

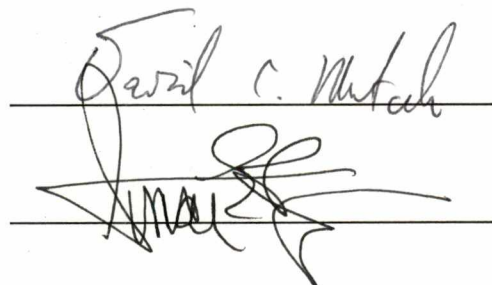
To the Graduate Council:

I am submitting herewith a thesis written by Norman Ma entitled *Performance of Simple Genetic Algorithm on Randomly Generated Problems*. I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.



Michael D. Vose, Major Professor

We have read this thesis and recommend its acceptance:



Accepted for the Council:



Vice provost
and Dean of The Graduate School

Performance of Simple Genetic Algorithm on Randomly Generated Problems

A Thesis
Presented for the
Master of Science
Degree

The University of Tennessee, Knoxville

Norman Ma

August 1990

Copyright©Norman Ma, 1990

Permission is granted to make and distribute verbatim copies and/or modified versions of this thesis provided the copyright notice and this permission notice are preserved on all copies and all versions.

Acknowledgment

I owe a great deal to the patience of my advisor, Michael D. Vose.

Abstract

This thesis presents a way to measure performance of simple genetic algorithm (SGA) and investigates the distribution of hard and easy problems for the SGA. We define a function that measures performance level of SGA by comparing best solution after each generation to the optimal solution. We test performance of SGA on random problems and investigate the relationship of the distribution of hard and easy problems to time, space, and performance.

Contents

1	Beginning Remarks	1
1.1	Table of Symbols	1
1.2	Formulas	2
1.3	Introduction	2
2	Project Description	3
2.1	Simple Genetic Algorithm	3
2.2	Fitness Functions	5
2.3	Performance Rating Functions	7
2.4	Performance Graph	8
2.5	Best Simple Genetic Algorithm	10
2.6	Convergence Definition	11
2.7	Iterative Averaging Method	15
3	Program Description	16
3.1	Dispatcher and Worker	19
4	Ending Remarks	21
4.1	Result	21
4.2	Result Analysis	22
4.3	Conclusion	60
4.4	Future Investigation	61
	References	62
	Appendix	64
	Vita	69

List of Figures

1	Simple Genetic Algorithm	4
2	Performance Graph	9
3	Choosing best SGA from a set of SGA of different crossover and mutations rates.	10
4	Convergence Relation	11
5	Program Parts	16
6	Approximation of Level Curve	22
7	Distance Between Level Curve and Origin	25
8	Radius Increases As String Increases	27
9	String Length 5 Performance Curves	29
10	String Length 6 Performance Curves	29
11	String Length 7 Performance Curves	30
12	String Length 8 Performance Curves	30
13	String Length 9 Performance Curves	31
14	String Length 10 Performance Curves	31
15	String Length 11 Performance Curves	32
16	String Length 12 Performance Curves	32
17	Predictions and Actual Data for $\ell = 13$ Performance Curves	33
18	Standard Deviation for $\ell = 12, \alpha = 0.4$	33
19	Standard Deviation for $\ell = 11, \alpha = 0.4$	34
20	Standard Deviation for $\ell = 10, \alpha = 0.4$	34
21	Standard Deviation for $\ell = 9, \alpha = 0.4$	35
22	Standard Deviation for $\ell = 12, \alpha = 0.6$	35
23	Standard Deviation for $\ell = 11, \alpha = 0.6$	36
24	Standard Deviation for $\ell = 10, \alpha = 0.6$	36
25	Standard Deviation for $\ell = 9, \alpha = 0.6$	37
26	Standard Deviation for $\ell = 12, \alpha = 0.8$	37
27	Standard Deviation for $\ell = 11, \alpha = 0.8$	38
28	Standard Deviation for $\ell = 10, \alpha = 0.8$	38
29	Standard Deviation for $\ell = 9, \alpha = 0.8$	39
30	Kurtosis for $\ell = 12, \alpha = 0.4$	40
31	Kurtosis for $\ell = 11, \alpha = 0.4$	40
32	Kurtosis for $\ell = 10, \alpha = 0.4$	41

33	Kurtosis for $\ell = 9, \alpha = 0.4$	41
34	Kurtosis for $\ell = 12, \alpha = 0.6$	42
35	Kurtosis for $\ell = 11, \alpha = 0.6$	42
36	Kurtosis for $\ell = 10, \alpha = 0.6$	43
37	Kurtosis for $\ell = 9, \alpha = 0.6$	43
38	Kurtosis for $\ell = 12, \alpha = 0.8$	44
39	Kurtosis for $\ell = 11, \alpha = 0.8$	44
40	Kurtosis for $\ell = 10, \alpha = 0.8$	45
41	Kurtosis for $\ell = 9, \alpha = 0.8$	45
42	Problem Distribution for $\ell = 12, \alpha = 0.8$	46
43	Problem Distribution for $\ell = 11, \alpha = 0.8$	47
44	Problem Distribution for $\ell = 10, \alpha = 0.8$	47
45	Problem Distribution for $\ell = 9, \alpha = 0.8$	48
46	Problem Distribution for $\ell = 8, \alpha = 0.8$	48
47	Problem Distribution for $\ell = 7, \alpha = 0.8$	49
48	Problem Distribution for $\ell = 6, \alpha = 0.8$	49
49	Problem Distribution for $\ell = 5, \alpha = 0.8$	50
50	Problem Distribution for $\ell = 12, \alpha = 0.6$	50
51	Problem Distribution for $\ell = 11, \alpha = 0.6$	51
52	Problem Distribution for $\ell = 10, \alpha = 0.6$	51
53	Problem Distribution for $\ell = 9, \alpha = 0.6$	52
54	Problem Distribution for $\ell = 8, \alpha = 0.6$	52
55	Problem Distribution for $\ell = 7, \alpha = 0.6$	53
56	Problem Distribution for $\ell = 6, \alpha = 0.6$	53
57	Problem Distribution for $\ell = 5, \alpha = 0.6$	54
58	Problem Distribution for $\ell = 12, \alpha = 0.4$	54
59	Problem Distribution for $\ell = 11, \alpha = 0.4$	55
60	Problem Distribution for $\ell = 10, \alpha = 0.4$	55
61	Problem Distribution for $\ell = 9, \alpha = 0.4$	56
62	Problem Distribution for $\ell = 8, \alpha = 0.4$	56
63	Problem Distribution for $\ell = 7, \alpha = 0.4$	57
64	Problem Distribution for $\ell = 6, \alpha = 0.4$	57
65	Standard Deviation Decreases as ℓ Increases	58
66	Problem Distribution for $\ell = 13, \alpha = 0.4$	59

67	Problem Distribution for $\ell = 13, \alpha = 0.6$	60
68	Problem Distribution for $\ell = 13, \alpha = 0.8$	60
69	Hierarchical Diagram.	67

1 Beginning Remarks

1.1 Table of Symbols

We use symbols listed below in this study. All sequences start with index 0. For example, initial population is $\mathcal{P}(0)$, first element of vector V is V_0 , first bit of string s is s_0 , and first term of a polynomial is term 0. Subscripts are indices. For example, 0th row 4th column component of array A is $A_{0,4}$ and 3rd bit of string s is s_3 . Objects in parenthesis are parameters to functions. For example, $\mathcal{F}(s)$ is fitness function with parameter s , and $\mathcal{P}(7)$ is population at 7th generation. For binary strings, least significant bit is the leftmost bit. For example, decimal interpretation of 010110 is 26.

Symbol	Meaning
c	Crossover rate.
m	Mutation rate.
ℓ	String length.
$\Omega(\ell)$	All possible binary strings of length ℓ .
g	Generation.
b	Population size.
ρ	Polynomial.
$\mathcal{P}$	Population.
$\mathcal{P}(i)$	Population associated with i th generation.
s	String.
$s(i)$	i th string in a population.
s_{best}	Best string produced by a SGA.
$s_{optimal}$	A string in $\Omega(\ell)$ having optimal fitness value.
v	Performance vector.
$v(i)$	Performance vector associated with i th polynomial.
$\mathcal{F}(\cdot)$	Fitness function.
$\mathcal{R}$	Performance rating function.
κ	Convergence rate.
n	Number of polynomials.
μ	Expected performance value.
$\mathcal{S}(\cdot)$	Selection operator.
$\mathcal{C}(\cdot \cdot)$	Crossover operator.
$\mathcal{M}(\cdot)$	Mutation operator.
$\mathcal{V}(\cdot)$	Convergence operator.
$(\cdot \wedge \cdot)$	AND operator.
$(\cdot \vee \cdot)$	OR operator.
$(\cdot \oplus \cdot)$	EXCLUSIVE-OR operator.

1.2 Formulas

Expected value, $\mathcal{E}(x)$, of random variable x is

$$\mathcal{E}(x) = \int_{-\infty}^{\infty} x f(x) dx = \int_{-\infty}^{\infty} x dF(x)$$

that is same as mean of x and is denoted by μ .

1.3 Introduction

Simple genetic algorithm (SGA)[5] is a method that attempts to optimize a non-negative function $\mathcal{F}$ (fitness function) over a finite domain of binary strings. For example, a string may represent a strategy for playing tic-tac-toe and fitness function could represent result of the strategy played against a player. Genetic search technique has found game playing strategies for Pente ¹, strategies for Traveling Salesman Problem [2], signal processing [1], optimized gas line flow control parameters. Recent research topics also include training of neural nets by genetic search. Representing solutions as binary strings and deciding which fitness function to use are left mainly to users' ingenuity. It is possible to apply genetic search to many problem domains, but no strong guideline exists for appropriateness of using genetic search on different problems and there has been no previous investigation into the distribution of hard and easy problems.

¹A 2 player game played on a 19×19 board that is similar to Go.

2 Project Description

It is acceptable to rate a SGA that finds an optimum solution better than a SGA that cannot find any solution. It is not such a clear-cut case when comparing a SGA that finds an acceptable solution early to a SGA that finds the optimal solution after many generations. Factors such as time, space, and quality of solution should be considered when rating these SGAs. We define a performance rating function that we hope is reasonable in measuring SGAs' performances. We like to know how both number of generations and population size influence SGA performance. We try to eliminate influence due to both mutation and crossover rates in order to simplify our investigation.

Without prior knowledge of distribution of SGA's performance rating on random problems, we need to know how many problems are needed to satisfactorily estimate SGA's expected performance. We prove a theorem that answers this question. Genetic search is applicable to a wide range of problems, and it may be difficult to generate enough kinds of problems to cover its possible usage. We thus decided to test SGA on random polynomials, since SGA is essentially optimizing a function over finite domain and this function can be represented as a polynomial. Lastly, we organize test results into tables that show how population size and number of generations relate to SGA's performance, and how performance relates to the distribution of hard and easy problems.

2.1 Simple Genetic Algorithm

A fitness function $\mathcal{F}$ maps strings to fitness values. SGA evolves a population of strings using probabilistic operators and feedback from $\mathcal{F}$, figure 1. Goal of SGA is to produce strings with high fitness values. Four parameters of SGA are crossover rate, mutation rate, population size, and number of generations. Two main parts of a SGA are

- Evaluate current population.
- Form next population from current population.

SGA generates the initial population randomly and uses probabilistic operators to form subsequent populations. Selection operator selects string from a population with probability of picking a particular string same as ratio of its fitness to the total

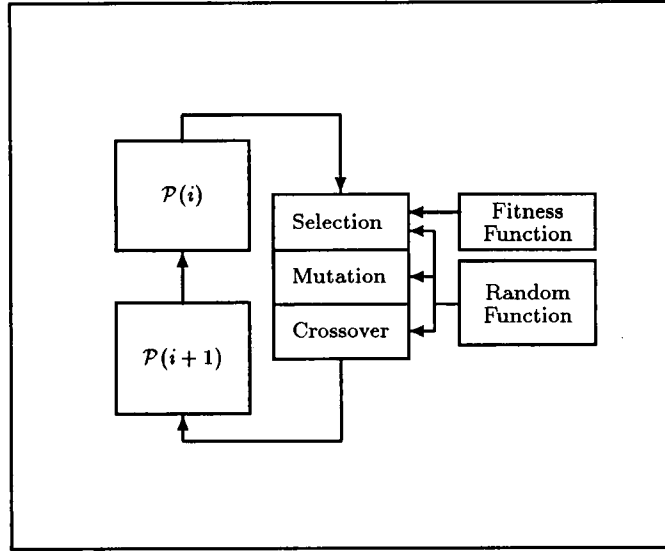


Figure 1: Simple Genetic Algorithm

population fitness. Crossover operator randomly selects a crossover site along two strings and with a certain probability exchanges their tails. Mutation operator flips each bit of a string with a certain probability. SGA applies these three operators to current population and collects the resulting strings as the next population. This procedure stops when population size is reached. SGAs have constant population size from generation to generation. Let these probabilistic operators be $\mathcal{S}(\cdot)$ – selection, $\mathcal{C}(\cdot, \cdot)$ – crossover, and $\mathcal{M}(\cdot)$ – mutation. Let $\mathcal{F}(\cdot)$ be fitness function.

We describe components of SGA and give an algorithmic description of the SGA process.

- b : Population size. Simple Genetic Algorithm has constant population size.
- $\mathcal{P}(0)$: Form the initial population randomly. For example, if string length is ℓ and population size b , use b randomly generated binary strings of length ℓ .
- $\mathcal{M}(s)$: For each bit of s , mutation operator flips that bit with probability m .
- $\mathcal{S}(\mathcal{P})$: Selection operator randomly picks a string $s(i)$ from $\mathcal{P}$ with probability of picking that string as

$$\frac{\mathcal{F}(s(i))}{\sum_{j=0}^{b-1} \mathcal{F}(s(j))}$$

- $\mathcal{C}(s(i), s(j))$: With probability c , crossover operator will uniformly select a random crossover site in $[0 \dots \ell-2]$ and exchange tail portions of $s(i)$ and $s(j)$. For example, if we cross $s(i) = 11111111$ and $s(j) = 00000000$ at position 2, the resulting pair of strings are $s(i) = 11100000$ and $s(j) = 00011111$. The SGA algorithm is

```

1       $\mathcal{P}(0) \leftarrow$  random population
2      For generation  $g = 0$  do
3          For every  $s \in \mathcal{P}(g)$ , calculate  $\mathcal{F}(s)$ 
4           $\mathcal{P}(g+1) \leftarrow \emptyset$ 
5          Repeat
6               $\mathcal{P}(g+1) \leftarrow \mathcal{P}(g+1) \cup \mathcal{M}(\mathcal{S}(\mathcal{P}(g)))$ 
7          Until size of  $\mathcal{P}(g+1)$  is  $b$ 
8          Take 2 strings from  $\mathcal{P}(g+1)$  systematically
9               $\mathcal{P}(g+1) \leftarrow \mathcal{C}(s(j), s(k))$ 
10         Until all strings have been through crossover
11          $\mathcal{P}(g) \leftarrow \mathcal{P}(g+1)$ 
12     Until some stopping criteria is reached

```

A small change to SGA algorithm ensures the best string in a population survives to the next population; the best string of a population randomly replace one string in the new population. This change is reasonable since it is desirable to keep best solution generated by SGA. Result of a SGA is the best string from the final population. When selecting strings for crossover and mutation, probability of a string being picked is proportional to ratio of its fitness to the total population fitness.

2.2 Fitness Functions

Any function over a finite domain can be represented by a polynomial. Let x be a binary number and x_i refers to i th bit of x . For example,

$$f(x) = 0.1980 + 2.520x_0 + 17.36x_0x_1 + 145.7x_0x_2 + 930.2x_0x_1x_2 + 8100x_0x_3$$

can represent e^x for odd values of x in $[1, 9]$. Using polynomials eliminates the difficulty of generating different problem types since the functional form is set. One may therefore investigate problem solving ability of SGAs by applying them to polynomials. All fitness functions in this project are polynomials. Let $a(i)$ be

a real number in $[-\frac{1}{2}, \frac{1}{2}]$ and x_j be a binary number, define a length ℓ polynomial $\rho(x_0, \dots, x_{\ell-1})$ as,

$$\rho(x_0, \dots, x_{\ell-1}) = \sum_{J \subseteq \{x_0, \dots, x_{\ell-1}\}} a(J) \prod_{x \in J} x$$

where $a(J)$ is a *coefficient*, $\prod_{x \in J} x$ is a *mask*, and a coefficient and a mask together form a *term*, $a(J) \prod_{x \in J} x$. Let ℓ be string length. By regarding binary strings as the binary representation of numbers, all polynomials in this project have domain in $[0 \dots 2^\ell - 1]$ and co-domain in $[-2^{\ell-1}, 2^{\ell-1}]$.

