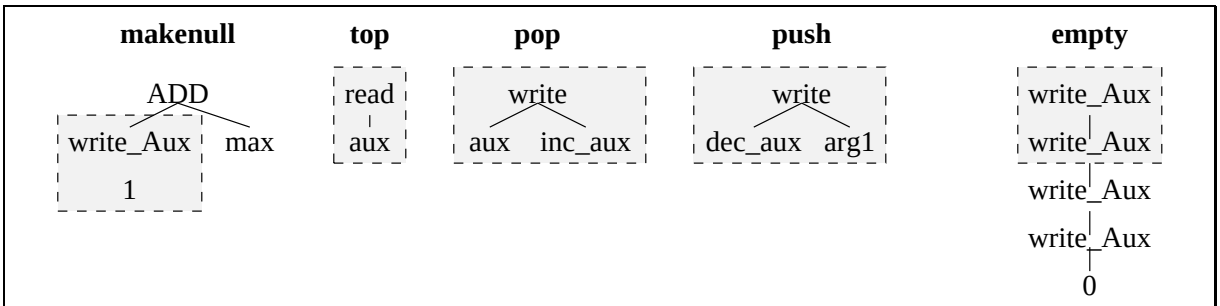

Figure 11: Evolved Stack 2

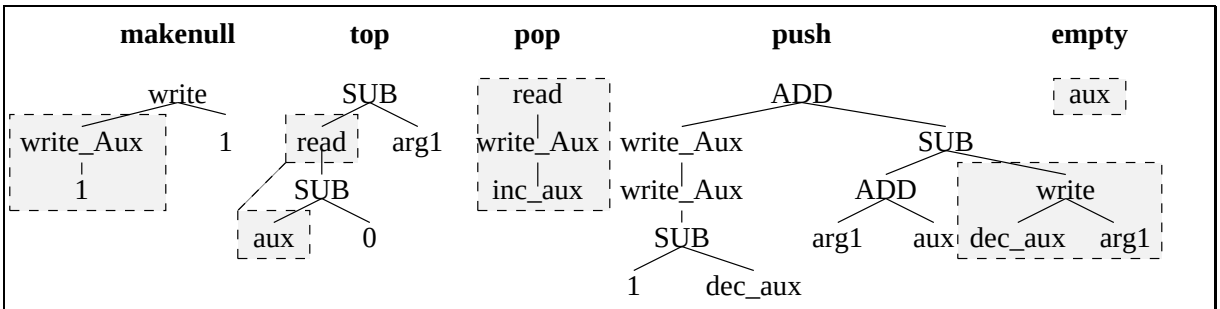

Figure 12: Evolved Stack 3

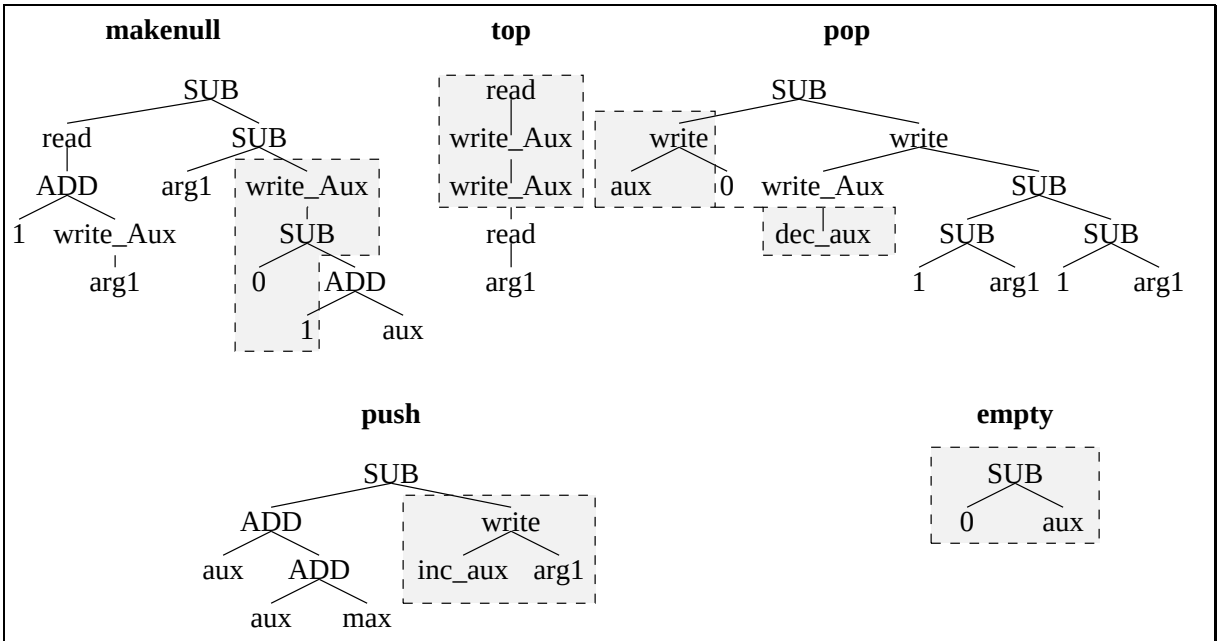

Figure 13: Evolved Stack 4

and 51). For each run the first essential primitive (or pair or primitives) that becomes extinct is selected and its covariance of frequency with fitness in the population is plotted. Figure 14 shows the covariance is predominantly negative and thus Price's theorem predicts the primitives' frequencies will fall. Figure 16 confirms this. In most cases they become extinct by generation nine.

Figure 15 shows the evolution of frequency, fitness covariance for the same primitives in a successful run (1) (Figure 17 shows the evolution of their frequency). While two of the primitives (Push/arg1 and Push/dec_aux) have large positive covariances for part of the evolution the other four are much as the runs shown in Figure 14 where they were the first essential primitive to become extinct. That is, in terms of correlation between population fitness ranking and essential primitives, successful and unsuccessful runs are similar. It appears there is a race between finding high fitness partial solutions on which a complete solution can evolve and the removal of essential primitives from the population caused by fitness based selection. I.e. if finding a critical building block had been delayed, it might not have been found at all as one or more essential primitives might have become extinct in the meantime.

In successful stack run (1) by generation five, a solution in which top, pop and push effectively use aux, write_Aux, inc_aux and dec_aux to maintain aux as a stack pointer has been discovered (c.f. Figure 17). This is followed by the fitness of Pop/inc_aux increasing and whereas its frequency had been dropping it starts to increase preventing Pop/inc_aux from becoming extinct, which would have prevented a solution like the one found from evolving. This maintenance of aux as a stack pointer requires code in three trees to co-operate. An upper bound on the chance of this building block being disrupted in the offspring of the first program to contain it can be calculated by assuming any crossover in any of the three trees containing part of the building block will disrupt it. This yields an upper bound of $3p_c/5 = 54\%$. In other words on average at least $p_r + 2p_c/5 = 46\%$ of the offspring produced by programs containing this building block will also contain the building block and so it should spread rapidly through the population. With many individuals in the population containing functioning top, pop and push trees, evolution of working makenull and empty trees rapidly followed and a complete solution was found.

4.1 Discussion

The loss of some critical primitives in so many runs can be explained in many cases by the existence of high scoring partial solutions which achieve a relatively high score by saving

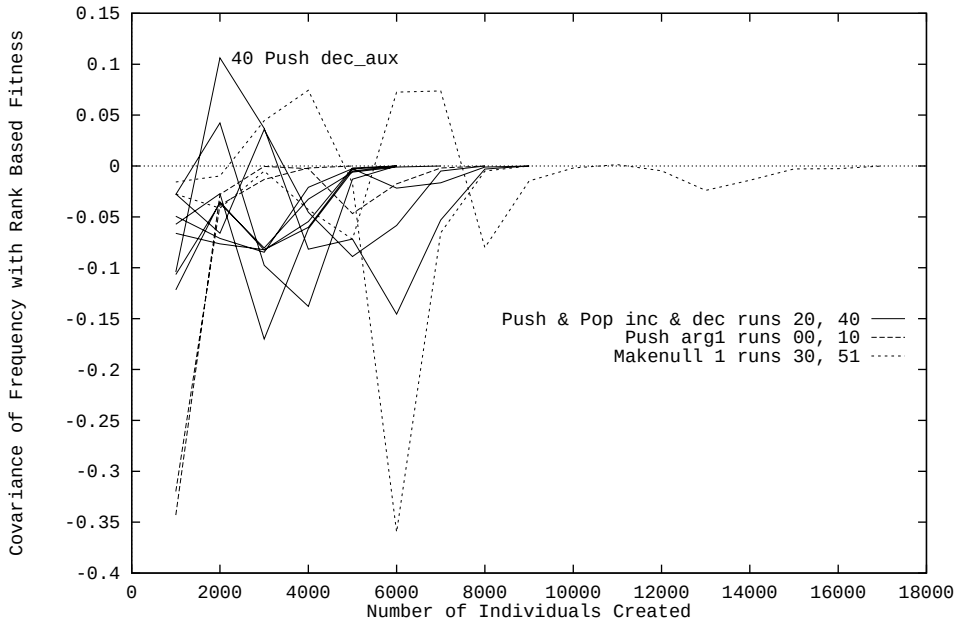

Figure 14: Evolution of the covariance of primitive frequency and $\left(\frac{R_i}{N}\right)^4 - \left(\frac{R_{i-1}}{N}\right)^4$ for the first critical primitive (or critical pair) to become extinct. Six typical stack runs.

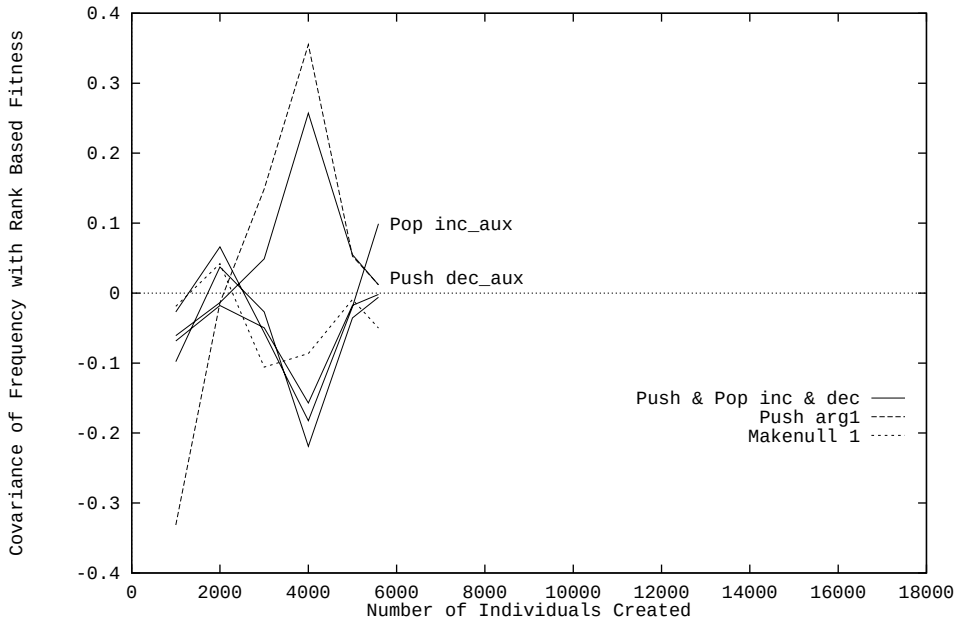

Figure 15: Evolution of the covariance of primitive frequency and $\left(\frac{R_i}{N}\right)^4 - \left(\frac{R_{i-1}}{N}\right)^4$ for critical primitives. Successful stack 1 run.

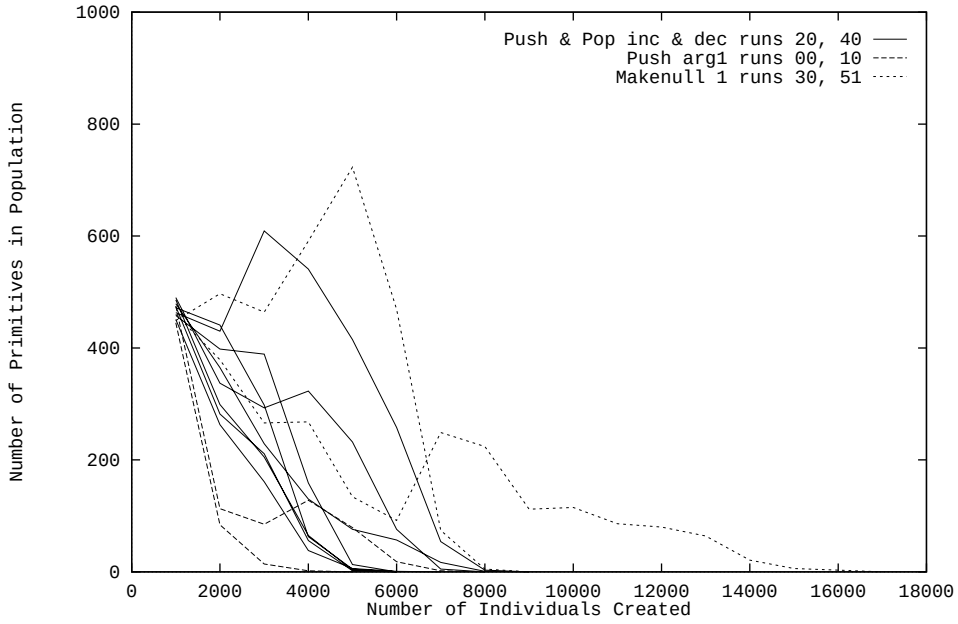

Figure 16: Evolution of number of primitives in the population for first critical primitive (or critical pair) to become extinct. Six typical stack runs.

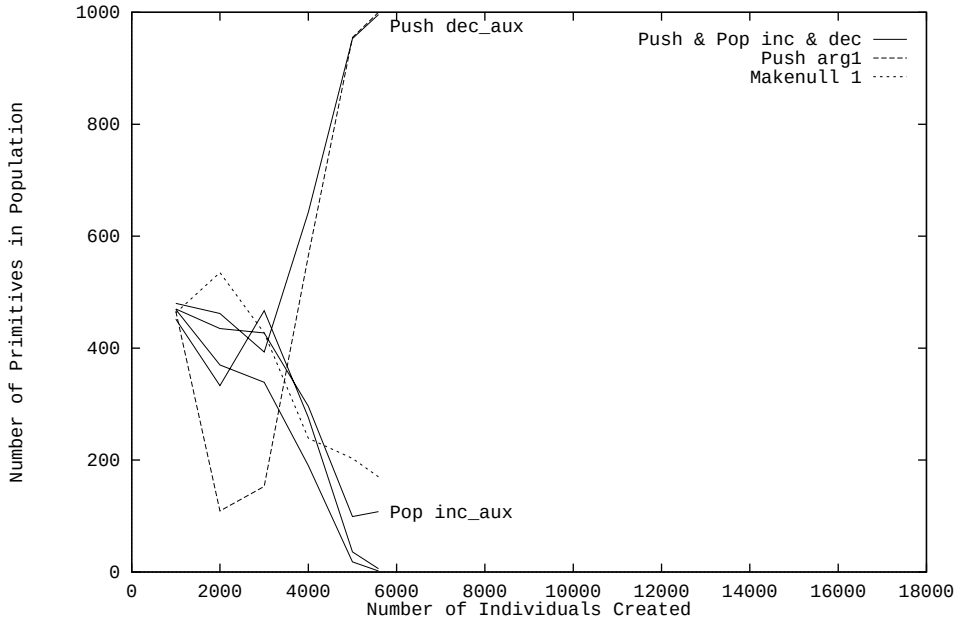

Figure 17: Evolution of number of primitives in the population for critical primitives. Successful stack 1 run.

only one item in aux. In such programs write_Aux, inc_aux and dec_aux may destroy the contents of aux and are likely to be detrimental (i.e. reduced fitness). As the number of such partial solutions increases write_Aux, inc_aux and dec_aux become more of a liability in the *current* population and are progressively removed from it. Thus trapping the population at the partial solution. This highlights the importance of the fitness function throughout the whole of the GP run. I.e. it must guide the evolution of the population toward the solution in the initial population, as well as later, when recognisable partial solutions have evolved.

[Langdon, 1996b] described a similar loss of primitives in the list problem and discussed potential solutions such as mutation, demes and fitness niches to allow multiple diverse partial solutions within the population and potentially slow down the impact of fitness selection on the population. Other approaches include: improving the fitness function (so it is no longer deceptive) e.g. by better design or using a dynamic fitness function which changes as the population evolves. A dynamic fitness function would aim to continually stretch the population, keeping a carrot dangling in front of it. (This is also known as the "Red Queen" [Carroll, 1871] approach where the population must continually improve itself). A dynamic fitness function could be pre-defined but dynamic GP fitness functions are often produced by co-evolution [Hillis, 1992; Angeline and Pollack, 1993; Angeline, 1993; Angeline and Pollack, 1994; Koza, 1991; Jannink, 1994; Reynolds, 1994; Ryan, 1995]. Where it is felt certain characters will be required in the problem's solution the initial population and crossover can be controlled in order to ensure individuals within the population have these properties ([Langdon, 1995] and [Langdon, 1996b] have described ways in which this can be implemented).

An alternative approach is to avoid specialist high level primitives (particularly where they interlock, so one requires another) and use only a small number of general purpose primitives. Any partial solutions are likely to require all of them and so none will become extinct. This is contrary to established GP wisdom [Kinnear, Jr., 1994, page 12], however recently (at the fall 1995 AAAI GP symposium) Koza advocated the use of small function sets containing only five functions ($+, -, \times, \div$ and a conditional branch).

5 Lost of Variety

We define variety as the number of unique individuals within the population. For example if a population contains three individuals A, B and C but A and B are identical (but different from C) then the variety of the population is 2 (A and B counting as one unique

individual). ([Koza, 1992, page 93] defines variety as a ratio of the number of unique individuals to population size). These definitions have the advantage of simplicity but ignore several important issues:

- Individuals which are not identical may still be similar.
- Individuals which are not identical may be total different, but *variety* makes no distinction between this and the first case.
- The differences between individuals may occur in "introns". That is in parts of the program tree which have no effect upon the program's behaviour, either because that part of the tree is never executed or because its effects are always overridden by other code in the program. For example, the value of a particular subtree may always be multiplied by zero which yields a result that is always zero no matter what value the subtree had calculated. Two such different programs have identical behaviour and fitness (but their offspring may not be the same, even on average).
- Behaviour of different program trees may be identical, either in general or in the specific test cases used to assign fitness. That is genetically diverse individuals may behave similarly, or even identically.

  As [Rocsa, 1996] points out, in the absence of side effects, diverse programs with identical behaviour can be readily constructed if the function set contains functions that are associative or commutative by simple reordering of function arguments.
- Even if programs behave differently, in general or when evaluating the given test cases, the fitness function may assign them the same fitness value. E.g. the fitness function may be based upon the number of correct answers a program returns so two programs which pass different tests but the same number of tests will have the same fitness.

Faced with the above complexity we argue that variety has the advantage of simplicity and forms a useful upper bound to the diversity of the population. That is if the variety is low then any other measures of genetic, phenotypic or fitness diversity must also be low. The opposite does not hold when it is high. (Other definitions include fitness based population *entropy* [Rosca and Ballard, 1996, Section 9.5] and using the ratio of sum of the sizes of every program in the population to the number of distinct subtrees within the population [Keijzer, 1996]).

In this section we consider the variety of GP populations using the 60 runs on the stack problem as examples. Firstly (Section 5.1) we show how the number of unique individuals evolves. Simple but general models of the evolution of variety were devised. While these gave some explanation but they failed to predict some important features. Instead detailed measurements of the stack population are presented in Section 5.2. These are used to give better, but more problem specific, explanations of the populations' behaviour. The low variety of stack populations is shown to be primarily due to the high number of "clones" (i.e. offspring which are identical to their parents) produced by crossover, which is itself a reflection of the low variety. Thus low variety reinforces itself. In one run (23) variety collapses to near zero but in most cases it eventually hovers near 60% of the population size. This is low compared to reports of 80% to 95% in [Koza, 1992, pages 159, 609 and 614] and [Keijzer, 1996].

5.1 Lost of Variety in Stack Populations

Measurements show variety starts in the initial population at its maximum value with every member of the population being different. This is despite the fact there is no uniqueness check to guarantee this. Once evolution of the population starts variety falls rapidly, but in most cases rises later to oscillate chaotically near a mean value of about 60% (see Figures 18 to 21). However in one run (23) variety does not increase and the population eventually converges to a single genotype and four of its offspring (i.e. of the 1000 individuals in the population there are only five different chromosomes, with about 970 copies of the fittest of these five).