There are 2^ℓ unique masks for length ℓ polynomials, thus there are 2^ℓ unique terms for a ℓ length polynomial. If each coefficient of a polynomial is a coordinate on an axis out of a set of 2^ℓ mutually perpendicular axes, then ℓ -length polynomials can be represented as points in a 2^ℓ -dimensional cube. Let *full-term* describe polynomials that have all non-zero coefficients. Using this polynomial space model, all points that are in interior of the cube are full-term polynomials. There are many more full-term polynomials than other polynomials from a measure theoretic point of view. We explore only full-term polynomials in this project. Evenly distributed points in the cube represent polynomials with uniformly generated coefficients.

SGA compares a string with masks of each term of the polynomial to find its fitness value. Let s be a string. If the “on” bits of a mask is subset of “on” bits in s , then SGA adds the corresponding coefficient to the result. Matching process can be done with following test condition that returns true only when result of bit-wise AND of s and mask is same as the mask,

$$mask = (mask \text{ bit-wise-and } s)$$

For example, if a length 3 polynomial is

$$\rho(s) = 0 + 0.30x_0 + 0.033x_1 - 0.078x_0x_1 + 0.38x_2 - 0.25x_0x_2 + 0.0076x_1x_2 + 0.42x_0x_1x_2$$

and $s = 011$, since 0th term has mask 000 and $(000 = (000 \text{ bit-wise-and } 011))$ is true, SGA includes coefficient 0 in the result. The first term’s mask is 100, and SGA does not add 0.30 to the result since $(100 = (100 \text{ bit-wise-and } 011))$ is false. Repeating for rest of the terms and fitness value is $0.0 + 0.033 + 0.0076 + 0.011 = 0.4316$ for string s .

Since representing polynomials using coefficients and masks is not efficient in a program, SGA uses a 2^ℓ -sized fitness table whose indices correspond to all

possible strings and contents are their fitness values to represent a polynomial. SGA fills the fitness table directly as each coefficient is generated. This method saves memory space otherwise needed to store coefficients and masks; we have eliminated a function that evaluates polynomial and SGA can now access all fitness values in constant time. The fitness table *ftable* filling routine is

```

For mask = 1 to  $2^\ell - 1$ 
  coeff = random( $-\frac{1}{2}, \frac{1}{2}$ )
  For value = 1 to  $2^\ell - 1$ 
    if (mask = (mask bit-wise-and value))
      ftablevalue = ftablevalue + coeff

```

mask in the code starts iteration from 1 because the test condition is always true when *mask* is 0. Thus the corresponding coefficient is added to all values and result is just shifting a polynomial vertically without changing its shape. *value* in the code above also starts iteration from 1 because the test condition is always false when *value* is 0. After filling *ftable*, SGA subtracts smallest number in *ftable* from every component of *ftable*. Subtracting smallest value shifts polynomial up if the value is negative or shifts polynomial down if smallest value is positive. The resulting polynomial has 0 as its lowest value. This operation assures a non-negative fitness function and SGA looks up this table for all its fitness value computations.

2.3 Performance Rating Functions

If SGA has a tough time optimizing a “hard” problem, , the same problem could be considered “easy” if SGA increases its population size and number of generations used on the problem; one can usually change intuitive notion of a polynomial as easy or hard by change space (population size) and time (number of generations) used by SGA. Thus, a reasonable performance rating will take all components of SGA into consideration. A rating function measures performance after each generation of a SGA. When SGA has optimized large percentage of random problems, its performance rating is close to 1, while a value close to 0 means SGA did not perform as well. Since initial population is random, initial average population fitness is around the expected fitness. Let ℓ be string length and all possible length ℓ strings be binary interpretation of integers from 0 through $2^\ell - 1$. The expected value is

$$\mu = \frac{1}{2^\ell} \sum_{s=0}^{2^\ell-1} \mathcal{F}(s)$$

To measure improvement from initial random population, we need to neutralize this tendency. We therefore subtract expected fitness in the rating function. the rating function is non-decreasing because the program guarantees that best string in a population appears in next generation. This rating function measures how close does best fitness encountered come to the optimal fitness,

$$\mathcal{R} = \frac{\mathcal{F}(s_{best}) - \mu}{\mathcal{F}(s_{optimal}) - \mu} \quad (1)$$

$\mathcal{R}$ returns a vector of values where each component corresponds to a performance value after a generation for generation 0 through $g - 1$. We call this vector the performance vector and denote it by v . For example, v_0 is performance rating of the initial random generation and v_g is the performance rating after the g th generation has completed. If no generation number is mentioned, we may then regard $\mathcal{R}(SGA)$ as a number, namely performance value of the last generation that it has completed.

2.4 Performance Graph

Population size and number of generations used are two parameters that have large influence on SGA's performance. It is more likely to encounter an optimal string using a large population. It is also more likely to encounter an optimal string using many generations. Increasing space (population size) or increasing time (generations) can improve SGA performance, and decreasing space and time allowed for SGA has a negative effect on SGA's performance. Considering space and time as resources, we like to know the relationship between these two parameters and what influence they may have on the distribution of hard and easy problems..

Let $\mathcal{R}$ denote a rating function. We may regard $\mathcal{R}(SGA)$ as a random variable since the performance of SGA may vary depending on the fitness function it is trying to optimize. Expected performance value $\mu = \mathcal{E}(\mathcal{R}(SGA))$ describes expected performance of a SGA. If μ is near 1 then SGA has optimized most polynomials². If μ is near 0, then the SGA has not performed well on most polynomials. Since μ is a function of SGA, both population size and number of generations affect the expected performance. A genetic algorithm that has insufficient resources is not interesting. On the other extreme, providing time and space required to make *every* problem easy would make genetic search inefficient. Therefore determining

²Polynomials represent fitness functions.

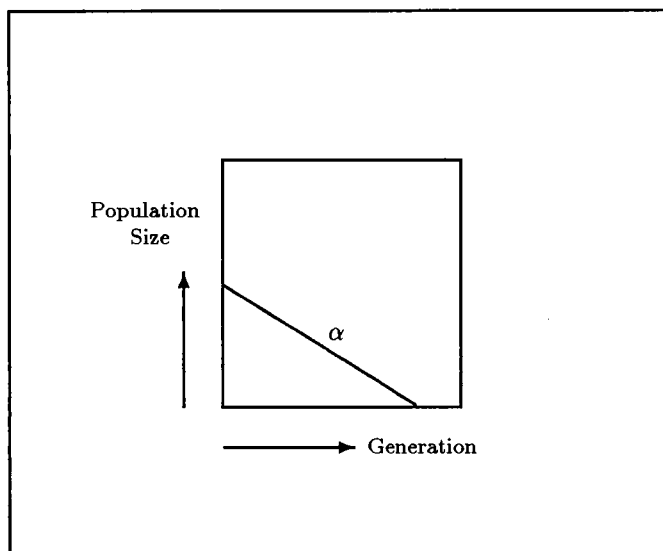


Figure 2: Performance Graph

the relationship between population size (space) and number of generations (time) that avoids these extremes would allow a range of performance values and would make an investigation of the distribution of hard and easy problems reasonable.

Let α be a performance value that avoids either SGA optimizes all random polynomials successfully or none. Many combinations of space and time may induce a particular performance value. For example, if a SGA with population size 5 and generation number 30 has performance rating α , it is likely to reach the same performance level using a larger population size and a smaller generation number. If we consider a graph with vertical axis as population size and number of generations as horizontal axis, then we can identify a curve in this “performance graph” along which $\mu = \alpha$, see figure 2. Additionally, if performance distributions have similar shapes along the curve, we may then conclude that time and space is an even trade for SGA. Time may be traded for space by moving along the curve since performance rating of SGA is constant along it. Trading time for space would be “even” in the sense that varying population size or number of generations does not affect SGA’s performance.

We eliminate effects of crossover and mutation rates by attempting to choose the best values for these parameters given population size and number of generations. Since relationship between these rates and SGA performance is not clear,

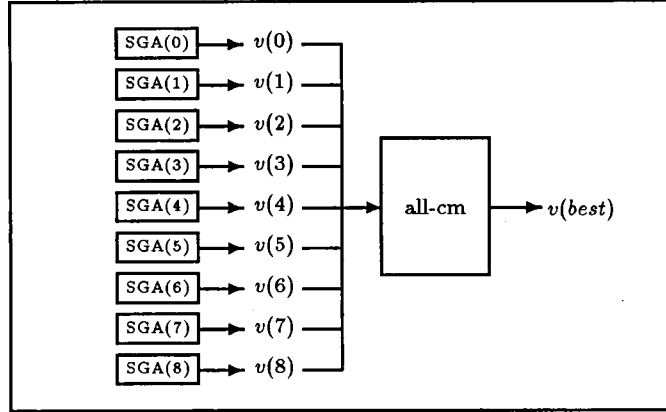


Figure 3: Choosing best SGA from a set of SGA of different crossover and mutations rates.

we compromise by trying various rates and using the best result.

Scope of random polynomials includes all possible polynomials of a particular string length described in the Fitness Function section 2.2. Investigating a different string length is a change of “problem size”. This investigation will start with string length 4 and increase problem size until computation is no longer feasible.

2.5 Best Simple Genetic Algorithm

Population size, number of generations, mutation rate, and crossover rate are four parameters that affect performance of SGA. To concentrate on population size and number of generations, we will eliminate effects of mutation and crossover rates by using a SGA that represents the best performance from a set of SGAs with different crossover and mutation rates figure 3. We pick a best SGA from a set of SGAs for each random problem then we investigate properties of these best SGAs generated from random problems. Let C be a set of crossover rates and M be a set of mutation rates that we will use,

$$C = \{0.4, 0.6, 0.8\}$$

$$M = \{0.1, 0.05, 0.01\}$$

There are 9 combinations of different rates. Result of a SGA experiment is a vector v of performance values where v_i corresponds to a performance value after

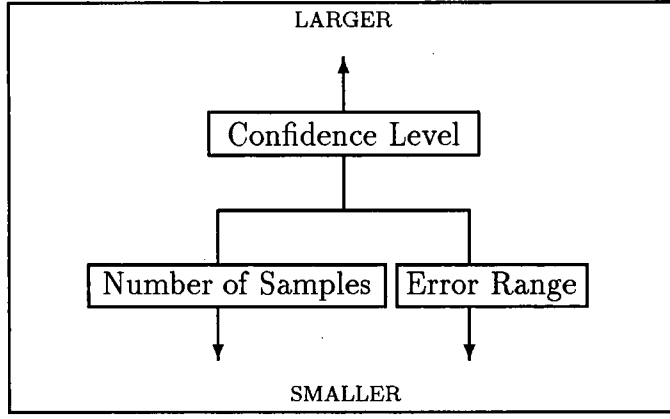


Figure 4: Convergence Relation

a generation. To produce a best SGA performance vector $v(best)$ from 9 SGA performance vectors, $v(0), v(1), \dots, v(8)$, each component of the best vector is the best value from corresponding components of the 9 vectors. Let g be number of generations that a SGA carried out and $i \in [0, 1, \dots, g-1]$, the

$$v(best)_i = \max_{j \in [0, \dots, 8]} v(j)_i$$

is the best vector. This approach avoids the difficult dilemma of deciding which specific one of the 9 performance vectors is best since it is not likely that a single vector has winning value at every generation. When we reference to performance vector of SGA on a problem, we mean this best performance vector extracted from the 9 actual SGAs.

2.6 Convergence Definition

How many samples of a bounded random variable from an unknown distribution assure a good estimate of its expectation qualified with error range and confidence level figure 4? We use following method for estimating expectation of a random variable with unknown distribution. Let $\xi_1, \xi_2, \dots$ be independent identically distributed random variables, and let $\xi = \sum_{k=1}^n \xi_k$. $\frac{1}{n}\xi$ is an estimate for expectation of ξ . Without knowing the probability distribution of ξ_k , we want to know how large n should be so that empirical mean satisfies,

$$\text{Prob}[\text{error} \geq 0.1] \leq 0.05$$

where 0.05 corresponds to 95% confidence.

Lemma 1 (Markov's Inequality): Let $\xi \geq 0$ be a random variable with $\mathcal{E}(\xi) < \infty$. Then for every $\lambda \geq 1$,

$$\text{Prob} [\xi \geq \lambda \mathcal{E}(\xi)] \leq \frac{1}{\lambda} \quad (2)$$

Proof:

$$\text{Let } m = \mathcal{E}(\xi) = \int_0^\infty x dF(x)$$

then

$$m \geq \int_{\lambda m}^\infty x dF(x) \geq \lambda m \int_{\lambda m}^\infty dF(x) = \lambda m(1 - F(\lambda m))$$

Q.E.D.

Theorem 1: Let ξ be a random variable with expectation M . Then for $\varepsilon > 0$, $t > 0$, if $\mathcal{E}(e^{\varepsilon(\xi-M)}) < \infty$, then

$$\text{Prob} \left[\xi - M \geq \frac{t + \ln \mathcal{E}(e^{\varepsilon(\xi-M)})}{\varepsilon} \right] \leq e^{-t} \quad (3)$$

Proof: Let $\eta = e^{\varepsilon(\xi-M)}$ and apply Markov inequality to get (with $\lambda = e^t$)

$$\text{Prob} [e^{\varepsilon(\xi-M)} \geq e^t \mathcal{E}(e^{\varepsilon(\xi-M)})] \leq e^{-t}$$

Since the logarithm is monotone, we get

$$\text{Prob} [\varepsilon(\xi - M) \geq t + \ln \mathcal{E}(e^{\varepsilon(\xi-M)})] \leq e^{-t}$$

Q.E.D.

Theorem 2: let $\xi_1, \xi_2, \dots$ identically distributed, independent random variables either taking values in real interval $[0, 1]$ or taking values in real interval $[-1, 0]$. For $n \geq 1$, $\varepsilon > 0$, $t > 0$

$$\text{Prob} \left[\frac{1}{n} \sum_{j=1}^n \xi_j - \mu \geq \frac{t + n \ln(1 + \frac{1}{4}(e^\varepsilon - 1 - \varepsilon))}{n\varepsilon} \right] \leq e^{-t} \quad (4)$$

where $\mu = \mathcal{E}(\xi_j)$.

Proof: Let $\xi = \sum_{j=1}^n \xi_j$. Note that $\mathcal{E}(\xi) = n\mu$ since ξ_j are identically distributed.

$$\mathcal{E}(e^{\varepsilon(\xi-n\mu)}) = \mathcal{E}(e^{\varepsilon(\sum_{j=1}^n (\xi_j - \mu))}) = \mathcal{E}\left(\prod_{j=1}^n e^{\varepsilon(\xi_j - \mu)}\right) = \prod_{j=1}^n \mathcal{E}(e^{\varepsilon(\xi_j - \mu)})$$

$$\begin{aligned}
&= \left\{ \sum_{k=0}^{\infty} \mathcal{E} \left(\frac{\varepsilon^k (\xi_j - \mu)^k}{k!} \right) \right\}^n \\
&= \left\{ \mathcal{E}(1) + \mathcal{E}(\varepsilon(\xi_j - \mu)) + \mathcal{E} \left(\frac{\varepsilon^2 (\xi_j - \mu)^2}{2} \right) + \sum_{k=3}^{\infty} \mathcal{E} \left(\frac{\varepsilon^k (\xi_j - \mu)^k}{k!} \right) \right\}^n \quad (5)
\end{aligned}$$

Note for $k > 2$

$$\mathcal{E} \left(\frac{\varepsilon^k (\xi_j - \mu)^k}{k!} \right) = \frac{\varepsilon^k}{k!} \int (\xi_j - \mu)^2 (\xi_j - \mu)^{k-2} dF(x) \leq \frac{\varepsilon^k}{k!} \sigma^2$$

Since $|\xi_j - \mu| \leq 1$. Therefore equation (5) is bounded from above by

$$\begin{aligned}
&\left\{ 1 + 0 + \frac{\varepsilon^2 \sigma^2}{2} + \sum_{k=3}^{\infty} \frac{\varepsilon^k \sigma^2}{k!} \right\}^n \\
&= \left\{ 1 + \frac{\varepsilon^2 \sigma^2}{2} + \sigma^2 (e^\varepsilon - 1 - \varepsilon - \frac{\varepsilon^2}{2}) \right\}^n
\end{aligned}$$

Since $0 \leq \xi_k \leq 1$ or else $-1 \leq \xi_k \leq 0$, we have $\sigma^2 \leq \frac{1}{4}$. Therefore

$$\begin{aligned}
\mathcal{E}(e^{\varepsilon(\xi - n\mu)}) &\leq \left\{ 1 + \frac{\varepsilon^2}{8} + \frac{1}{4}(e^\varepsilon - 1 - \varepsilon - \frac{\varepsilon^2}{2}) \right\}^n \\
&= (1 + \frac{1}{4}(e^\varepsilon - 1 - \varepsilon))^n
\end{aligned}$$

Applying theorem 1 to ξ and using our estimate for $\mathcal{E}(e^{\varepsilon(\xi - M)})$, we get

$$\text{Prob} \left[\frac{1}{n} \sum_{j=1}^n \xi_j - \mu \geq \frac{t + n \ln(1 + \frac{1}{4}(e^\varepsilon - 1 - \varepsilon))}{n\varepsilon} \right] \leq e^{-t}$$

Q.E.D.