The number of duplicate individuals created by reproduction rises rapidly initially but then hovers in the region of 8.5% of the population size (see Figure 22). This means initially most duplicate individuals are created by reproduction but this fraction falls rapidly as more duplicates are produced by crossover so after the seventh generation only about a quarter of duplicate individuals in the population were created by reproduction and the remaining three quarters are created by crossover (see Figure 18). In stack populations, crossover produces more duplicates shortly after each new improved solution is found (see Figure 23).

5.2 Measurements of GP Crossover's Effect on Variety

This section examines in detail the role of crossover in reducing variety in the stack populations. We discover there are two main causes; crossover which just involves swapping terminals and crossover which entails replacing whole trees. Where variety is low both

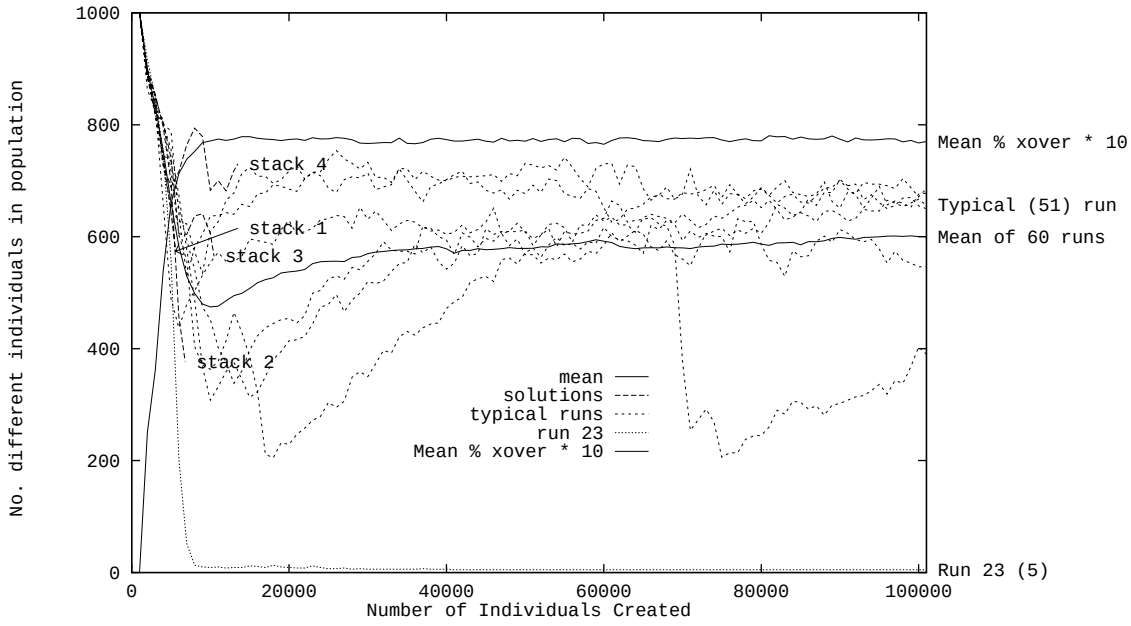

Figure 18: Number of different individuals in stack populations and proportion of subsequent duplicates produced by crossover in stack selected runs.

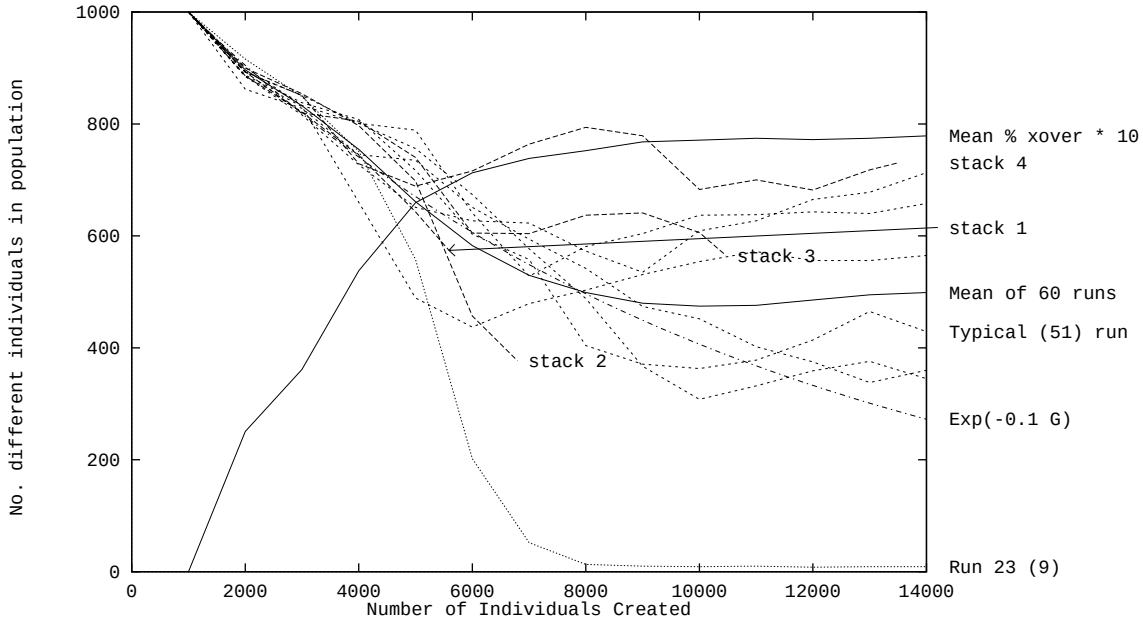

Figure 19: Detail of above

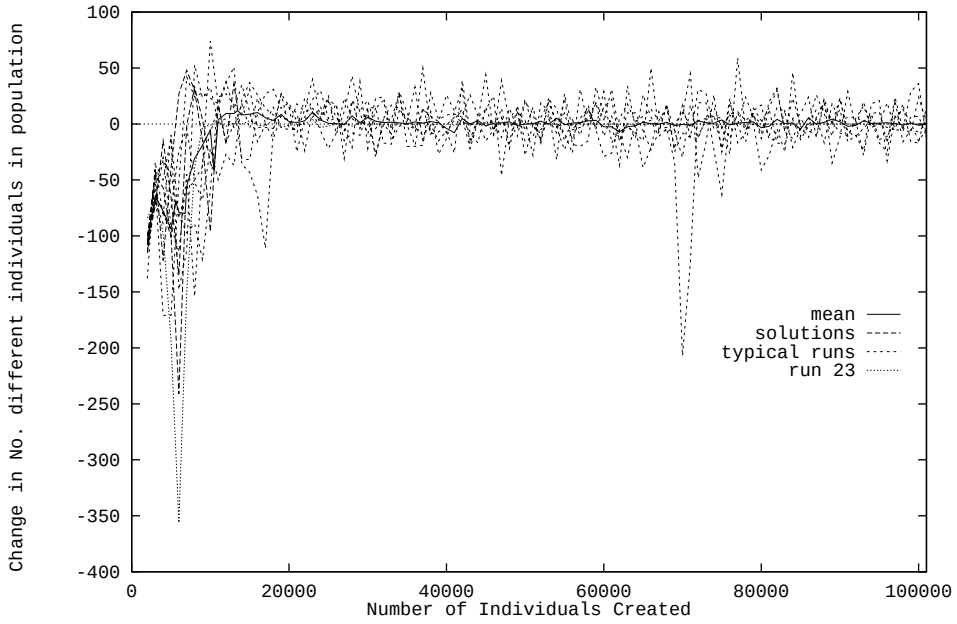

Figure 20: Change in number of different individuals in stack populations.

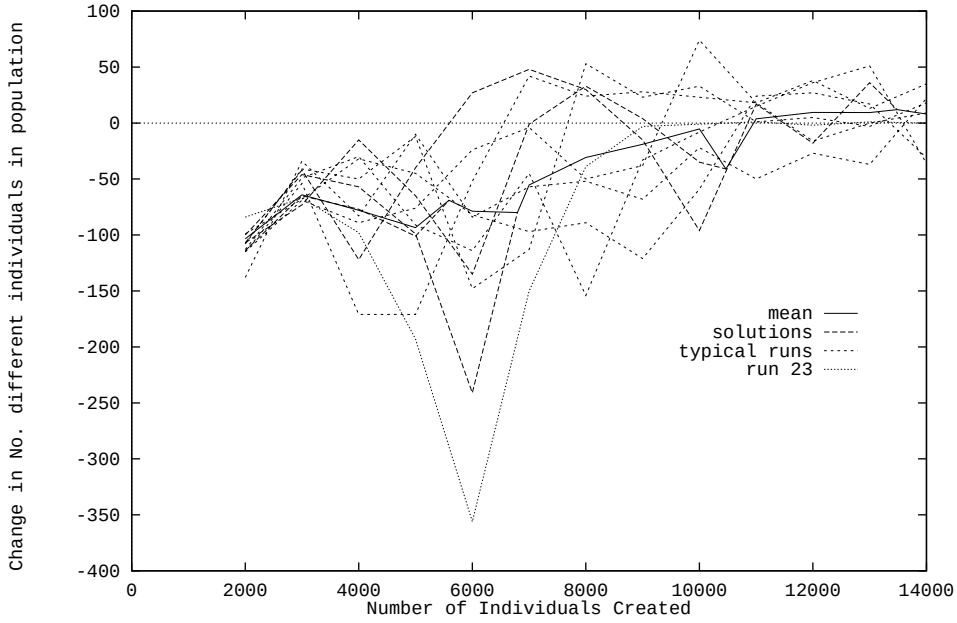

Figure 21: Detail of above

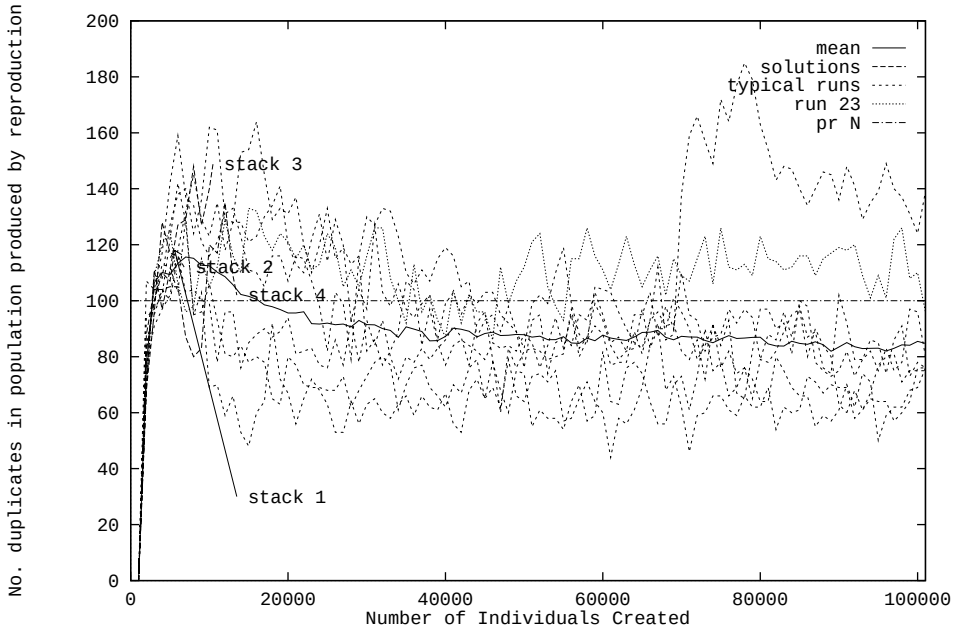

Figure 22: Number of duplicate individuals in stack populations that were produced by reproduction in selected runs.