Theorem 3: For $\xi_1, \xi_2, \dots$ identically distributed, independent random variables in $[0, 1]$, for $n \geq 1$, $\varepsilon > 0$, $t > 0$

$$\text{Prob} \left[\left| \frac{1}{n} \sum_{j=1}^n \xi_j - \mu \right| \geq \frac{t + n \ln(1 + \frac{1}{4}(e^\varepsilon - 1 - \varepsilon))}{n\varepsilon} \right] \leq 2e^{-t} \quad (6)$$

where $\mu = \mathcal{E}(\xi)$.

Proof: By theorem 2 we have

$$\text{Prob} \left[\frac{1}{n} \sum_{j=1}^n \xi_j - \mu \geq \frac{t + n \ln(1 + \frac{1}{4}(e^\varepsilon - 1 - \varepsilon))}{n\varepsilon} \right] \leq e^{-t}$$

Now apply theorem 2 to the random variables $-\xi_1, -\xi_2, \dots$ to get

$$\text{Prob} \left[-\frac{1}{n} \sum_{j=1}^n \xi_j + \mu \geq \frac{t + n \ln(1 + \frac{1}{4}(e^\varepsilon - 1 - \varepsilon))}{n\varepsilon} \right] \leq e^{-t}$$

Therefore

$$\text{Prob} \left[\left| \frac{1}{n} \sum_{j=1}^n \xi_j - \mu \right| \geq \frac{t + n \ln(1 + \frac{1}{4}(e^\varepsilon - 1 - \varepsilon))}{n\varepsilon} \right] \leq 2e^{-t}$$

Q.E.D.

We want to be at least 95% confident that 0.1 is the maximum error from n samples of the random variable. To solve for n , apply the criteria to equation (6)

$$\text{Prob} \left[\left| \frac{1}{n} \sum_{j=1}^n \xi_j - \mu \right| \geq \frac{t + n \ln(1 + \frac{1}{4}(e^\varepsilon - 1 - \varepsilon))}{n\varepsilon} = 0.1 \right] \leq 2e^{-t} = 1 - 0.95$$

Combine the two equalities we have n as a function of ε

$$n = \min_{\varepsilon} f(\varepsilon) = \min_{\varepsilon} \left(\frac{\ln(\frac{1-0.95}{2})}{\ln(1 + \frac{1}{4}(e^\varepsilon - 1 - \varepsilon)) - 0.1\varepsilon} \right)$$

Minimize $f(\varepsilon)$ and take ceiling of the result, we have $n = 198$. Our goal is to use as low a n as possible for a reasonable data gathering time, while keep error rate and confidence factor at an acceptable level.

Let χ be confidence level in $[0.0, 1.0]$, ψ be maximum error in $[0.0, 1.0]$, and n as samples of random variable. For number of samples as function of confidence level and maximum error,

$$n = \min_{\varepsilon} \left(\frac{\ln(\frac{1-\chi}{2})}{\ln(1 + \frac{1}{4}(e^\varepsilon - 1 - \varepsilon)) - \psi\varepsilon} \right)$$

For maximum error as function of number of samples and confidence level,

$$\psi = \min_{\varepsilon} \left(\frac{n \ln(1 + \frac{1}{4}(e^\varepsilon - 1 - \varepsilon)) - \ln(\frac{1-\chi}{2})}{\varepsilon n} \right)$$

For confidence level as function of maximum error and number of samples,

$$\chi = \max_{\varepsilon} \left(1 - 2e^{(n \ln(1 + \frac{1}{4}(e^\varepsilon - 1 - \varepsilon)) - n\psi\varepsilon)} \right)$$

Below is a table illustrating tradeoffs between confidence level and error rate when 198 samples are used.

Error	Confidence level
0.12	99.0%
0.10	95.0%
0.092	90.0%

2.7 Iterative Averaging Method

Iterative Averaging Method runs SGAs associated with the 198 random polynomials until their average performance value is greater than a set value. Let $v(i)$ be a performance vector associated with i th polynomial described in the Simple Genetic Algorithm section. Denote $v(i)_0$ as performance rating of initial generation, $v(i)_k$ as performance rating for k th generation. Let $L(\cdot)$ be a function that takes any performance vector v and returns the largest generation number for which v has a performance value. For generation k greater than generation j , $v_k \geq v_j$, because SGA guarantees best string in a generation will survive to the next generation. When $g > L(v)$, define v_g as $v(L(v))$. Let g be a generation number and $j \in [0, \dots, n-1]$,

$$A(g) = \frac{1}{n} \sum_{i=0}^{n-1} v(i)_g$$

$$A'(g, j) = \frac{1}{j+1} \sum_{i=0}^j v(i)_g$$

$A(\cdot)$ is generation average and $A'(\cdot, \cdot)$ is partial generation average. Let j be the smallest generation number that satisfies $A(j+1) > \alpha$. A performance vector v is "extended" by applying SGA to population $\mathcal{P}(L(v))$. Iterative averaging method has four steps,

Step 1 Extend $v(0)$ until $v(0)_{L(v(0))} \geq \alpha$.

Step 2 Extend $v(1)$ through $v(n-1)$ until generation g where either $v(j)_g \geq \alpha$ or $A'(g, j) \geq \alpha$ is true.

Step 3 Choose a $v(j)$ so j satisfies

$$L(v(j)) = \min_{i \in \{0, \dots, n-1\}} L(v(i))$$

and let ϖ be $L(v(j))$. Extend $v(j)$ to generation $\varpi+1$, and calculate $A(\varpi+1)$. Repeat step 3 until $A(\cdot)$ exceeds α .

Step 4 Extend all performance vectors so their length is at least $\varpi+1$.

3 Program Description

The program contains 5 parts,

- SGA.
- Best SGA.
- Iterative averaging data collector.
- Random polynomial generator.
- Distributed Processing Interface.

At center of this program, figure 5, is implementation of the SGA algorithm described in section 2.1, Simple Genetic Algorithm. Function `all-cm()` calls SGA with different mutation and crossover rates. `all-cm()` returns the best SGA result as described in section 2.5, Best Simple Genetic Algorithm Definition. Next layer up, `get-data()` calls `all-cm()` and implements iterative averaging technique. It also interfaces to the outside world in setting string length, population size, number of polynomials, rating function to use, etc. Global variables and data structures usage cut down on parameter passing. Some characteristics of this program include representing all strings as unsigned integers, bit-wise crossover and mutation operators, direct generation of polynomials look-up table, small number of function calls, iterative implementation of an algorithm, and using arrays instead of lists. Performance is the primary goal and program correctness is the second equally important consideration. We have not prove the program's correctness, but we have traced through test cases using the stepping utility to provide confidence for correctness of the program.

We carry out the above testing methods for every questionable result. A recovery functionality will restart the program at where it was unexpectedly terminated. If program has not generate all the polynomials, it will first complete generating rest of the polynomials. If the program stops while in the iterative averaging routine, it will first read in polynomial extension and performance results from the polynomial files that were saved previously. The recovery routine fills in two data structures that used by the iterative averaging method. `*pxr*` contains the generation that each polynomial has been extended. `*r-ary*` contains the performance values of every polynomial. These two structures are kept in the memory and are lost when the process is killed.

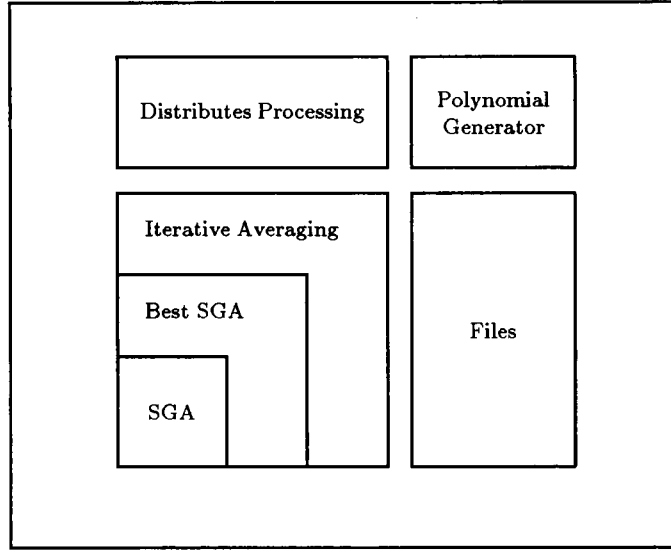


Figure 5: Program Parts

Most significant part of the program is the SGA modules. These modules implement algorithm described in the Simple Genetic Algorithm section. Look-up table represents polynomials economically; fitness value of any string is available with a single array reference. We have also tried to increase the efficiency of crossover and mutation operator. The program uses a mask and logical operators to carry out crossover on integers. Specifically, let i be the crossover site, ℓ be string length, and define crossover mask $M(i)$ as $2^{i+1} - 1$. Let $s(1)$ and $s(2)$ be strings, $\wedge$ and $\vee$ be bit-wise AND and OR operators, then following operations carries out the crossover operation,

$$s(1) \leftarrow ((s(1) \wedge M) \vee (s(2) \wedge (\text{not}M)))$$

$$s(2) \leftarrow ((s(2) \wedge M) \vee (s(1) \wedge (\text{not}M)))$$

For example, let $s(1)$ be 11001100, $s(2)$ be 10101010, and crossover site at position 2 ($M(2)=11100000$), then new strings are

$$11001010 \leftarrow ((11001100 \wedge 11100000) \vee (10101010 \wedge 00011111))$$

$$10101100 \leftarrow ((10101010 \wedge 11100000) \vee (11001100 \wedge 00011111))$$

Initialization routine creates a mask table that contains all crossover masks and its inverse for each string length. This method does crossover in 6 basic operations.

All strings are stored as integers and binary values used above are for illustrative purposes only. Mutation operator changes a string by EXCLUSIVE-ORing 1 to each bit of the string inside a loop with mutation probability. Let $\text{random}(1.0)$ return a floating point number in $[0 - 1)^3$, let $ctable_i$ be 2^i , let m be mutation rate, and let $\oplus$ be EXCLUSIVE-OR operator. Following code carries out mutation on a string.

```

For  $i = 0$  to  $\ell - 1$ 
    if  $\text{random}(1.0) < m$ 
         $string = string \oplus ctable_i$ 

```

Data collector includes data structures for storing values need for calculating performance ratings and routines to initialize SGA environment and implement iterative averaging method. It is also the interface for setting SGA's string length, number of polynomials to generate for testing, rating function to use, population size, random polynomial parameters. In addition, data collector also reads and writes initialization and result files. A large portion of data collector is implements iterative averaging method described in Iterative Averaging Method section. The method requires stopping an ongoing SGA and resuming it at a later time. To accomplish stop and restart requirement, SGA saves its environment for later restart or for continuation after unexpected termination. This requires a large amount of space considering that there are 9 SGAs working on each random polynomial. Each of 9 SGAs have different crossover and mutation rates for obtaining upper bound on performance of SGA on a polynomial. 198 random polynomials are need to reach a desired confidence level as stated in the Convergence Definition section. That is a total of $9 \times 198 = 1782$ SGAs that need to be saved for each single row of performance matrix, a substantial amount. Structure for saving a state includes performance results of 9 SGAs and fitness function related information. The structure includes

- Generation number. Last generations each of the 9 SGAs have completed.
- Array of 9 SGA states (each state includes crossover rate, mutation rate, and population).
- Performance vectors of 9 SGAs.
- Population size.

³This interval includes 0.0 but 1.0 is not included.

- String length.
- Fitness function look-up table.
- Average fitness values of $\Omega(\ell)$.
- Optimal fitness.

3.1 Dispatcher and Worker

It is time consuming to collect a performance graph. For example, it takes average about 12 hours on the *Sun* model 3/50 to collect one row of the length 10 performance graph, and it takes average of 56 hours to fill a single row of the length 11 performance graph, with even longer time for small population sizes. Therefore without parallel processing it would not be feasible to collect data for interesting string lengths. Dispatcher and workers are processes that collect data to fill the performance matrix. The physical manifestation of dispatcher and each worker are `dispatch.lsp` and `ga.lsp` respectively. Dispatcher assigns jobs to available workers. Each job is a command for a worker to gather data for one row of the performance matrix, and each process (both workers and dispatcher) occupies one of the thirty-one available *Sun* workstations connected via Ethernet. A worker by itself can collect data to fill the performance matrices. Reason for distributed processing is to speed up data gathering time. Dispatcher keeps status of each worker in an array of records where each record contains following information about each worker,

NAME Name of the processor.

READY Indicates if the processor is ready for new job.

JOB Job carrying out by the processor right now.

ST Starting time.

ET Elapsed time.

Dispatcher first reads in a file containing processor names and start workers on processors. Dispatcher then read in a file containing jobs and assign jobs to available workers in the following manner

```

1  initialize
2  start all processors
3  DO UNTIL job list is empty AND all outstanding jobs are done
4      find all available processors
5      save results from finished jobs
6      UNLESS jobs list is empty
7          assign jobs
8          send jobs

```

A complementary interface exists in each worker

```

1  initialize
2  DO UNTIL received stop signal from dispatcher
3      LOOP UNTIL receiving a command from dispatcher
4          calculate result
5          send result to dispatcher

```

Dispatcher initializes all ready fields of worker records are to nil, indicating workers are not ready to receive jobs. Dispatcher waits for a set time for workers to initialize and send ready signal. After receiving ready signals from workers and change each corresponding worker record's ready field to t, indicating the worker is ready to receive a job. Dispatcher fills in job field using jobs from job list and sends jobs to workers. Dispatcher terminates when job list is empty and all outstanding jobs are completed. After sending ready signal to dispatcher, a worker settles in a loop waiting for job command from dispatcher, carries out the job, and sends result to dispatcher. A worker repeats this loop until it receives a stop signal from dispatcher. At that point, worker terminates. The table below details how dispatcher modifies records. If pattern under the Pattern column appears, function listed in the first column will take action listed under action column and change record to the pattern under Set column. For example, dispatcher goes through the array and look for records with ready field equals to t and job field equals nil. After finding a match, dispatcher assigns a job to worker by filling in the job field, while leaving other fields alone.

	Pattern			Set			Action
	ready	job	result	ready	job	result	
F4	nil	-	exists	t	-	-	
F5	t	-	exists	-	nil	remove	Save result
F6	t	nil	-	-	fill	-	Assign job to worker
F7	t	not nil	-	nil	-	-	Send job to worker

Worker saves intermediate files in local processor's file system to avoid jamming the network. Only essential messages pass through the network.

4 Ending Remarks

Simple Genetic Algorithm is a widely applicable optimization method. We have formulated performance rating functions to solidify intuitive notion of performance by SGAs. We then test SGAs' performance on random problems. SGA works on polynomials to simplify problem generation process and to cover the SGA problem gamut. We tried to eliminate effects of mutation and crossover rates by choose a best SGA from a set of SGAs having different mutation and crossover rates. We present results using performance graphs where its horizontal axis shows time used by SGA and vertical axis shows space used by SGA. We also devised a model that describes relationship between time, space, problem size, and performance values. We use following symbols in result analysis,

Symbol	Meaning
t	Time (Generation).
s	Space (Population size).
α	Performance value.
ℓ	String length (Problem Size).
p, q	Level curve parameters.
r	Distance from a level curve to origin.
$a-h$	Parameters describing relation between p and q .
x, y	Generic horizontal and vertical axes values.

4.1 Result

We have collected data listed in the table below. Each combination of length and population size yields a vector where vector component is a performance value at completion of a generation of SGA. For example, there are 18 vectors collected for string length 7 problems.

Length	Population Size
4	2-6
5	2-6
6	2-14
7	2-19
8	2-22
9	2-29
10	2-30
11	2-30, 33
12	3-45
13	3, 5-23, 25-30, 32, 34, 36, 38, 39, 41, 42

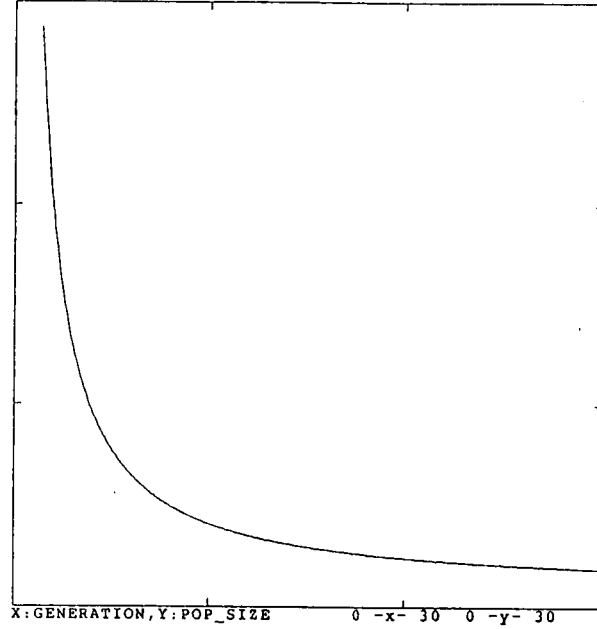


Figure 6: Approximation of Level Curve

4.2 Result Analysis

Let generation be time used by SGA and population size be space used by SGA, our goal is to investigate the relation between time, space, performance level and distribution of hard and easy problems for SGA. We first reorganize results from experiments into sets of time, space, performance, and problem size values, (t, s, α, ℓ) . We use performance graphs to describe relation between time and space. Each performance graph is a plot of performance value as function of time and space (time on the horizontal axis and space along the vertical axis) with problem size constant, and we form level curves by connecting same performance values on the performance graph.

Smallest interesting problem size is 3 (population size 3) and smallest time is 1 (single generation of SGA). Noting that for a particular performance value, decreasing population size generally results in SGA running more generations and increasing population size generally results in SGA running fewer generations, this suggests approximating level curves using hyperbolic curves that open toward $[\infty, \infty]$ and cornering the origin, figure 6. There is no evidence that suggests time and space tradeoffs are symmetrical, therefore we approximate level curves with

$$s = \left(\frac{c}{t}\right)^p + 1 \quad (7)$$

c is a parameter that roughly describes how “far” a curve is from the origin, and p is parameter that disengages the built in hyperbolic symmetry along the $x = y$ line. Let $q = p \ln c$ and rearrange

$$\begin{aligned}
 s - 1 &= \left(\frac{c}{t}\right)^p \\
 \ln(s - 1) &= p \ln\left(\frac{c}{t}\right) \\
 \ln(s - 1) &= p(\ln c - \ln t) \\
 &= p \ln c - p \ln t \\
 &= q - p \ln t \\
 &= \ln e^q - \ln t^p \\
 \ln(s - 1) &= \ln \frac{e^q}{t^p}
 \end{aligned}$$

and finally

$$s = e^q t^{-p} + 1 \quad (8)$$

that is our model of level curves. Parameters p and q determines a level curve completely. If we can predicate p and q as problem size and performance changes, then we have a model of relationship of time, space, problem size, and performance value for SGA. We name this relation $\Phi(t, s, \ell, \alpha) = 0$.

Let m be slope of a line and b be the line's vertical intercept, we first describe a method of fitting set of points, $(x, y)_i$ to a linear model.

$$y = mx + b \quad (9)$$

We minimize square of the error function

$$\varepsilon^2 = \sum (y - mx - b)^2 \quad (10)$$

Let n be number of points. Take partial derivative with respect to m

$$\begin{aligned}
 \frac{\partial \varepsilon}{\partial m} \sum (y - mx - b)^2 &= -2 \sum (y - mx - b)x \\
 &= -2 \sum (xy - m \sum x^2 - b \sum x)
 \end{aligned}$$

Set result to 0 and isolate m and b

$$\begin{aligned}
 0 &= -2 \sum (xy - m \sum x^2 - b \sum x) \\
 0 &= \sum (xy - m \sum x^2 - b \sum x) \\
 \sum xy &= m \sum x^2 + b \sum x
 \end{aligned} \quad (11)$$

Take partial derivative of equation 10 with respect to b

$$\begin{aligned}
\frac{\partial \epsilon}{\partial b} \sum (y - mx - b)^2 &= -2 \sum (y - mx - b) \\
&= -2(\sum y - \sum mx - \sum b) \\
0 &= -2(\sum y - m \sum x - b \sum 1) \\
0 &= \sum y - m \sum x - bn \\
\sum y &= m \sum x + bn
\end{aligned} \tag{12}$$

Equations 11 and 12 form a matrix equation that we can solve for m and b ,

$$\begin{pmatrix} \sum x^2 & \sum x \\ \sum x & n \end{pmatrix} \begin{pmatrix} m \\ b \end{pmatrix} = \begin{pmatrix} \sum xy \\ \sum y \end{pmatrix}$$

where results for m and b are

$$\begin{aligned}
m &= \frac{\sum x \sum y - n \sum xy}{(\sum x)^2 - n \sum x^2} \\
b &= \frac{\sum x \sum xy - \sum y \sum x^2}{(\sum x)^2 - n \sum x^2}
\end{aligned}$$

We have just derived slope and intercept of a line that minimizes squared of error for $(x, y)_i$. We will use this minimization of squared error method to fit $(t, s)_i$ to the hyperbolic model, equation 7. We take logarithm of both sides of

$$\begin{aligned}
s &= e^q t^{-p} + 1 \\
\ln(s - 1) &= \ln \frac{e^q}{t^p} \\
\ln(s - 1) &= q - p \ln t
\end{aligned}$$

Replace $\ln(s - 1)$ with y and $-\ln t$ with x

$$y = q - px$$

and we apply squared of error minimization to this equation and find hyperbolic curves for the level curves. We could have fit the hyperbolic curve directly to our data using square of error minimization. It is also possible to minimize horizontal squared of error or perpendicular distance to desired curve. We chose a fitting method that seems most convenient to us. We now derive a function of p and q that describes the distance between a level curve and the origin as radius of a circle, figure 7 $r^2 = t^2 + s^2$. This function has an important role in analysis of our

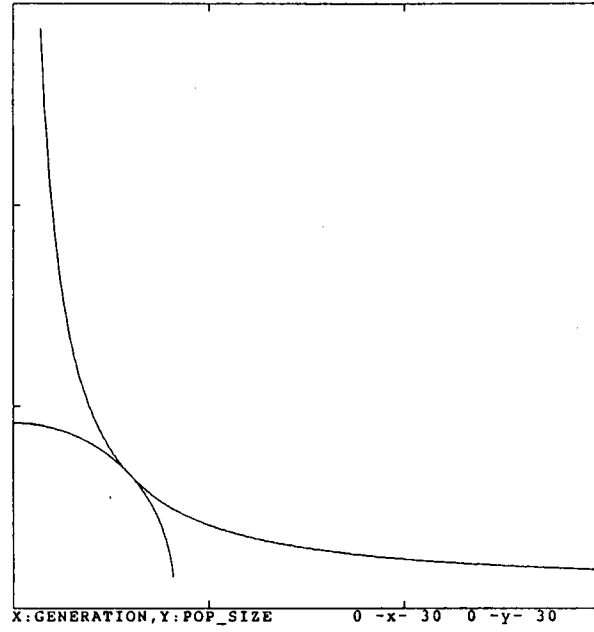


Figure 7: Distance Between Level Curve and Origin

model. We start from description of a level curve and taking its derivative with respect to t

$$s = \frac{e^q}{t^p} \xrightarrow{\frac{d}{dt}} \frac{ds}{dt} = -p \frac{e^q}{t^{p+1}}$$

apply same operation to the circle equation

$$r^2 = t^2 + s^2 \xrightarrow{\frac{d}{dt}} 2r \frac{dr}{dt} = 2t + 2s \frac{ds}{dt}$$

isolate $\frac{dr}{dt}$ and replace $\frac{ds}{dt}$ with $-p \frac{e^q}{t^{p+1}}$

$$\frac{dr}{dt} = \frac{1}{2r} \left(2t + 2s \left(-p \frac{e^q}{t^{p+1}} \right) \right)$$

Instance of t when above equation is 0 is the first coordinate of the point where a level curve meets the smallest circle centered at origin. We note that above equation is 0 when $2t$ is equal to $2s \left(-p \frac{e^q}{t^{p+1}} \right)$. Rearrange the zero condition and solve for t

$$\begin{aligned} 2t &= - \left(2s \left(-p \frac{e^q}{t^{p+1}} \right) \right) \\ 2t &= - \left(2s \left(-p \frac{st^p}{t^{p+1}} \right) \right) \\ t &= sp \frac{st^p}{t^{p+1}} \end{aligned}$$

$$\begin{aligned}
t &= spst^{-1} \\
t^2 &= ps^2 \\
t^2 &= p \frac{e^{2q}}{t^{2p}} \\
t^{2(p+1)} &= pe^{2q} \\
2(p+1) \ln t &= \ln p + \ln e^{2q} \\
t &= e^{\frac{\ln p}{2(p+1)}} e^{\frac{q}{p+1}} \\
t &= e^{\ln p \frac{1}{2p+1}} e^{\frac{q}{p+1}} \\
t &= e^{\frac{q}{p+1}} \sqrt{p^{\frac{1}{p+1}}}
\end{aligned}$$

Substitute t back into the circle equation and find r as a function of p and q

$$\begin{aligned}
r^2 &= t^2 + s^2 \\
r^2 &= t^2 + \frac{t^2}{p} \\
r^2 &= t^2 \left(1 + \frac{1}{p}\right) \\
r &= t \sqrt{1 + \frac{1}{p}}
\end{aligned}$$

and finally

$$r = e^{\frac{q}{p+1}} \sqrt{p^{\frac{1}{p+1}} \left(1 + \frac{1}{p}\right)}$$

Note that quantity in square root bracket goes to 1 when either $p \rightarrow 0$ or $p \rightarrow \infty$, thus r is essentially the ratio $\frac{p}{q+1}$. From experimental data, figure 8, we see that radius increases as string length increases, and we expect radius to increase as string length increases. It is tempting to model both p and q as linear function of ℓ , since this model immediately gives us prediction of p and q as length increases. However, if we model both p and q as linear function of string length,

$$p = a\ell + b$$

$$q = c\ell + d + 1$$

we note that radius

$$\begin{aligned}
\lim_{\ell \rightarrow \infty} r &= \lim_{\ell \rightarrow \infty} \left(e^{\frac{a\ell+b}{c\ell+d+1}} \right) \sqrt{c\ell + d^{\frac{1}{c\ell+d+1}} \left(1 + \frac{1}{c\ell + d + 1}\right)} \\
&= e^{\frac{a}{c}}
\end{aligned}$$

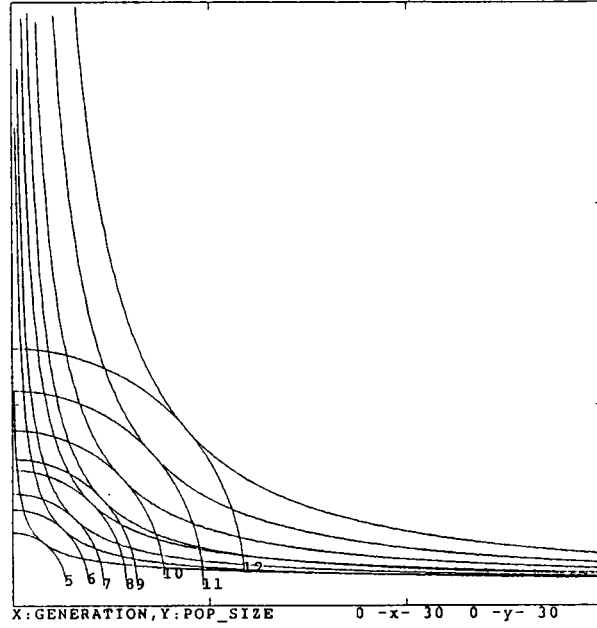


Figure 8: Radius Increases As String Increases

converges to a constant as string length increases. If this is true, then there would exist finite time and space resources such that any arbitrary problem could be solved by a SGA to a performance level α . Distance of level curves from the origin is an useful piece of information for guiding model development. We search for simple linear relationships for $\Phi(t, s, \alpha, \ell) = 0$ and found four that could fill the requirement of describing $\Phi()$. They are

$$\ell \text{ vs. } q$$

$$\alpha \text{ vs. } q$$

$$\ell \text{ vs. } \ln r$$

$$\alpha \text{ vs. } \ln r$$

We combine the above four relations in two equations,

$$q = \ell(a\alpha + b) + c\alpha + d \quad (13)$$

$$\ln r = \ell(e\alpha + f) + g\alpha + h \quad (14)$$

The problem now becomes find parameters (a, b, c, d, e, f, g) that minimize error for our level curve equation

$$s - 1 = e^q t^{-p}$$

using squared error function

$$\epsilon^2 = \sum_{\ell} \sum_{\alpha} \sum_{t,s} (\ln s - 1 - q + p \ln t)^2$$

After using non-linear optimization code[7] to minimize ϵ^2 , we have

$$\begin{aligned} a &= 0.006203717963557785 \\ b &= 0.05741445188108189 \\ c &= 0.01035219159712265 \\ d &= -2.316011897765724 \\ e &= 0.0005950768507815831 \\ f &= 0.1500011040546082 \\ g &= 0.02155919549315095 \\ h &= -1.59999968328718 \end{aligned}$$

Using parameters a through h , equations 13, equation 14, and

$$r = e^{\frac{q}{p+1}} \sqrt{p^{\frac{1}{p+1}} \left(1 + \frac{1}{p}\right)}$$

we can numerically solve for p and q , which defines a level curve, given ℓ and α . We show experimental level curves for string length 5 through 12, figures 9 through 16. and level curve predictions for string length 13 figure 17 using the model we just defined. The model's prediction is very close to the actual data.