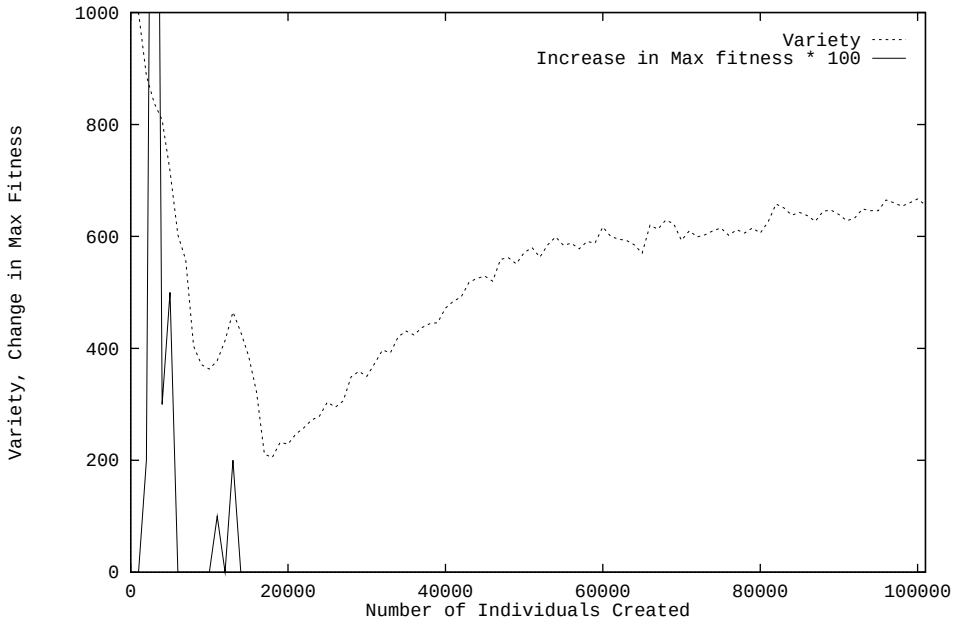

Figure 23: Number of different individuals in stack populations and change in maximum fitness in a typical stack run (51).

lead to further production of clones of the first parent. Quantitative models of these two effects are in close agreement with measurements.

Figure 24 shows the proportion of cases where the offspring produced by crossover are identical to one or other of its parents. (In a typical stack run all offspring which are duplicates of other members of the population are identical to one or other parent). In a typical run of the stack problem about one third of crossovers produce offspring which are identical to their first parent. Table 5 gives the total number of offspring produced by crossover during the run that are clones for various size of crossover fragments.

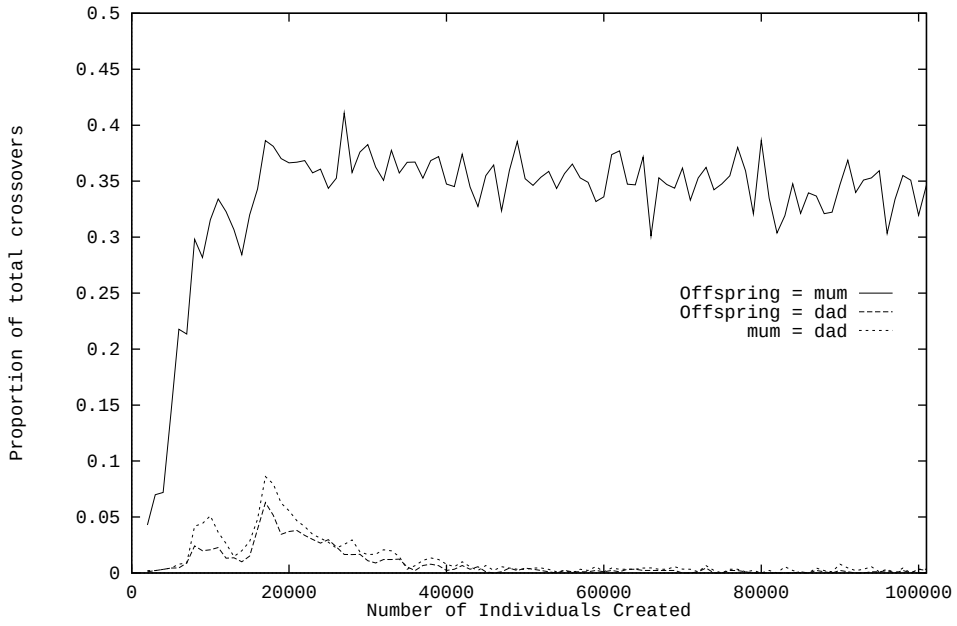

Figure 24: Proportion of crossovers that yield offspring identical to one or other parents, typical stack (51) run (Also shows proportion where the two parents are identical).

For crossover to produce a clone of the first parent the fragment of code that is lost much be identical to that copied from the second parent. As crossover fragments which are taller are generally larger we would expect the chance of this happening to reduce rapidly with fragment height. Whilst Table 5 shows this is generally true, it is definitely not the case for fragment height 2.

In stack run 51 18,644 individuals are produced by crossover which are identical to their first parent and where the inserted subtree had a height of 2, i.e. fragments consisting of one function and its arguments which are all terminals. Of these 18,644, there were 16,536 individuals where the tree in which crossover occurred contained only one function and so crossover entailed replacing the whole tree with another from the other parent, which turned out to be identical to the first. In this regard the stack problem is atypical, normally

Table 5: Number of crossovers of each height of subtree inserted in a typical stack run (51) and number of these crossovers which produced a non-unique offspring.

Fragment height			Identical to				
	Total	%	mum	dad	both	either	%
1	28,783	32	9,513	38	128	9,679	32
2	28,277	31	18,644	60	305	19,009	62
3	15,360	17	1,060	79	28	1,167	4
4	3,884	4	303	42	6	351	1
5+	13,784	15	202	33	10	245	.8
Totals	90,088	100	29,722	252	477	30,451	100
	Percent		33	.3	.5	34	

trees or ADFs will have multiple functions and we would expect few clones to be produced by crossover of trees with a of height 2. In this run of the stack problem most of the clones are produced by crossover in trees which are short (height of 2) and identical in both parents. Thus we see clones (which reduce variety) being caused by lack of diversity in the population.

5.2.1 Production of Clones by Crossover in Full Binary Trees

In a full binary tree of height h there are $2^h - 1$ nodes of which 2^{h-1} are terminals and $2^{h-1} - 1$ are internal nodes. Consider crossover between two identical trees where each node is distinct. For crossover to produce an individual which is identical to its parents the crossover points selected in both parents must be the same. The chance of this happening would simply be $(2^h - 1)^{-1}$ if nodes were chosen at random. However the parameter pUnRestrictWt (cf. Section 2.4) means only 70% of crossover points are chosen totally at random. In the remaining 30% of cases the chosen point must be an internal tree node. From Equation 6 we see for large trees pUnRestrictWt's effect is to increase the chance of producing a clone by 9%. The probabilities for smaller trees are tabulated in Table 6.

$$
\begin{aligned}
\text{p(clone)} &= \text{p(Tree1 internal)} \times \text{p(Tree2 same internal)} + \\
&\quad \text{p(Tree1 external)} \times \text{p(Tree2 same external)} \\
&= \left((1 - p_{any}) + p_{any} \frac{2^h - 1 - 2^{h-1}}{2^h - 1} \right) \times \text{p(Tree2 same internal)} + \\
&\quad p_{any} \frac{2^{h-1}}{2^h - 1} \times \text{p(Tree2 same external)}
\end{aligned}
$$

$$
\begin{aligned}
&= \left((1-p_{any}) + p_{any}\frac{2^{h-1}-1}{2^h-1}\right) \times \text{p(Tree2 same internal)} + \\
&\quad p_{any}\frac{2^{h-1}}{2^h-1} \times \text{p(Tree2 same external)} \\
&= \left((1-p_{any}) + p_{any}\frac{2^{h-1}-1}{2^h-1}\right) \times \left((1-p_{any}) + p_{any}\frac{2^{h-1}-1}{2^h-1}\right)/(2^{h-1}-1) + \\
&\quad p_{any}\frac{2^{h-1}}{2^h-1} \times p_{any}\frac{2^{h-1}}{2^h-1}/2^{h-1} \\
&= \frac{\left((1-p_{any}) + p_{any}\frac{2^{h-1}-1}{2^h-1}\right)^2}{(2^{h-1}-1)} + \frac{\left(p_{any}\frac{2^{h-1}}{2^h-1}\right)^2}{2^{h-1}} \qquad (5)
\end{aligned}
$$

As h increases

$$
\begin{aligned}
\text{p(clone)} &\approx \frac{(1-p_{any}/2)^2}{(2^{h-1}-1)} + \frac{p_{any}^2/4}{2^{h-1}} \\
&\approx \frac{(1-p_{any}/2)^2}{2^{h-1}} + \frac{p_{any}^2/4}{2^{h-1}} \\
&= \frac{(1-p_{any}/2)^2 + p_{any}^2/4}{2^{h-1}} \\
&= \frac{1-p_{any}+p_{any}^2/4+p_{any}^2/4}{2^{h-1}} \\
&= \frac{1-p_{any}+p_{any}^2/2}{2^{h-1}}
\end{aligned}
$$

Since $p_{any} = 0.7$ for large h

$$
\begin{aligned}
&= 1.09\ 2^{-h} \\
\text{p(clone)} &\approx 1.09\ (2^h-1)^{-1} \qquad (6)
\end{aligned}
$$

Table 6: Chance of offspring being identical to parents when crossing two identical full binary trees

Tree height	Chance of clone		$p_{any} = 1$
1	1	1.000	1.000
2	$\left((1-p_{any}) + p_{any}\frac{1}{3}\right)^2 + \frac{\left(p_{any}\frac{2}{3}\right)^2}{2}$	0.393	.333
3	$\frac{\left((1-p_{any})+p_{any}\frac{3}{7}\right)^2}{3} + \frac{\left(p_{any}\frac{4}{7}\right)^2}{4}$	.160	.143
4	$\frac{\left((1-p_{any})+p_{any}\frac{7}{15}\right)^2}{7} + \frac{\left(p_{any}\frac{8}{15}\right)^2}{8}$	.074	.067
5	$\frac{\left((1-p_{any})+p_{any}\frac{15}{31}\right)^2}{15} + \frac{\left(p_{any}\frac{16}{31}\right)^2}{16}$	.035	.032

The chance of producing a clone from two identical trees in a real GP population may not be exactly as given by Equation 5. This is because: the trees may not be full binary trees, i.e. they will be smaller if there are terminals closer to the root than the maximum

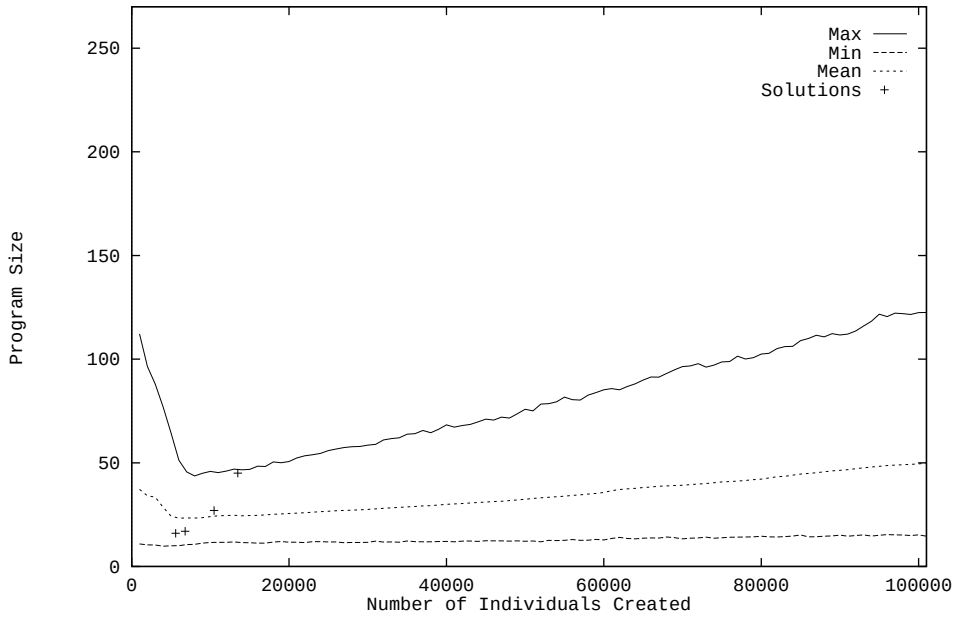

Figure 25: Evolution of program size, means of 60 stack runs. The lengths of the four solutions to the stack problem are also shown.

height of the tree, or if functions have one argument rather than two. Conversely trees can be also be larger if functions have three or more arguments. Also the chance of producing a clone is increased if actual trees contain repeated subtrees.

In the case of two identical trees of height two and crossover fragments of height two the chance of producing a clone is equal to the chance of selecting the root in the first tree which depends upon the number of arguments the tree has. For n arguments, the chance of producing a clone is $(1 - p_{any}) + p_{any}/(n+1) = 1 - n\ p_{any}/(n+1)$ which is 65%, 53%, 48% and 44% for $n = 1, 2, 3$ and 4. In other words given a population where the best solution found has a height of two and the inserted crossover fragment is also of height two and there is a high chance of selecting (copies of) the individual to be both parents we expect the offspring to be a clone between 53% and 65% of the time, which is consistent with the figure of 16,536 such clones produced in a typical stack run (cf. page 29).

Thus one of the major causes of the fall in variety in the stack populations can be traced to finding partial solutions early in the evolution of the population with relatively high fitness where trees within it are short. As the whole individual is composed of five trees, its total size need not be very small. Figure 25 provides additional evidence for this as it shows on average stack individuals shrink early in the run to 23.3 at generation six. I.e. on average each tree contains 4.7 primitives and as there must be many trees shorter than this, many trees must have a height of two or less.

Table 7: Chance of selecting a terminal as a crossover fragment in a full binary tree

Height		Both parents
1	100 %	100 %
2	47 %	22 %
3	40 %	16 %
4	37 %	14 %
∞	35 %	12.25 %

5.2.2 Production of Clones by Crossover Swapping Terminals

The other major reason for crossover to produce clones in the stack runs is crossover fragments which contain a single terminal (cf. Table 5). The proportion of clones these crossovers produce can be readily related to lack of diversity. The proportion of crossover fragments which are a single terminal depends upon the depth and bushiness of the trees within the population, which in turn depends upon the number of arguments required by each function in the function set and how the distribution of functions evolves. The proportion of crossover fragments which are a single terminal is clearly problem dependent and changes with run and generation within the run, however as a first approximation in the stack problem it can be treated as a constant for each type of tree (cf. Figure 27).

For a full binary tree of height h the chance of selecting a terminal as a crossover fragment is $p_{any}2^{h-1}/(2^h-1)$ and the chance of crossover swapping two terminals is $\left(p_{any}2^{h-1}/(2^h-1)\right)^2$. Table 7 gives the numerical values for trees of different heights. Note the chance of selecting a terminal converges rapidly to 35% for large trees.

If parents were chosen at random the chance of selecting the same terminal in two trees would be simply the sum of the squares of their proportions in the population. Thus if the terminals are equally likely (as would be expected in the initial population) the chance of selecting two the same is just the reciprocal of the number of terminals and this rises as variety falls eventually reaching unity if all but one terminal are removed from the population. Figure 26 shows how this measure evolves for each tree in a sample of stack runs. Note in run (23) all five trees quickly converge on a single terminal. In many of the other runs the population concentrates on one or two terminals, so the chance of an offspring produced by changing a single terminal being a clone of one of its parents is much increased.

Typically 15.8% of crossovers replace one terminal with another terminal (cf. Table 8). This is near the proportion expected for full binary trees with a height of three or more.

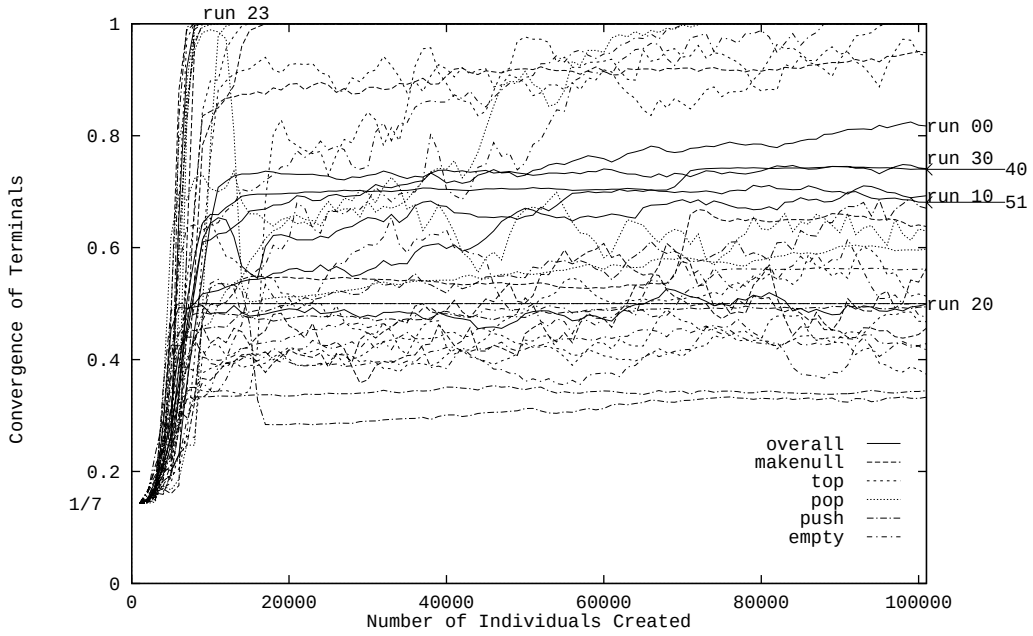

Figure 26: Evolution of (Terminal Concentration)^2 in each operation tree, for six typical stack runs and run (23).

Table 8 shows reasonable agreement between the predicted number of clones produced by crossover inserting a single terminal and the actual number averaged over a typical run of the stack problem.

The second major source of crossover produced reduction in variety (cf. Table 5) is thus explained by the fall in terminal diversity, itself a product of the fall in variety. So again we see low variety being reinforced by crossover, i.e. the reversal of its expected role of creating new individuals.