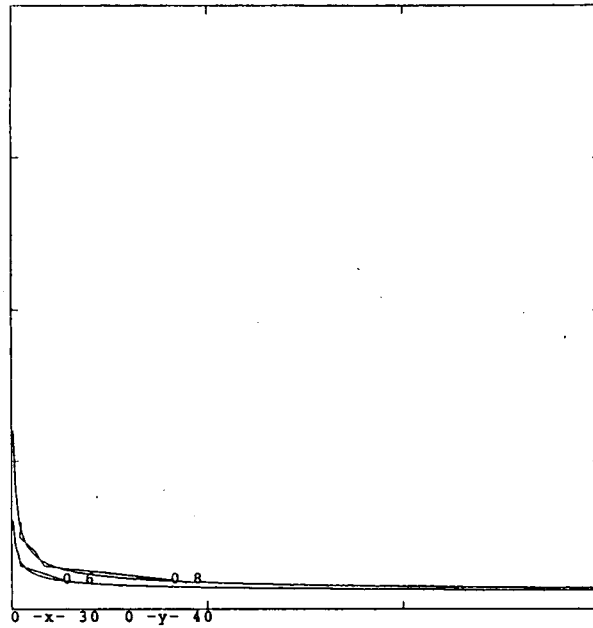


Figure 9: String Length 5 Performance Curves

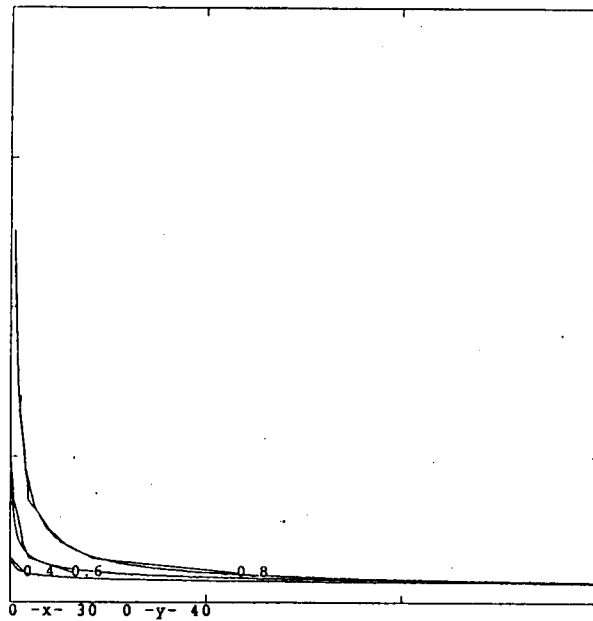


Figure 10: String Length 6 Performance Curves

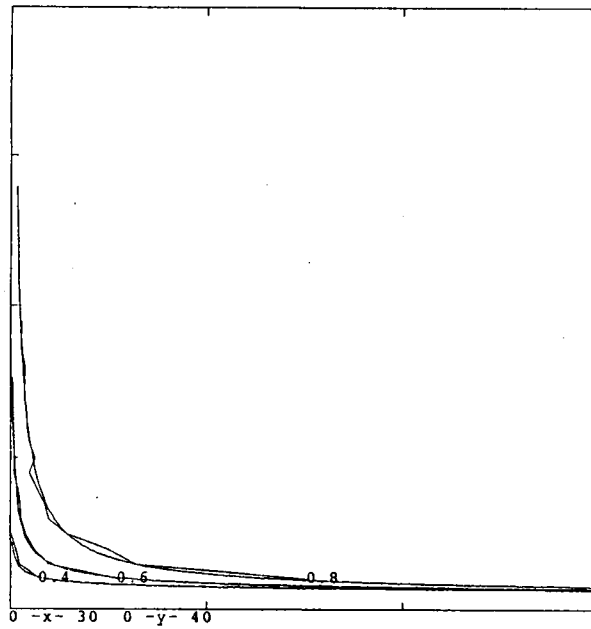


Figure 11: String Length 7 Performance Curves

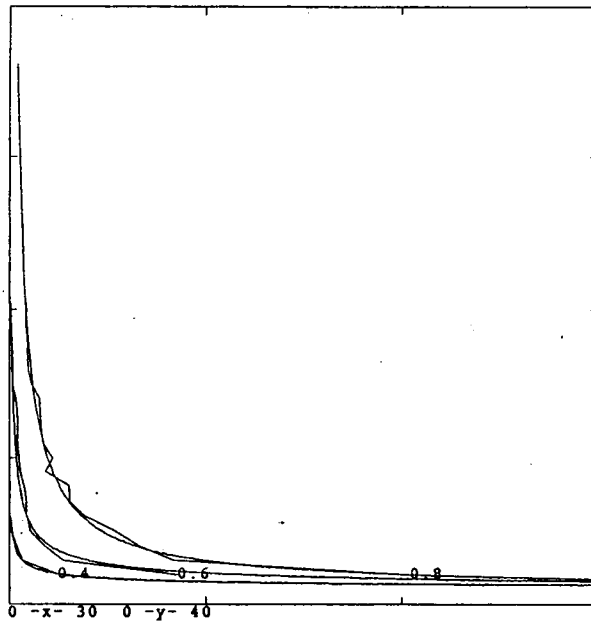


Figure 12: String Length 8 Performance Curves

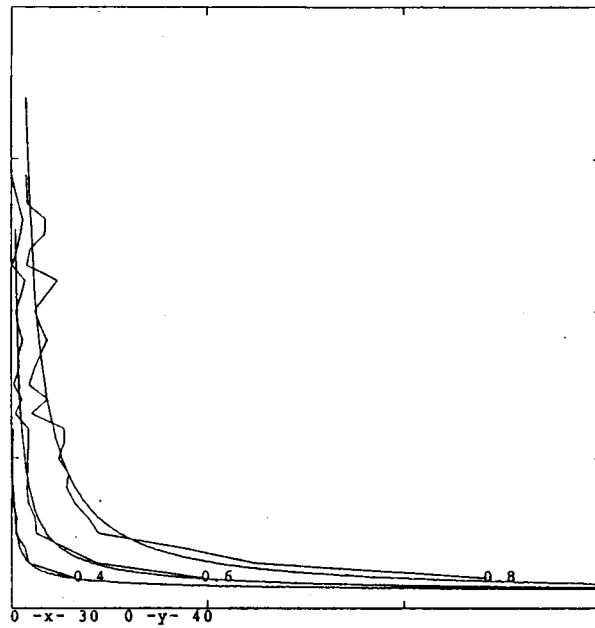


Figure 13: String Length 9 Performance Curves

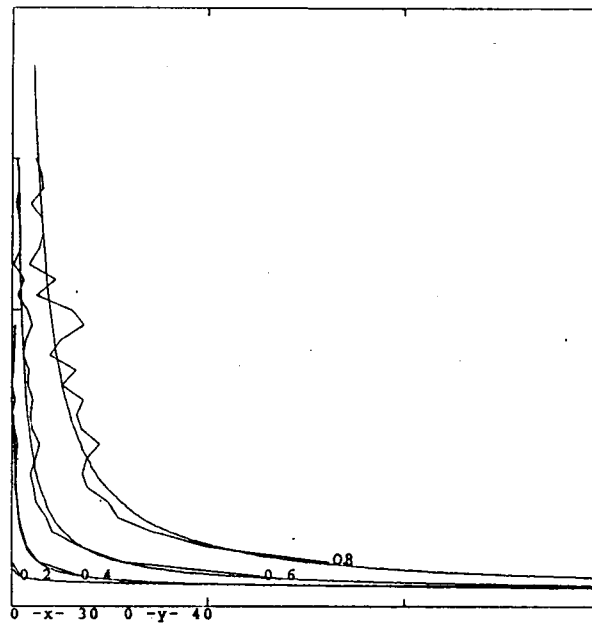


Figure 14: String Length 10 Performance Curves

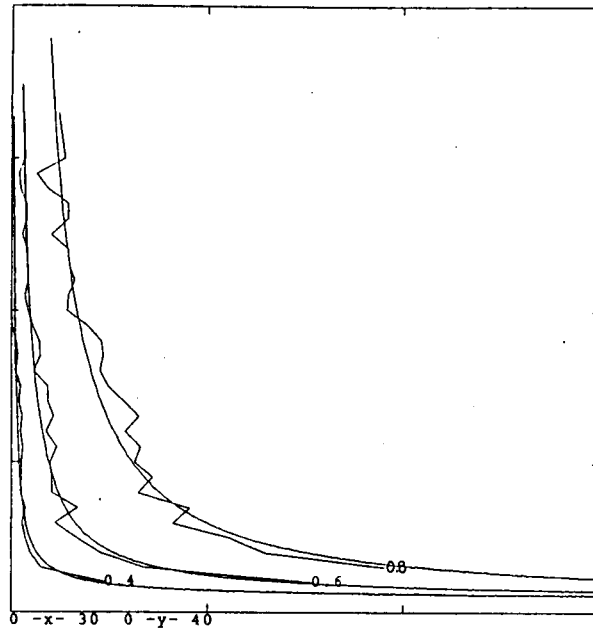


Figure 15: String Length 11 Performance Curves

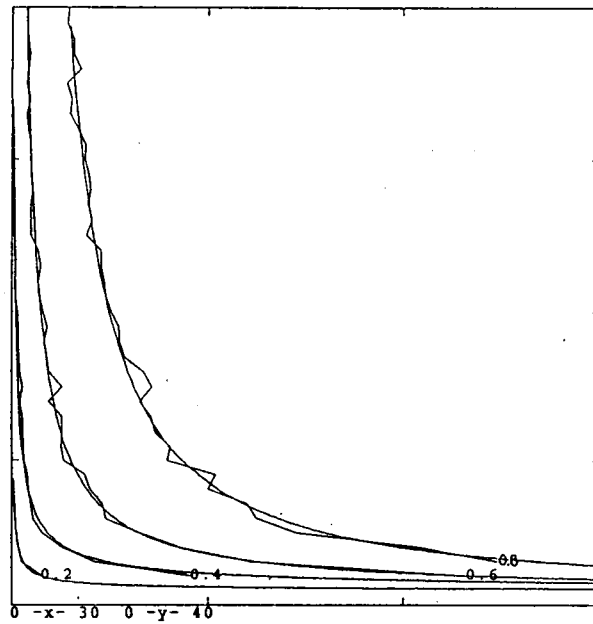


Figure 16: String Length 12 Performance Curves

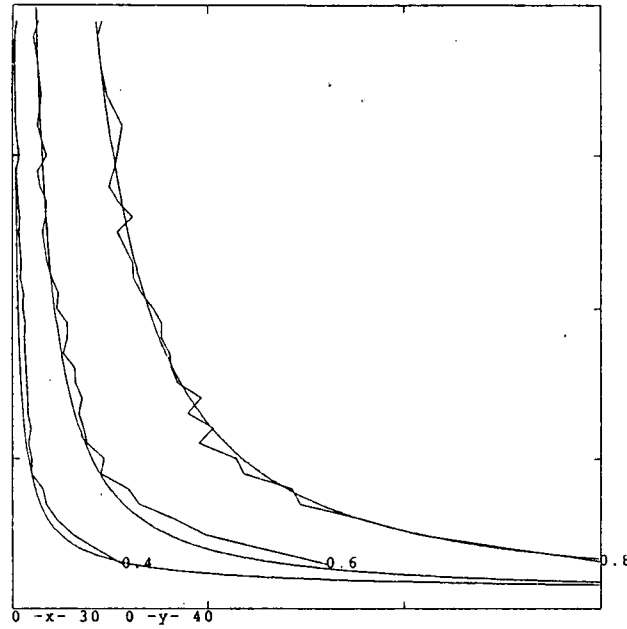


Figure 17: Predictions and Actual Data for $\ell = 13$ Performance Curves

Each point on level curve is average of SGA performance on 198 polynomials and we form a level curve by connecting points where mean performance level is constant. The shapes of the distribution of the performance values for the SGA using these 198 polynomials is fairly constant along level curve. We use standard deviation (second moment about mean) of the distributions along a level curve for a more objective analysis of these distributions. We also use kurtosis (third moment with respect to the mean) to measure characteristics of the distributions along a level curve. We first show standard deviation (y axis) versus population size (x axis) for string length 5 through 12 and performance values 0.4, 0.6, and 0.8 figures 18 through 29