Table 8: Number of clones produced by changing a terminal in run (51) of the stack problem

Tree	No. Crossovers	Terminal Only	$\sum(term\ conc)^2$	Predicted	Actual
makenull	18,020	3,326	.924424	3,074.6	3,075
top	17,914	3,022	.798273	2,412.4	2,684
pop	18,013	4,895	.565901	2,770.1	2,819
push	18,021	2,306	.318201	733.8	740
empty	18,120	668	.511968	342.0	339
Totals	90,088	14,217		9,334.9	9,657

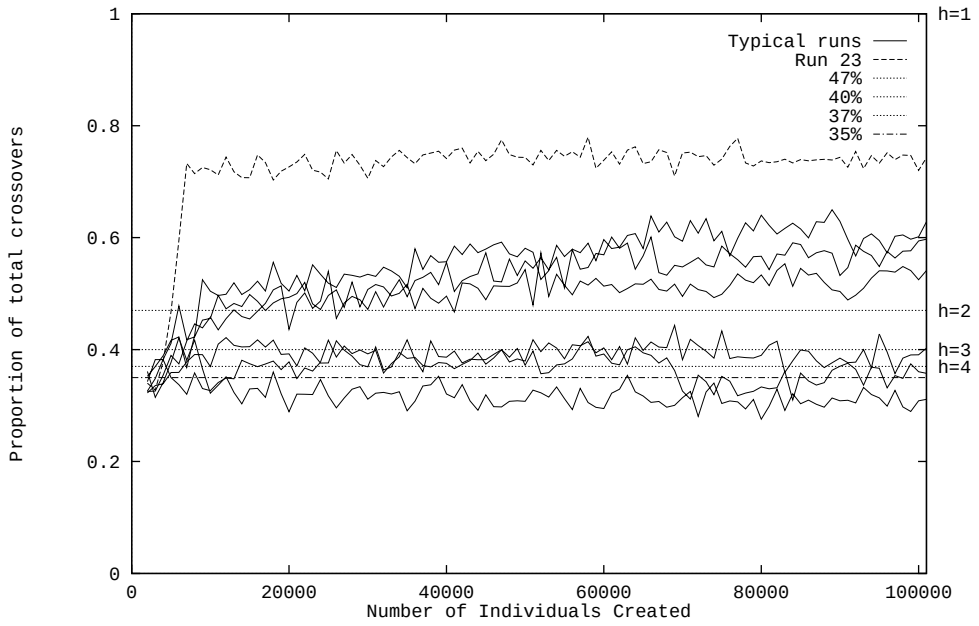

Figure 27: Proportion of crossovers where a terminal is inserted for six typical stack runs and run (23) (averaged across all five trees within each individual).

6 Measurements of GP Crossover's Effects

In this section we analyse how successful crossover is at finding new solutions with higher fitness and conclude in the case of the stack problem, crossover quickly tires and the rate of finding improvements slows rapidly so after generation eight very few are found and typically no improvements are found after generation 16. Note this includes all crossovers not just those that produce offspring that are better than anyone else in the population.

Table 9 gives the number of crossovers which produced an offspring fitter than both its parents, for run (23), six typical runs and the four successful runs. The successful runs produce about 50% more successful runs than typical runs. The parents of successful crossovers and their offspring are plotted in Figures 28 and 29 for a typical and a successful run respectively. However the number of successful crossovers is more than the number of different fitness values, that is there are fitness values which have been "discovered" by multiple successful crossovers. Clusters of particularly popular fitness values that were "rediscovered" many times can be seen in Figures 28 and 29. E.g. fitness value 128 is discovered 22 times in run (51) (22 is 13% of all the successful crossovers).

The proportion of successful crossovers in six selected stack runs is shown in Figure 30. Note the number of crossovers that produce improved offspring is small and quickly falls so after generation 16 there are almost no crossovers that improve on both parents (or

Table 9: No. of Successful Crossovers, in Typical and Successful Stack Runs

Run	Crossover point in Tree					Total	Best Fitness
	Makenull	Top	Pop	Push	Empty		
23	33	32	54	18	46	183	130
00	22	34	57	43	20	176	108
10	27	34	85	29	24	199	108
20	36	41	31	13	59	180	128
30	22	25	44	21	44	156	131
51	38	31	48	16	30	163	139
40	63	75	50	26	90	304	150
27	72	67	47	18	53	257	160
32	69	56	42	25	77	269	160
09	42	63	54	22	75	256	160
53	33	55	44	38	25	195	160

indeed improve on either).

Figure 31 shows the fitness of individuals selected to be crossover parents. This shows the convergence of the population with almost all parents having the maximum fitness value. (The asymmetry of the fitness function makes the mean fitness of the population lower than the fitness of the median individual).

7 Discussion

Natural evolution of species requires variation within a population of individuals as well as selection for survival and reproduction. In the previous sections we have seen how, even on the most basic measure, variety in the stack populations falls to low levels primarily due to crossover producing copies of the first parent at high rates. Initially this is caused by the discovery of relatively high fitness partial solutions containing very small trees which dominate the population, reducing variety which causes feedback via crossover produced clones so keeping variety low, in one case causing it to collapse entirely. As we argued in Section (4) in most stack runs lack of variety with corresponding extinctions of primitives prevents solutions like those found from evolving.

In any genetic search a balance between searching the whole search space for an optimum and concentrating the search in particular regions is required. Some convergence of the population is expected as the GA concentrates on particularly fruitful areas. In most stack runs partial solutions are found which act similarly to a stack of one item and so

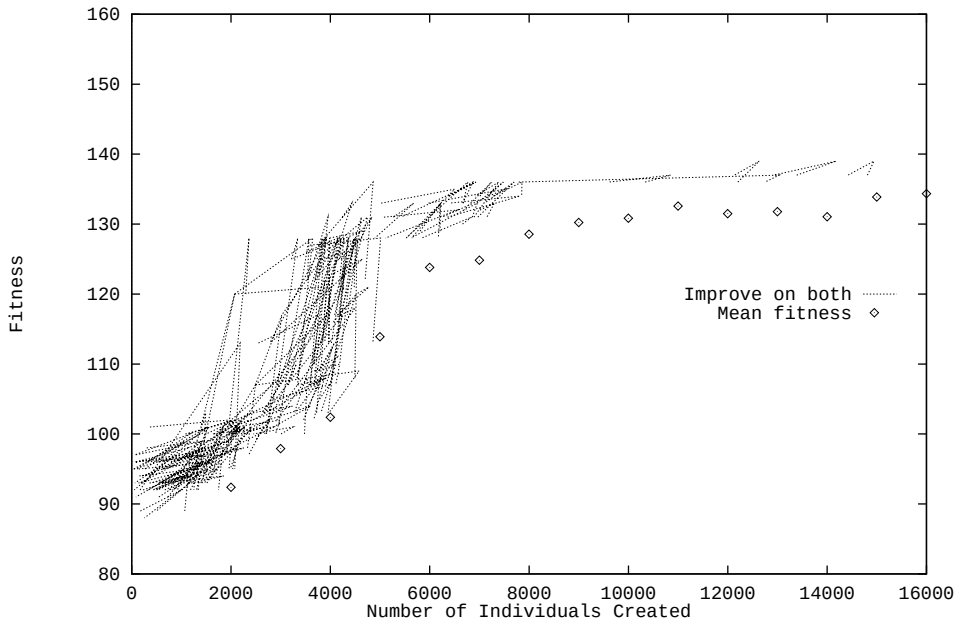

Figure 28: All crossovers that produced offspring fitter than both parents, typical stack run (51).

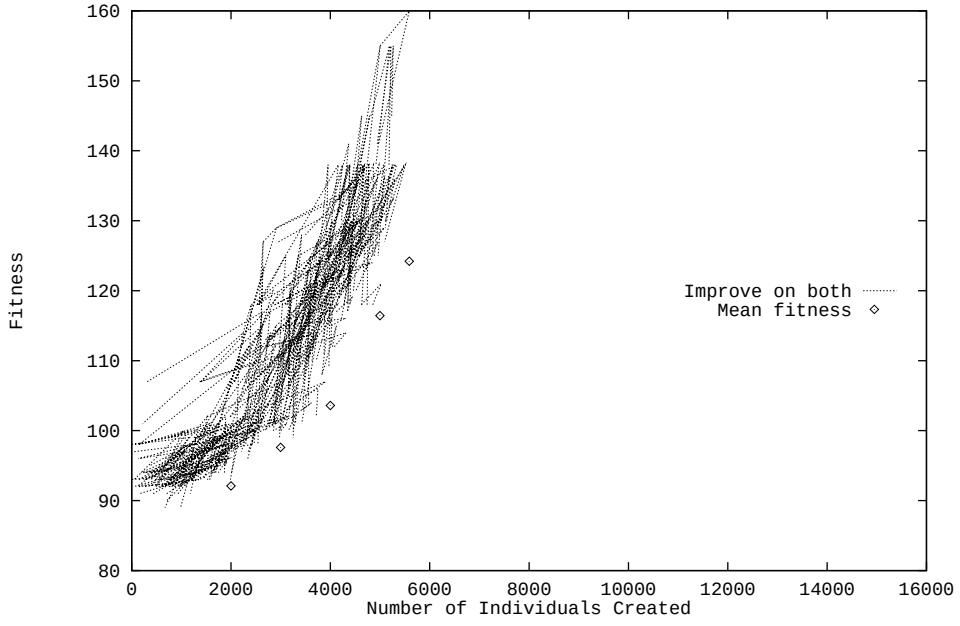

Figure 29: All crossovers that produced offspring fitter than both parents, successful run (09).

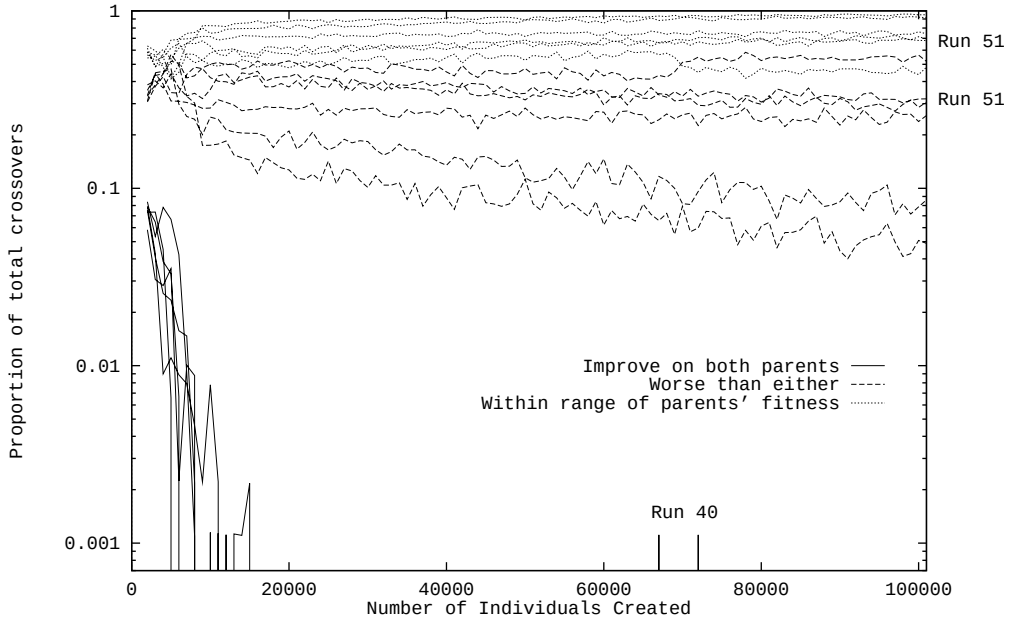

Figure 30: Proportion of crossovers that produced offspring fitter than both parents, worse than both or neither. Six typical stack runs.