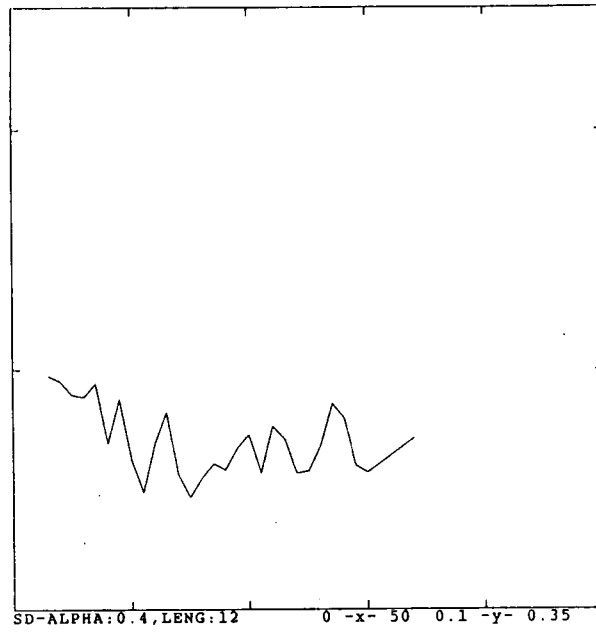


Figure 18: Standard Deviation for $\ell = 12, \alpha = 0.4$

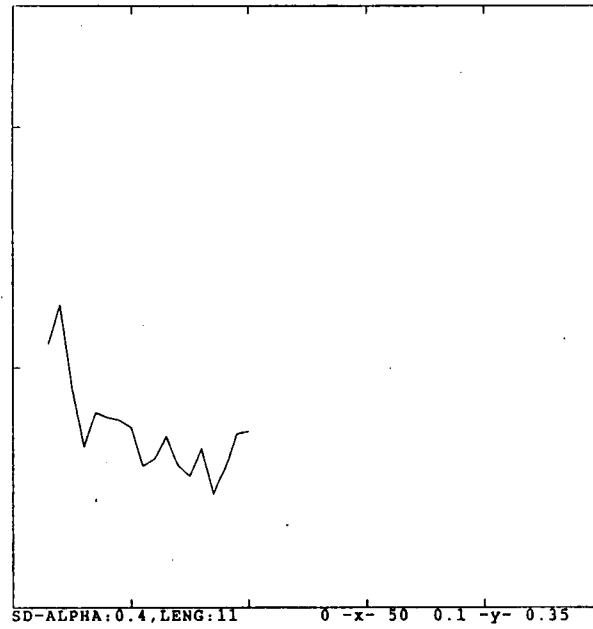


Figure 19: Standard Deviation for $\ell = 11, \alpha = 0.4$

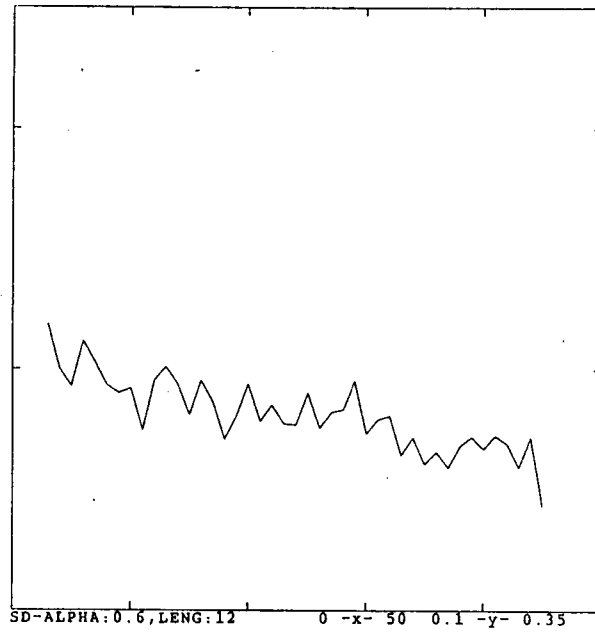


Figure 22: Standard Deviation for $\ell = 12, \alpha = 0.6$

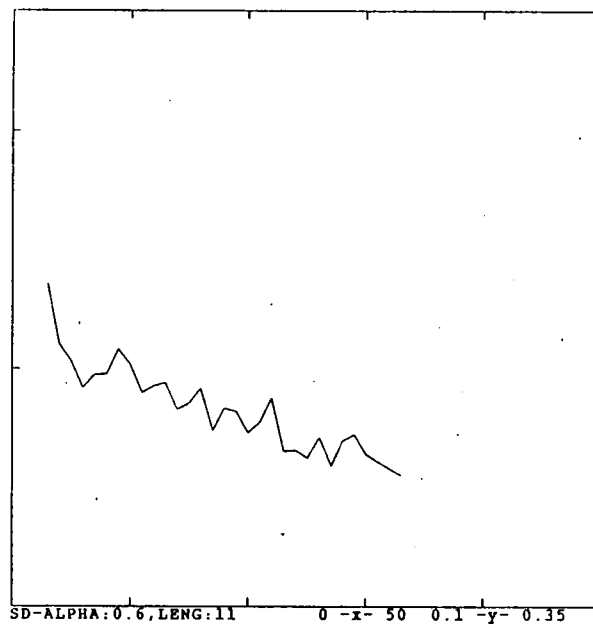


Figure 23: Standard Deviation for $\ell = 11, \alpha = 0.6$

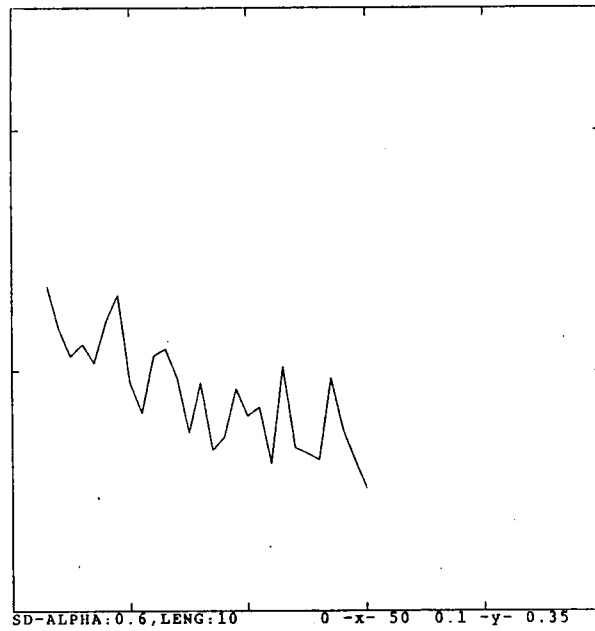


Figure 24: Standard Deviation for $\ell = 10, \alpha = 0.6$

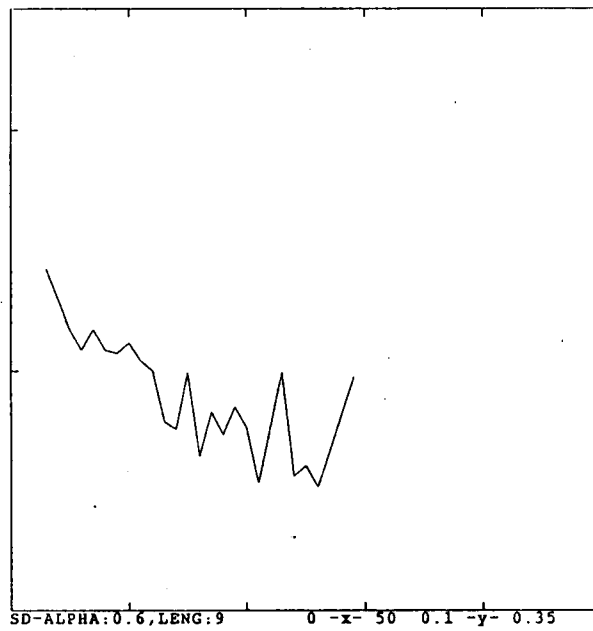


Figure 25: Standard Deviation for $\ell = 9, \alpha = 0.6$

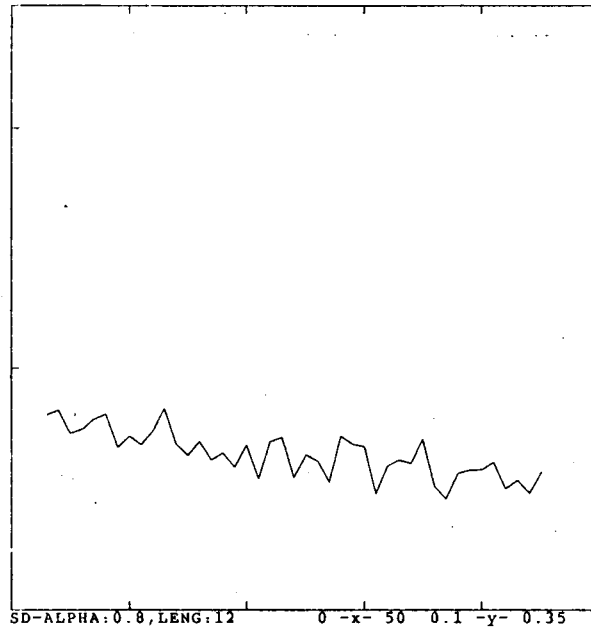


Figure 26: Standard Deviation for $\ell = 12, \alpha = 0.8$

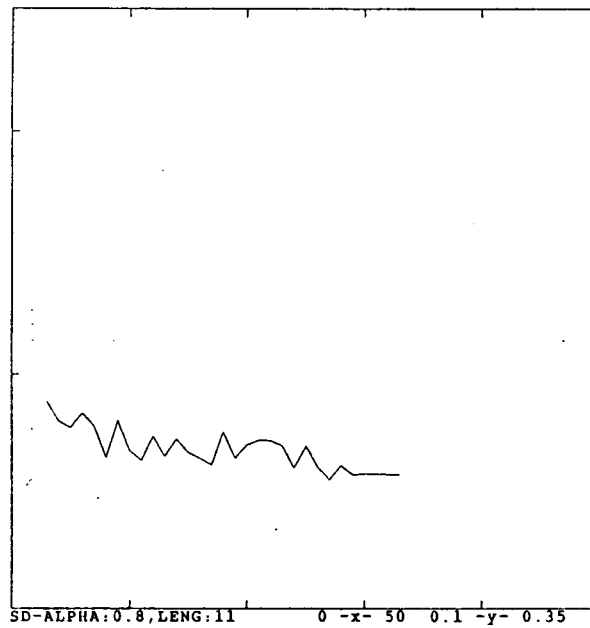


Figure 27: Standard Deviation for $\ell = 11, \alpha = 0.8$

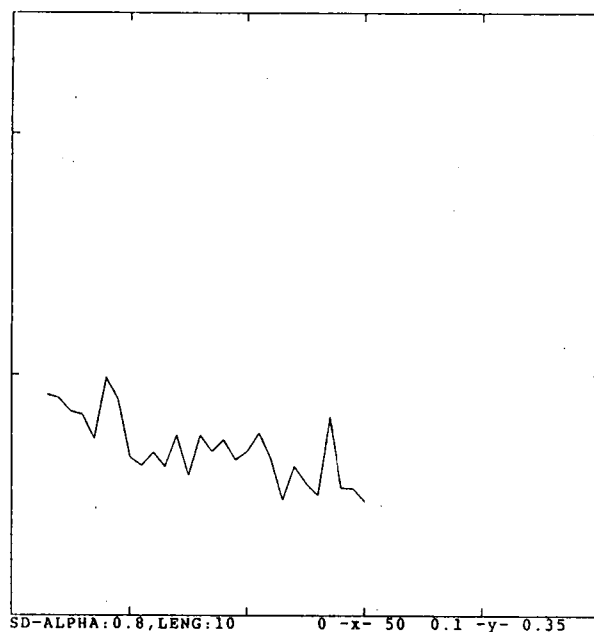


Figure 28: Standard Deviation for $\ell = 10, \alpha = 0.8$

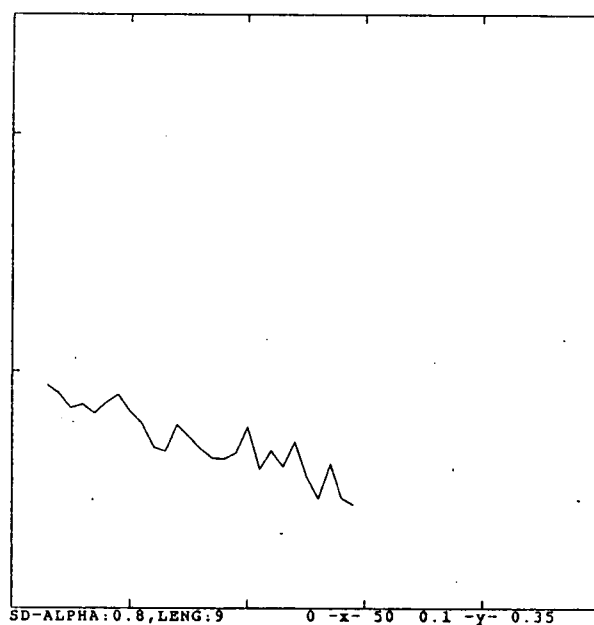


Figure 29: Standard Deviation for $\ell = 9, \alpha = 0.8$

Standard deviations versus population size do not vary greatly as population size increases. We do notice a slight decreasing standard deviation trend as population size increases, but downward trend becomes slower as string length

increases. We now graph the kurtosis of problem distribution along level curves figures 30 through 41. Kurtosis will give us some indication of more general behavior of the distributions. We graph kurtosis (y axis) versus population size (x axis).