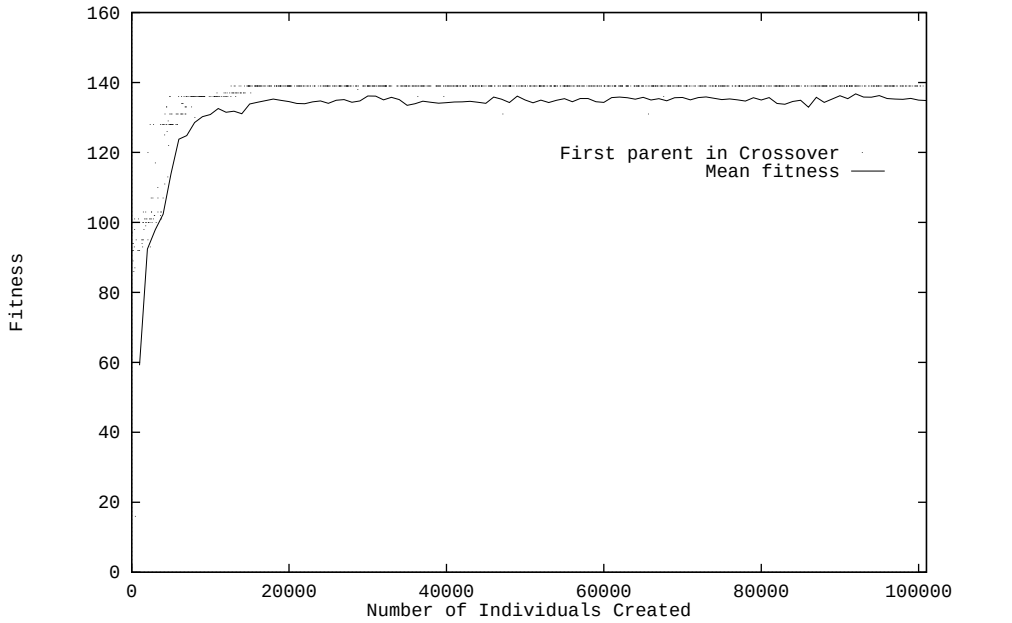

Figure 31: Fitness of parents selected for crossover in typical stack (51) run. (Extrema and 1% of data for first parent only are plotted).

receive a high relative fitness and the population begins to converge to them. This would be fine apart from two problems: firstly the solutions contain short trees which causes rapid production of clones but more seriously there is no straightforward path from their implementation of a stack of one item to a general stack of many items. These two problems are to some extent specific to the stack problem, the five tree architecture and the terminal/function set used. A smaller terminal/function set without special primitives to manipulate "aux", having only general primitives and indexed memory, might avoid the trapping by "deceptive solutions" but partial solutions of any sort might then not evolve in a reasonable time. In the stack problem each terminal and function can appear in each of the five trees but crossover acts only between like trees so each tree is genetically isolated from each other. (This is known as *branch typing* and is commonly used with ADFs [Koza, 1994, page 86]. An alternative *point typing* allows crossover to move genetic material between trees). Branch typing means there are effectively $5 \times 12 = 60$ primitives in the stack problem. [Andre, 1996] also reports GP runs with similar numbers of primitives where one or more functions either evolved out of the population (i.e. became extinct) or became rare and suggests it was a factor in the decision to use mutation (albeit at a low rate). However he cautions that further experiments are required for confirmation.

The impact of deceptive partial solutions within the population might be reduced by partitioning the population into "demes" [Stender, 1993; Collins, 1992; Tackett, 1994; Koza and Andre, 1995; Juille and Pollack, 1995], using fitness niches to ensure diverse solutions are retained [Goldberg, 1989]or perhaps using co-evolution to reward solutions to parts of the test case which most of the population is unable to solve.

Mutation could also be used to increase population diversity but a high mutation rate might be required to escape from a deceptive local optimum. This would increase the degree of randomness in the search but might introduce a beneficial element of "hill climbing", see [O'Reilly and Oppacher, 1996] and [Iba *et al.*, 1994b]).

While other GPs may not suffer from lack of variety, convergence of some sort is required if the GP is not to be random search. For example [Keijzer, 1996] shows convergence in terms of subtrees with GP populations reusing subtrees in many individuals. (GP may take advantage of this by reducing the space taken to store the population in memory [Keijzer, 1996] and on disk (by using file compression) Where side-effects are controlled, retaining information on the evaluation of common subtrees within the population can also considerably reduce program execution time, c.f. [Handley, 1994]).

Existing GP systems could be modified to:

1. Increase variety by disabling the production of clones by the reproduction operator, e.g. by setting p_r to zero.

2. Detect when an offspring is identical to one of its parents. This information can be readily gathered and can be used either to:

   (a) reduce GP run time or

   (b) Increase variety.

   In many problems (a) can be readily achieved by avoiding the fitness evaluation of the offspring and instead just copying the fitness value of its (identical) parent. Variety can be kept high (b) by preventing the duplicate offspring from entering the population. Typically this would prevent all duplicates produced by crossover. (It would also be feasible to *guarantee* every member of the population is unique by forbidding duplicates from entering the population. Using hashing techniques this can be done efficiently).

Given current GP populations sizes it would appear to be sensible to ensure variety remains high so the compromise between converging on good search location and exploring untried areas retains a high degree of exploration. Thus both changes 1. and 2.b) should be tried.

The use of $p_r = 0.1$ stems from the decision to use parameters as similar to [Koza, 1992] as possible. It is also the supplied default value with GP-QUICK [Singleton, 1994]. However the use of reproduction is not universal, for example the CGPS [Nordin, 1994; Nordin and Banzhaf, 1995; Francone *et al.*, 1996] does not implement it. As far as is known, GP systems do not currently detect that crossover has produced a child which is identical to one of its parents for the purposes of either reducing run time (2.a) or increasing variety (2.b). [Koza, 1992, page 93] ensures every member of the initial population is unique but allows duplicates in subsequent generations. While hashing allows detection of duplicates in the whole population to be done quickly, in these experiments most duplicates were directly related to each other and so could be readily detected without comparison with the whole population.

It appears to be common practice for GP to "run out of steam" so after 20–30 generations no further improvement in the best fitness value in the population occurs or improvement occurs at a very low rate. Accordingly few GP runs are continued beyond generation 50. ([Iba *et al.*, 1994a]'s STROGANOFF system provides a counter example with runs of 400 generations). It is suggested that failure of crossover to improve on the

best individual in the population may, as we saw in Section 6, be accompanied by a general failure of crossover to make any improvement. This "death of crossover" means further evolution of the population is due to unequal production of individuals with the same (or worse) fitness as their parents, in fitness terms (and possible also phenotypically) at best they are copies of their parents. Typically this serves only to increase the convergence of the population.

An number of attempts to "scale up" GP have been made based upon imposing functional abstraction on individuals in the population [Koza, 1994; Angeline, 1993; Rosca, 1995]. These have had a degree of success. Another approach is to accept that complex problems will require many generations to solve and look to the various mechanisms described above and new techniques to allow long periods of GP evolution with controlled convergence of the GP population and means to retain and reuse (partial) solutions.

8 Summary

Earlier we discussed Price's selection and covariance theorem and showed it can be applied to genetic algorithms and applied it to genetic programming, where we used it to explain the evolution of the frequency of various critical primitives in stack populations including their rapid extinction in many cases. These extinctions are seen as the main reason why many runs of the stack problem (described in [Langdon, 1995]) failed. In Section 5 it was shown that the loss of these primitives was accompanied by a general loss in variety. While general models have been developed to try and explain this they were only partially successful and quantitatively successful models based upon full binary trees of particular heights were developed. Section 6 concludes by looking at just the successful crossovers in the stack runs and concludes they are small in number, in many cases they "rediscover" solutions that have already been found and convergence of the population is accompanied by absence of crossovers that produce offspring fitter than their parents as well as none that are fitter than the best existing individuals in the population.

To some extent these problems are fundamental. Viewing GP as a search process there is necessarily a trade-off between concentrating the search in promising parts of the search space which increases the chance of finding local optima versus a wider ranging search which may therefore be unsuccessful but may also find a more remote but better global optimum. In GA terms a local search corresponds to a more converged population. The stack experiments indicate, after the fact, that the search was too focused too early and so the global optima were missed in many runs. There are many techniques that can be

used to ensure population diversity remains high (and so the search is defocused) such as splitting the population into demes, fitness niches and mutation, some of which were used in [Langdon, 1995; Langdon, 1996b; Langdon, 1996a]. Techniques based on biased mate selection to preserve diversity are discussed in [Ryan, 1994].

Defocusing the search means the search is more random and will take longer, if indeed it succeeds. Other approaches to avoid getting trapped at local optima ("premature convergence") change the search space, for example by changing the representation by changing the primitives from which solutions are composed or changing the fitness function.

Changing the primitives can easily be done by hand. It would be interesting to discover to what extend the problems are due to provision of the auxiliary register (scalar variable, cf. Section 1.3) primitives which allow the evolution of stacks but also allow ready formation of deceptive partial solutions. If these were not used, would stacks still evolve? Alternatively perhaps cleverer genetic operations could avoid the trap by changing programs from using one type of memory to another in a consistent manner so new programs continue to work as before. While strongly typed GP can reduce the size of the search space [Montana, 1995], it may also transform it so that it is easier to search.

There are a number of techniques which automatically change the representation. The following three techniques co-evolve the representation as the population itself evolves; The Genetic Library Builder (GLiB) [Angeline, 1994], Automatically Defined Functions (ADFs) [Koza, 1994] and Adaptive Representations [Rosca, 1995]. [Koza, 1994, page 619] argues ADFs and other representations provide a different *lens* with which to view the solution space and that ADFs may help solve a problem by providing a better lens.