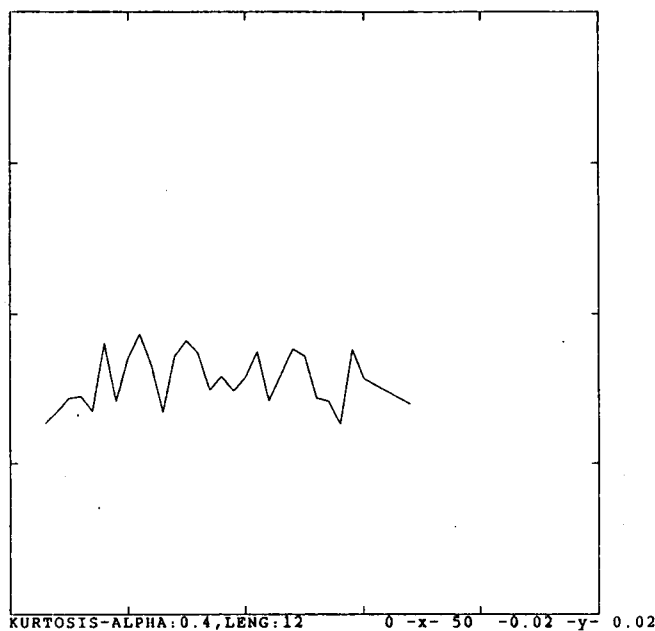


Figure 30: Kurtosis for $\ell = 12, \alpha = 0.4$

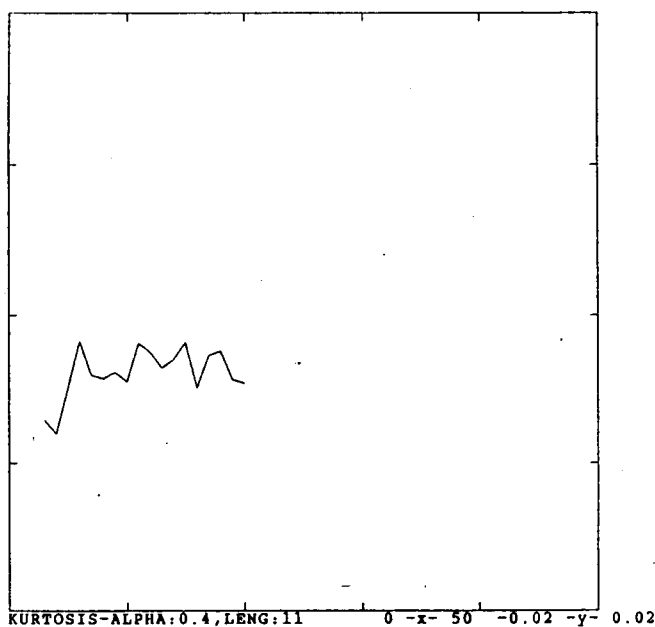


Figure 31: Kurtosis for $\ell = 11, \alpha = 0.4$

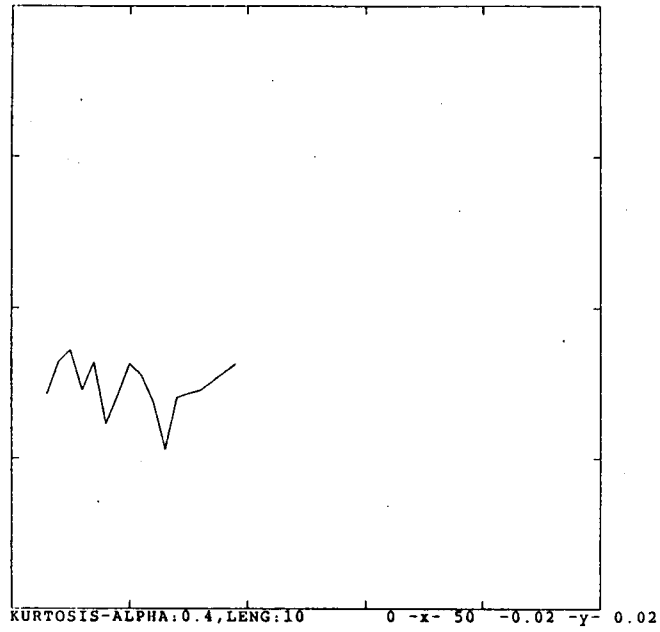


Figure 32: Kurtosis for $\ell = 10, \alpha = 0.4$

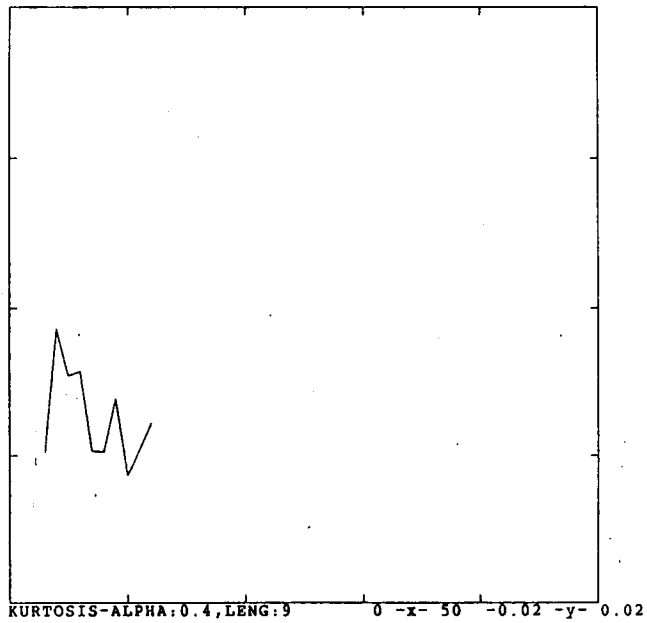


Figure 33: Kurtosis for $\ell = 9, \alpha = 0.4$

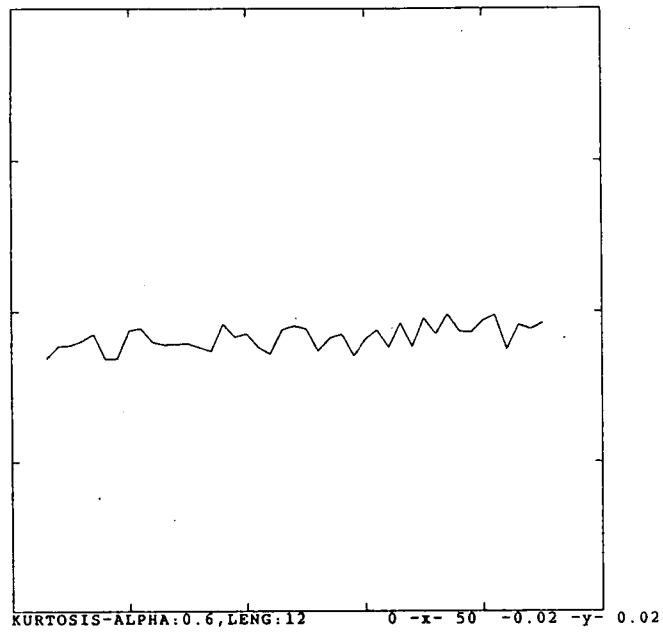


Figure 34: Kurtosis for $\ell = 12, \alpha = 0.6$

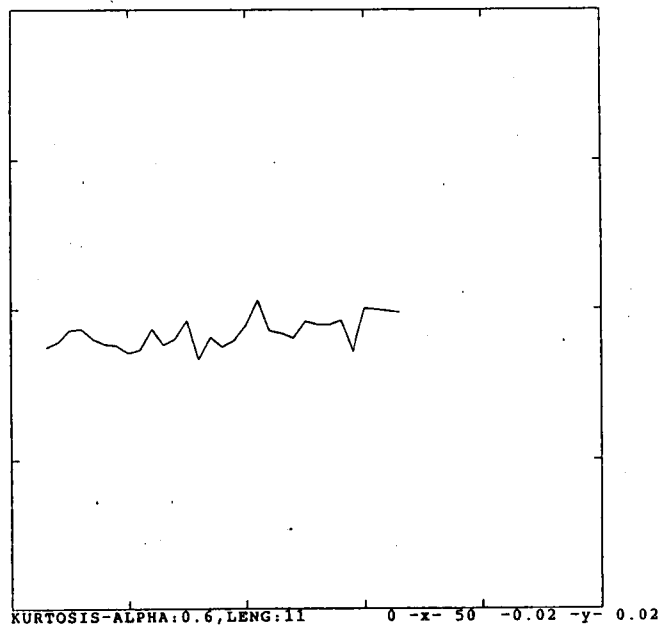


Figure 35: Kurtosis for $\ell = 11, \alpha = 0.6$

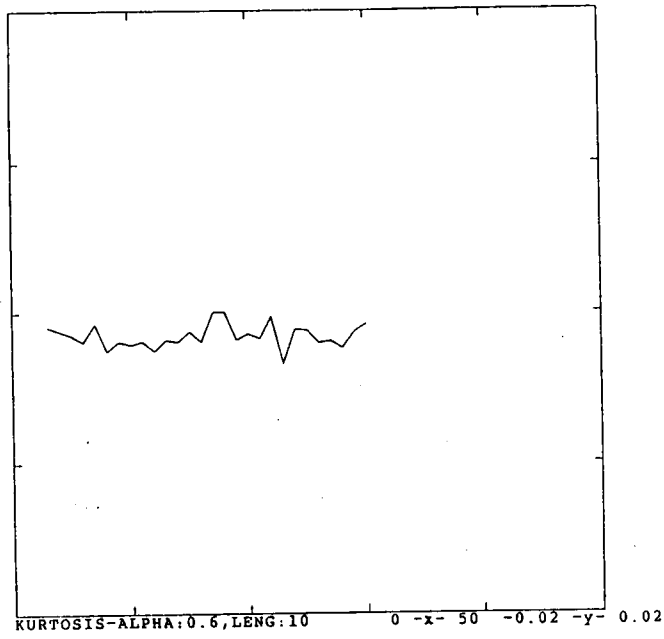


Figure 36: Kurtosis for $\ell = 10, \alpha = 0.6$

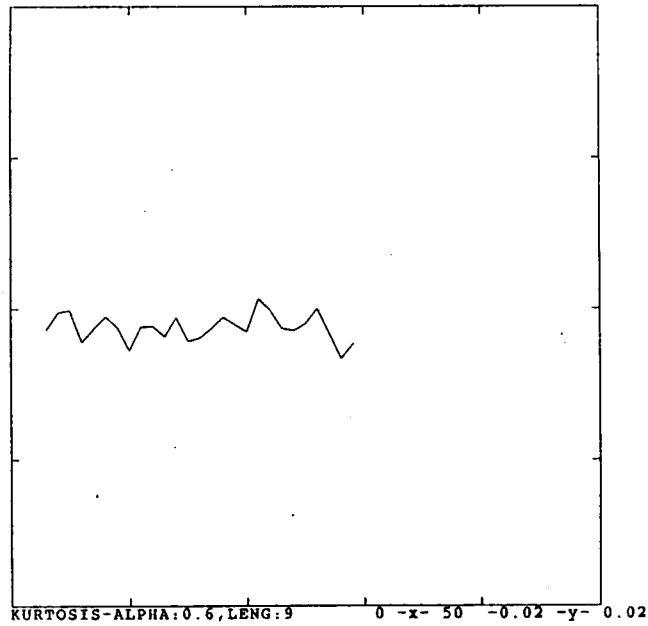


Figure 37: Kurtosis for $\ell = 9, \alpha = 0.6$

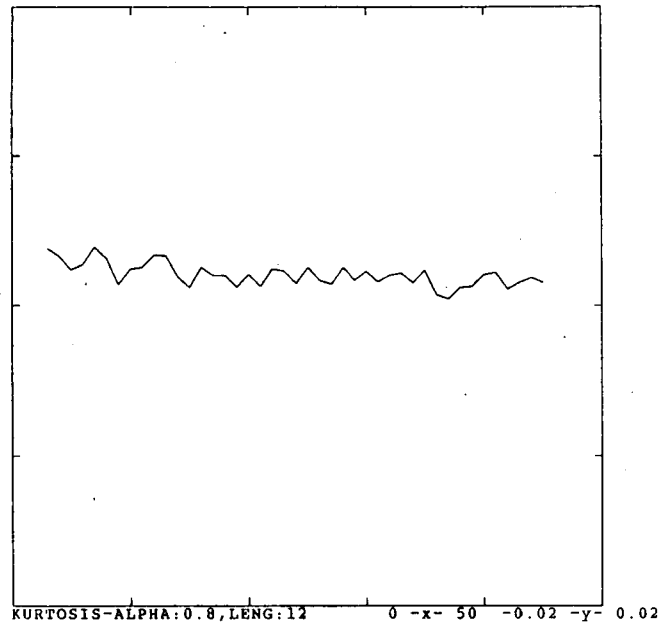


Figure 38: Kurtosis for $\ell = 12, \alpha = 0.8$

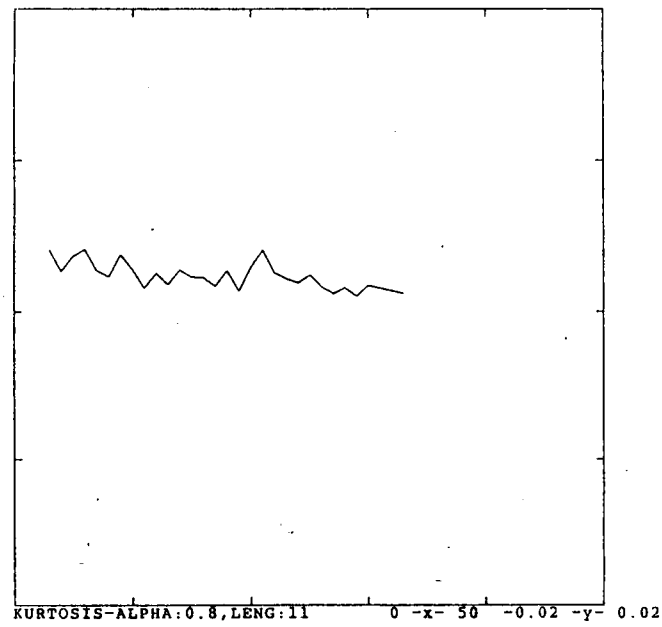


Figure 39: Kurtosis for $\ell = 11, \alpha = 0.8$

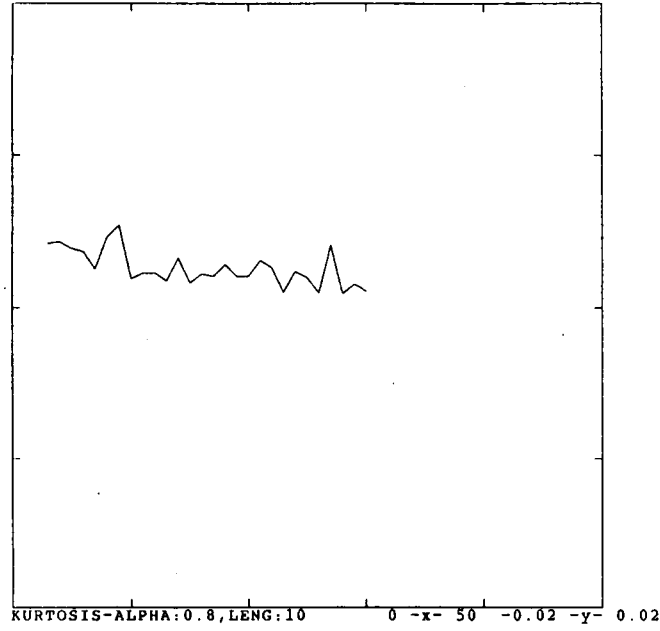


Figure 40: Kurtosis for $\ell = 10, \alpha = 0.8$

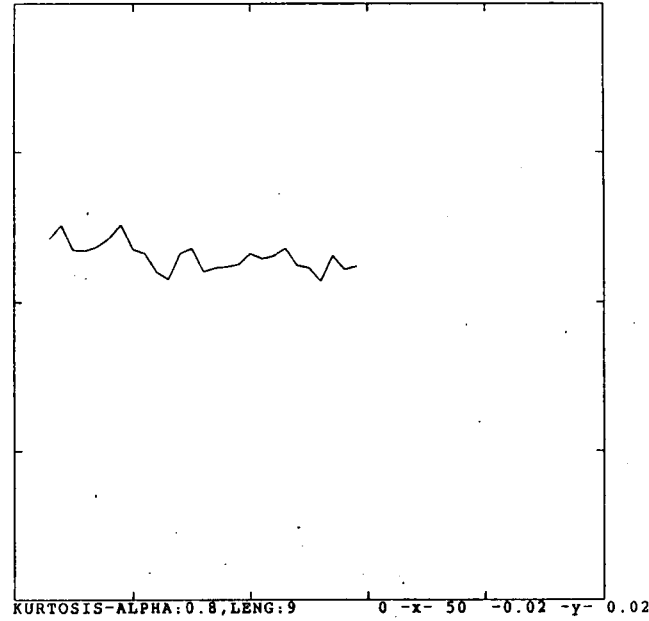


Figure 41: Kurtosis for $\ell = 9, \alpha = 0.8$

The kurtosis for distribution of hard and easy problems along level curves as population size increases are essentially constant except for local noises. This evidence in addition to behavior of standard deviation of problem distribution along

level curves support the conjecture that distributions of randomly generated polynomials that have constant mean is empirically constant. We sum the distributions along each level curve to characterize distribution of easy and hard problems for each performance value. We graph normalized problem distributions (probability densities) for performance values 0.4, 0.6, and 0.8 for string length 5 through 12, figures 42 through 64. We now analyze behavior of these densities as string length changes. There is a visual trend of decreasing deviation as string length increases and densities form general bell shaped curves. We graph a fitted normal curve on top of each density to get a general approximation of the densities.