The fitness function may be readily changed by hand. For example provision of an additional test case may "plug a gap" which GP populations are exploiting to achieve high fitness on the test case but at the expense of not generalising to the problem as a whole. Co-evolution can provide an automatic means of dynamically changing the fitness function [Siegel, 1994]. There is increasing interest in using co-evolution [Sen, 1996; Reynolds, 1994; Ryan, 1995] and improved performance has been claimed [Hillis, 1992]. However a more dynamic framework makes analysis of population behaviour harder.

In GP runs the concentration of primitives and variety within the population should be monitored (both can be done with little overhead). Should a primitive fall to low concentration (such as close to the background level provided by mutation) or total extinction this should be taken as an indication of possible problems and so worthy of further investigation. Similarly if the number of unique individuals in the population falls below 90%

this should also be investigated. [Keijzer, 1996] provides a means to measure the concentration of groups of primitives (sub trees) but the implementation is not straightforward for most existing GP systems and the interpretation of the results is more complex.

Acknowledgments

W. B. Langdon is funded by the EPSRC and The National Grid Company plc.

I would like to thank my supervisors M. Levene and P. C. Treleaven, for their critisims and ideas, Lee Altenberg for helpful comments on this work and Andy Singleton for GP-QUICK.

References

[Altenberg, 1994] Lee Altenberg. The evolution of evolvability in genetic programming. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, pages 47–74. MIT Press, 1994.

[Altenberg, 1995] Lee Altenberg. The Schema Theorem and Price's Theorem. In L. Darrell Whitley and Michael D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 23–49, Estes Park, Colorado, USA, 31 July–2 August 1994 1995. Morgan Kaufmann.

[Andre, 1996] David Andre. Personal communication, 15 Jul 1996.

[Angeline and Pollack, 1993] Peter J. Angeline and Jordan B. Pollack. Competitive environments evolve better solutions for complex tasks. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 264–270, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[Angeline and Pollack, 1994] P. J. Angeline and J. B. Pollack. Coevolving high-level representations. In C. G. Langton, editor, *Artificial Life III*, volume XVII of *SFI Studies in the Sciences of Complexity*, pages 55–71. Addison-Wesley, 1994.

[Angeline, 1993] Peter John Angeline. *Evolutionary Algorithms and Emergent Intelligence*. PhD thesis, Ohio State University, 1993.

[Angeline, 1994] Peter John Angeline. Genetic programming and emergent intelligence. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 4, pages 75–98. MIT Press, 1994.

[Angeline, 1996] Peter J. Angeline. An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 21–29, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Carroll, 1871] Lewis Carroll. *Through the Looking-Glass, and What Alice Found There.* Macmillan, 1871.

[Collins, 1992] Robert J. Collins. *Studies in Artificial Evolution.* PhD thesis, UCLA, Artificial Life Laboratory, Department of Computer Science, University of California, Los Angeles, LA CA 90024, USA, 1992.

[Ewens, 1989] W. J. Ewens. An interpretation and proof of the fundamental theorem of natural selection. *Theoretical Population Biology*, 36(2):167–180, 1989.

[Ewens, 1992a] W. J. Ewens. Addendum to "The fundamental theorem of natural selection in Ewens' sense (case of many loci)" by Catilloux and Lessard. *Theoretical Population Biology*, 48(3):316–317, 1992.

[Ewens, 1992b] W. J. Ewens. An optimizing principle of natural selection in evolutionary population genetics. *Theoretical Population Biology*, 42(3):333–346, 1992.

[Fisher, 1958] Ronald A. Fisher. *The Genetical Theory of Natural Selection.* Dover, 1958. Revision of first edition published 1930, OUP.

[Francone *et al.*, 1996] Frank D. Francone, Peter Nordin, and Wolfgang Banzhaf. Benchmarking the generalization capabilities of A compiling genetic programming system using sparse data sets. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 72–80, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Frank, 1995] S. A. Frank. George Price's contributions to evolutionary genetics. *Journal of Theoretical Biology*, 175:373–388, 1995.

[Goldberg, 1989] David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning.* Addison-Wesley, 1989.

[Handley, 1994] S. Handley. On the use of a directed acyclic graph to represent a population of computer programs. In *Proceedings of the 1994 IEEE World Congress on*

Computational Intelligence, pages 154–159, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[Hillis, 1992] W. Daniel Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. In Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, volume X of *Sante Fe Institute Studies in the Sciences of Complexity*. Addison-Wesley, 1992.

[Holland, 1973] John H. Holland. Genetic algorithms and the optimal allocation of trials. *SIAM Journal on Computation*, 2:88–105, 1973.

[Holland, 1992] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, 1992. First Published by University of Michigan Press 1975.

[Iba *et al.*, 1994a] H. Iba, T. Sato, and H. de Garis. System identification approach to genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 401–406, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[Iba *et al.*, 1994b] Hitoshi Iba, Hugo de Garis, and Taisuke Sato. Genetic programming with local hill-climbing. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature III*, pages 334–343, Jerusalem, 9-14 October 1994. Springer-Verlag.

[Jannink, 1994] Jan Jannink. Cracking and co-evolving randomizers. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 20, pages 425–443. MIT Press, 1994.

[Juille and Pollack, 1995] Hugues Juille and Jordan B. Pollack. Parallel genetic programming and fine-grained SIMD architecture. In E. S. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 31–37, MIT, Cambridge, MA, USA, 10–12 November 1995. AAAI.

[Keijzer, 1996] Maarten Keijzer. Efficiently representing populations in genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 13, pages 259–278. MIT Press, Cambridge, MA, USA, 1996.

[Kinnear, Jr., 1994] Kenneth E. Kinnear, Jr. A perspective on the work in this book. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 1, pages 3–19. MIT Press, 1994.

[Koza and Andre, 1995] John R. Koza and David Andre. Parallel genetic programming on a network of transputers. Technical Report CS-TR-95-1542, Stanford University, Department of Computer Science, January 1995.

[Koza, 1991] John R. Koza. Genetic evolution and co-evolution of computer programs. In Christopher Taylor Charles Langton, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, volume X of *SFI Studies in the Sciences of Complexity*, pages 603–629. Addison-Wesley, Redwood City, CA, USA, 1991.

[Koza, 1992] John R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[Koza, 1994] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts, May 1994.

[Langdon, 1995] W. B. Langdon. Evolving data structures using genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 295–302, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Langdon, 1996a] W. B. Langdon. Scheduling maintenance of electrical power transmission networks using genetic programming. In John Koza, editor, *Late Breaking Papers at the GP-96 Conference*, pages 107–116, Stanford, CA, USA, 28–31 July 1996. Stanford Bookstore.

[Langdon, 1996b] William B. Langdon. Data structures and genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 20, pages 395–414. MIT Press, Cambridge, MA, USA, 1996.

[Montana, 1995] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[Nordin and Banzhaf, 1995] Peter Nordin and Wolfgang Banzhaf. Evolving turing-complete programs for a register machine with self-modifying code. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 318–325, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Nordin, 1994] Peter Nordin. A compiling genetic programming system that directly manipulates the machine code. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.

[O'Reilly and Oppacher, 1996] Una-May O'Reilly and Franz Oppacher. A comparative analysis of GP. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 2, pages 23–44. MIT Press, Cambridge, MA, USA, 1996.

[Park and Miller, 1988] Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 32(10):1192–1201, Oct 1988.

[Price, 1970] George R. Price. Selection and covariance. *Nature*, 227, August 1:520–521, 1970.

[Price, 1972] George R. Price. Fisher's 'fundamental theorem' made clear. *Annals of Human Genetics*, 36:129–140, 1972.

[Reynolds, 1994] Craig W. Reynolds. Competition, coevolution and the game of tag. In Rodney A. Brooks and Pattie Maes, editors, *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 59–69, MIT, Cambridge, MA, USA, 6-8 July 1994. MIT Press.

[Rocsa, 1996] Justinain Rocsa. GP population variety. GP electronic mailing list, 21 Jun 1996.

[Rosca and Ballard, 1996] Justinian P. Rosca and Dana H. Ballard. Discovery of subroutines in genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 9, pages 177–202. MIT Press, Cambridge, MA, USA, 1996.

[Rosca, 1995] Justinian P. Rosca. Genetic programming exploratory power and the discovery of functions. In John Robert McDonnell, Robert G. Reynolds, and David B. Fogel, editors, *Evolutionary Programming IV Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pages 719–736, San Diego, CA, USA, 1-3 March 1995. MIT Press.

[Ryan, 1994] Conor Ryan. Pygmies and civil servants. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 11, pages 243–263. MIT Press, 1994.

[Ryan, 1995] Conor Ryan. GPRobots and GPTeams - competition, co-evolution and co-operation in genetic programming. In E. S. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 86–93, MIT, Cambridge, MA, USA, 10–12 November 1995. AAAI.

[Sen, 1996] Sandip Sen. Adaptation, coevolution and learning in multiagent systems. Technical Report SS-96-01, AAAI Press, Stanford, CA, March 1996.

[Siegel, 1994] Eric V. Siegel. Competitively evolving decision trees against fixed training cases for natural language processing. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 19, pages 409–423. MIT Press, 1994.

[Singleton, 1994] Andy Singleton. Genetic programming with C++. *BYTE*, pages 171–176, February 1994.

[Stender, 1993] Joachim Stender, editor. *Parallel Genetic Algorithms: Theory and Applications.* IOS press, 1993.

[Syswerda, 1989] Gilbert Syswerda. Uniform crossover in genetic algorithms. In J. David Schaffer, editor, *Proceedings of the third international confernece on Genetic Algorithms*, pages 2–9, 10 Moulton Street, Cambridge, MA 02238, USA, Jun 1989. Morgan Kaufmann, San Mateo, California.

[Syswerda, 1991] Gilbert Syswerda. A study of reproduction in generational and steady state genetic algorithms. In Gregory J. E. Rawlings, editor, *Foundations of genetic algorithms*, pages 94–101. Morgan Kaufmann, San Mateo, 1991.

[Tackett, 1994] Walter Alden Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programs.* PhD thesis, University of Southern California, Department of Electrical Engineering Systems, 1994.

[Tackett, 1995] Walter Alden Tackett. Greedy recombination and genetic search on the space of computer programs. In L. Darrell Whitley and Michael D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 271–297, Estes Park, Colorado, USA, 31 July–2 August 1994 1995. Morgan Kaufmann.

[Teller, 1994] Astro Teller. The evolution of mental models. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 9, pages 199–219. MIT Press, 1994.