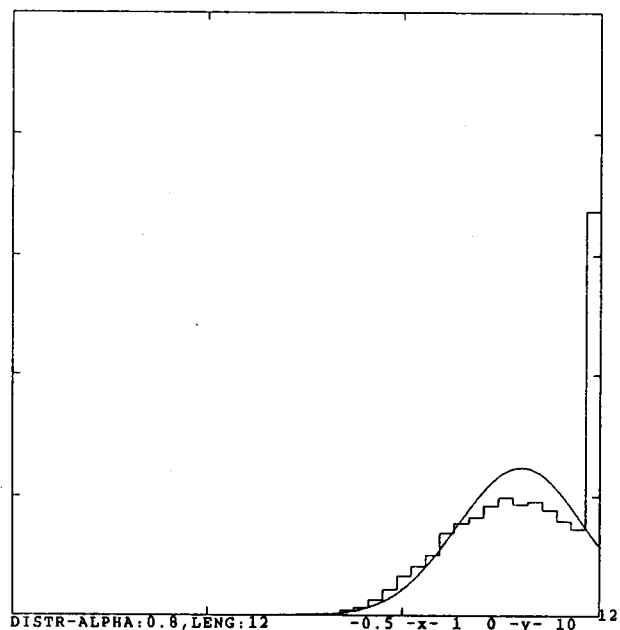


Figure 42: Problem Distribution for $\ell = 12, \alpha = 0.8$

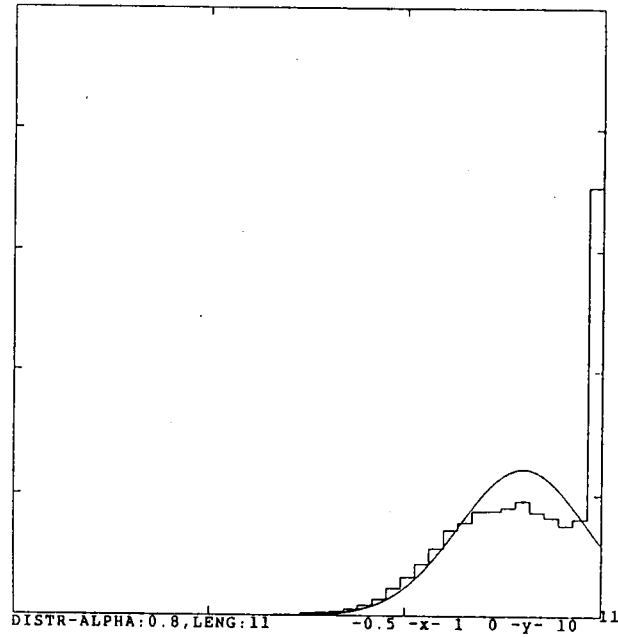


Figure 43: Problem Distribution for $\ell = 11, \alpha = 0.8$

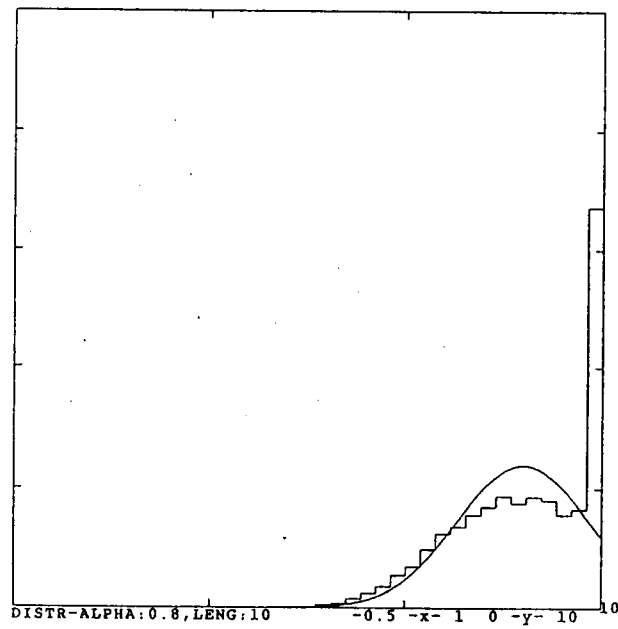


Figure 44: Problem Distribution for $\ell = 10, \alpha = 0.8$

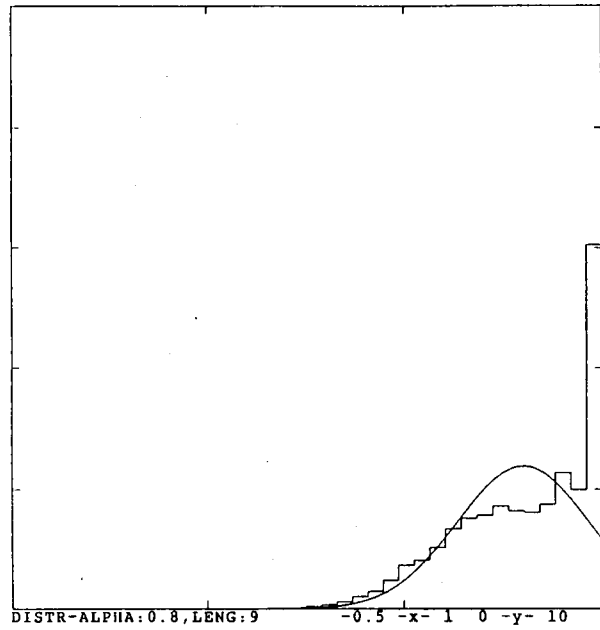


Figure 45: Problem Distribution for $\ell = 9, \alpha = 0.8$

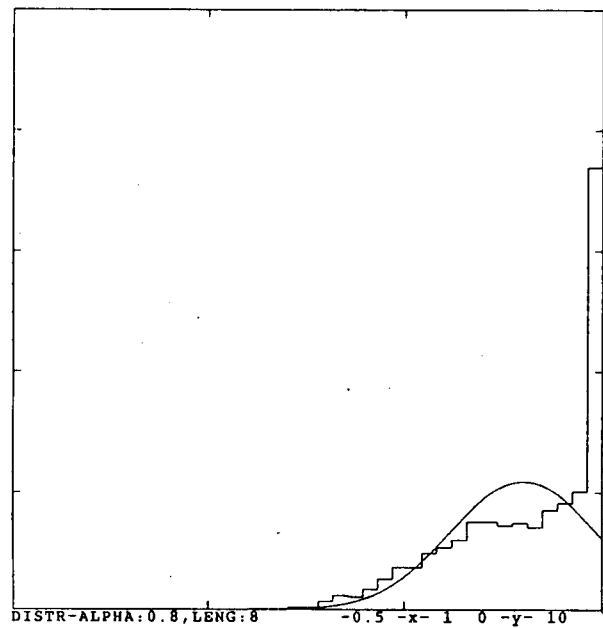


Figure 46: Problem Distribution for $\ell = 8, \alpha = 0.8$

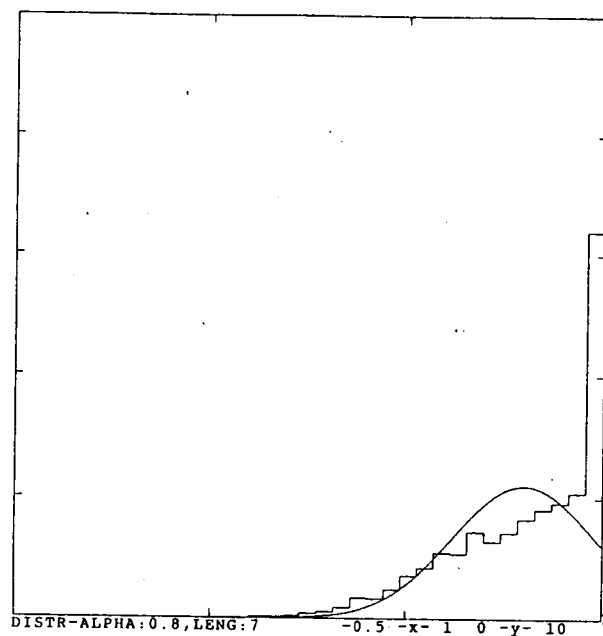


Figure 47: Problem Distribution for $\ell = 7, \alpha = 0.8$

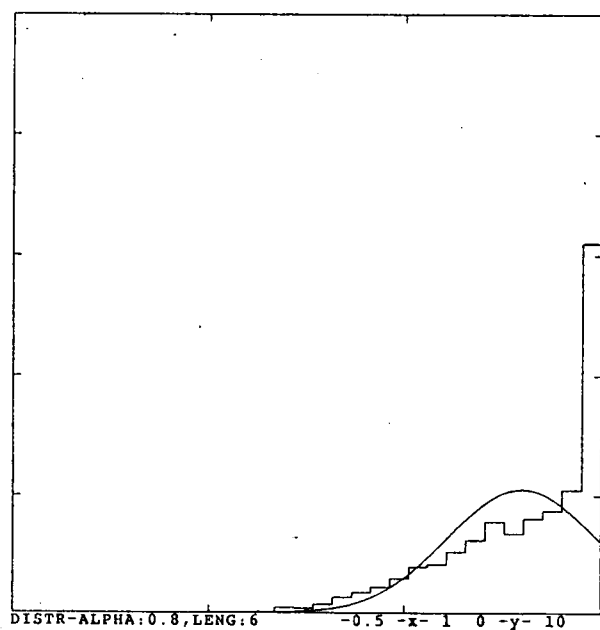


Figure 48: Problem Distribution for $\ell = 6, \alpha = 0.8$

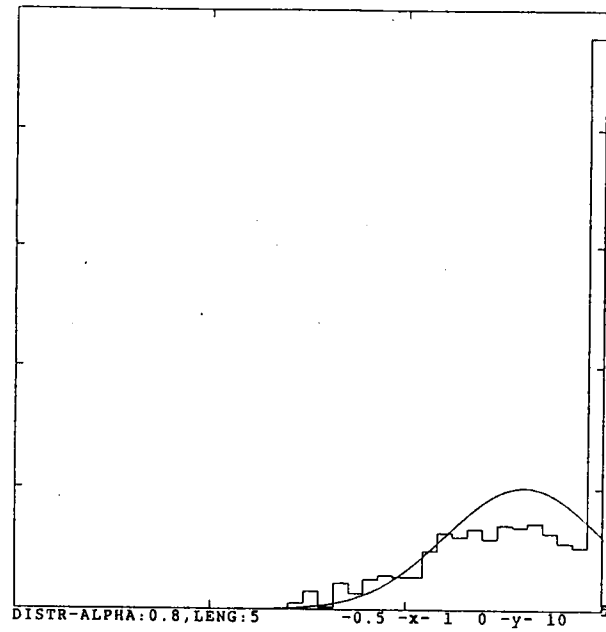


Figure 49: Problem Distribution for $\ell = 5, \alpha = 0.8$

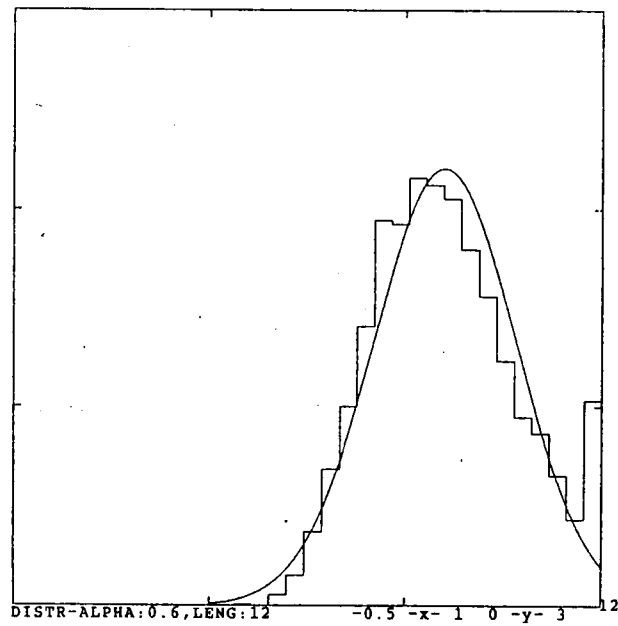


Figure 50: Problem Distribution for $\ell = 12, \alpha = 0.6$

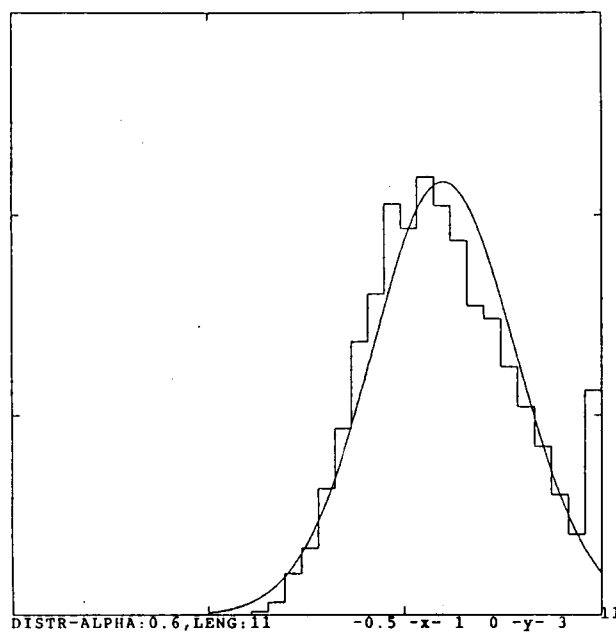


Figure 51: Problem Distribution for $\ell = 11, \alpha = 0.6$

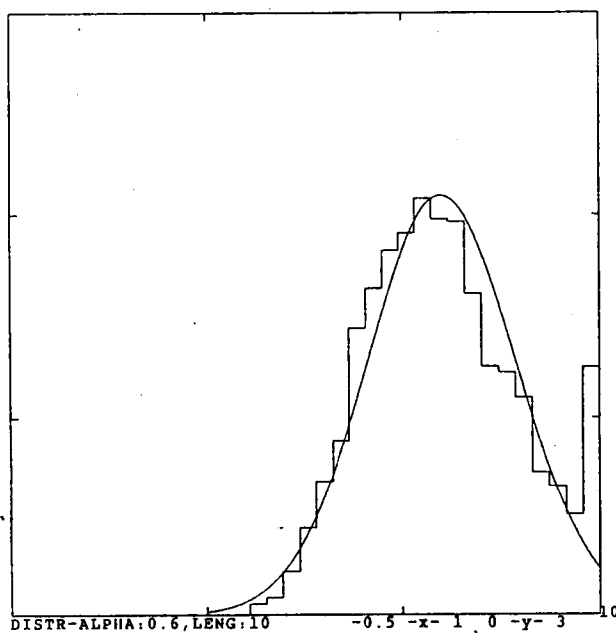


Figure 52: Problem Distribution for $\ell = 10, \alpha = 0.6$

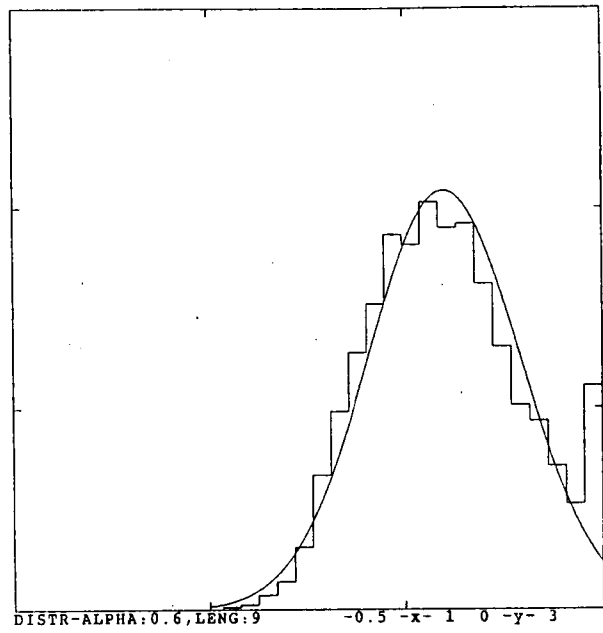


Figure 53: Problem Distribution for $\ell = 9, \alpha = 0.6$

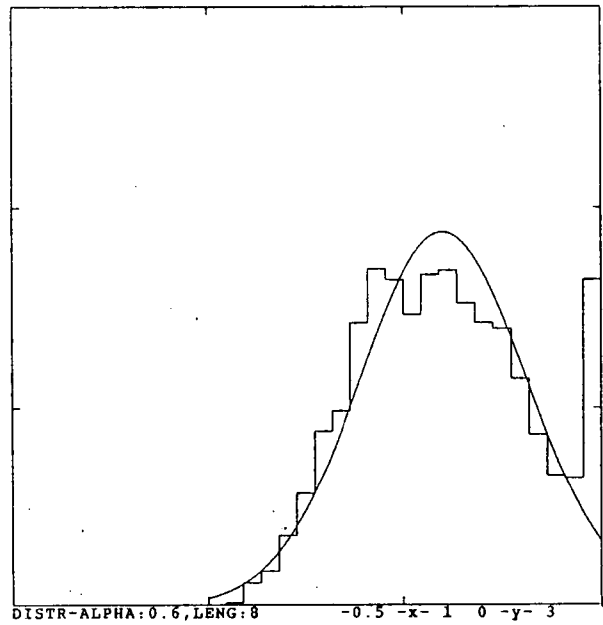


Figure 54: Problem Distribution for $\ell = 8, \alpha = 0.6$

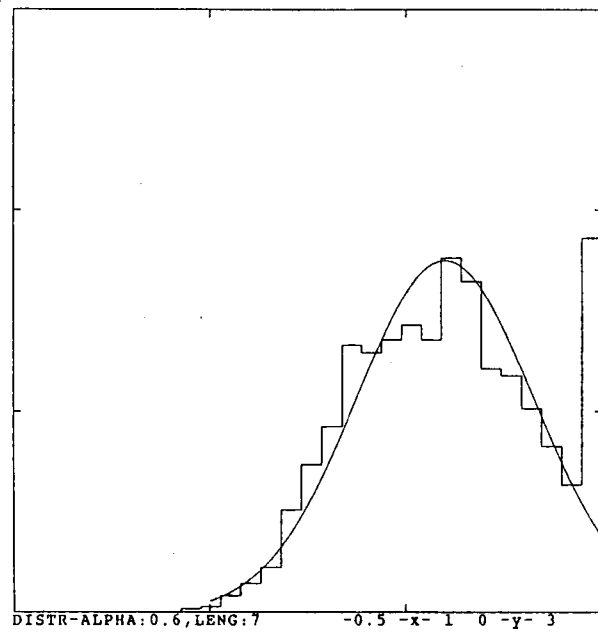


Figure 55: Problem Distribution for $\ell = 7, \alpha = 0.6$

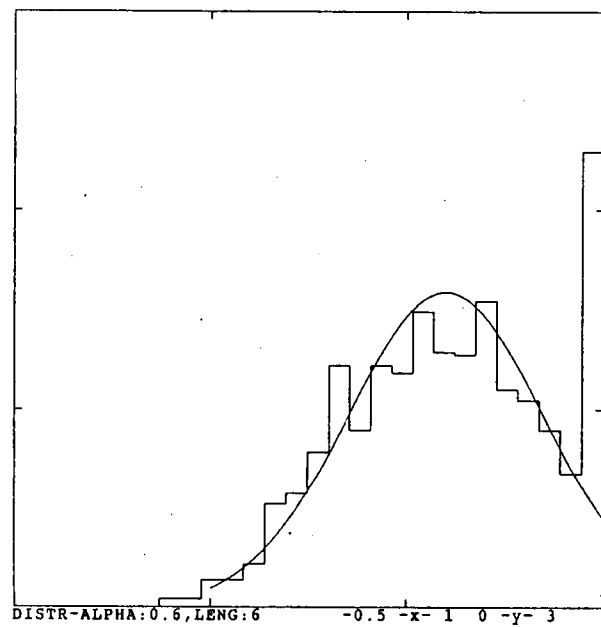


Figure 56: Problem Distribution for $\ell = 6, \alpha = 0.6$

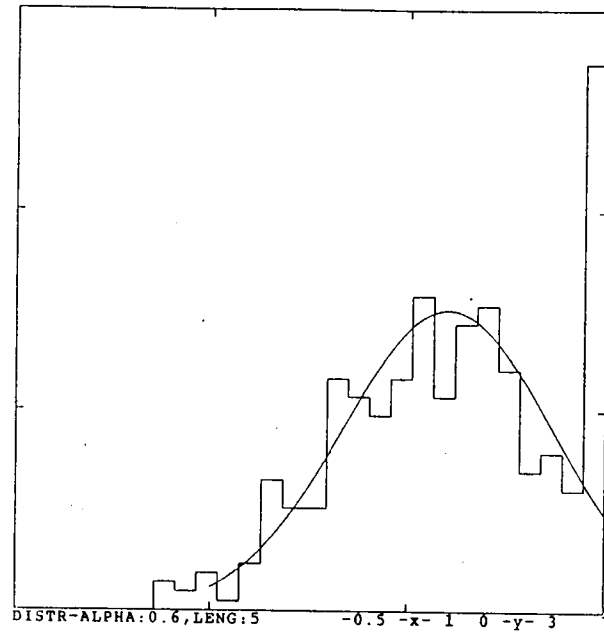


Figure 57: Problem Distribution for $\ell = 5, \alpha = 0.6$

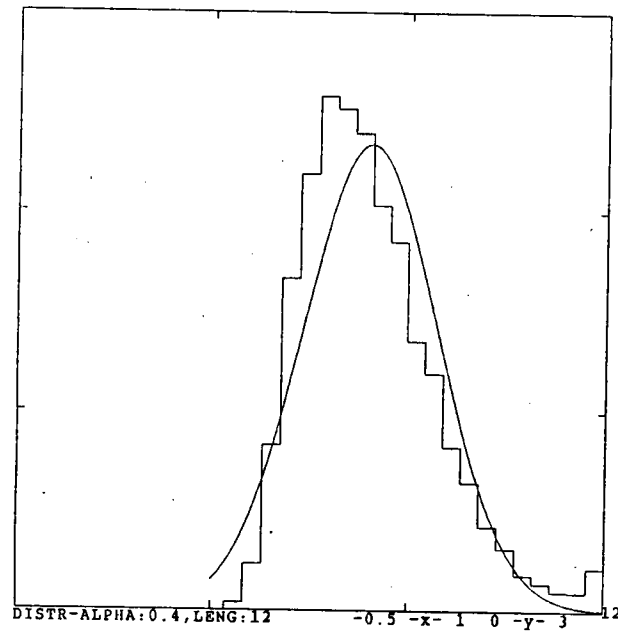


Figure 58: Problem Distribution for $\ell = 12, \alpha = 0.4$

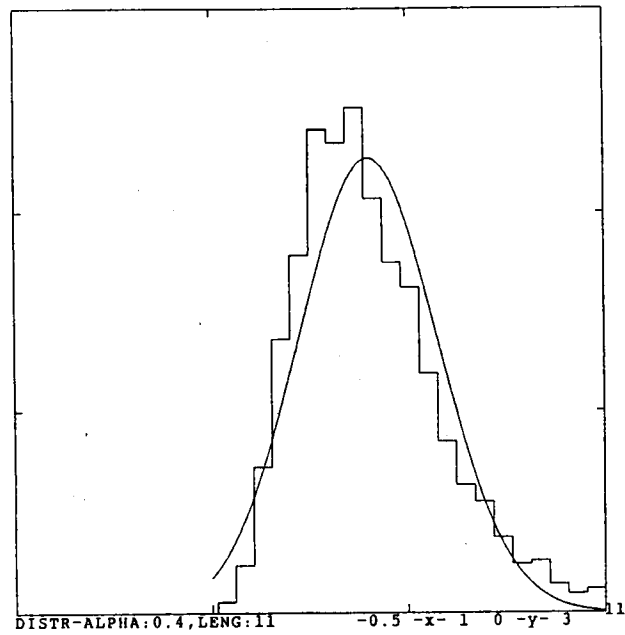


Figure 59: Problem Distribution for $\ell = 11, \alpha = 0.4$

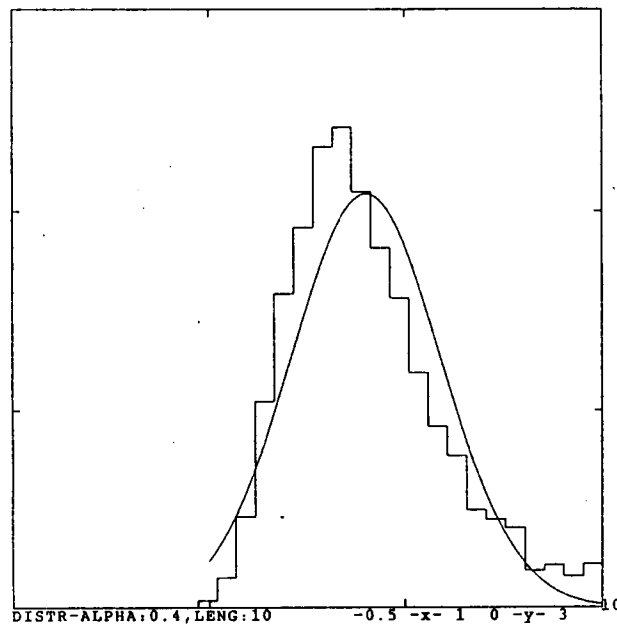


Figure 60: Problem Distribution for $\ell = 10, \alpha = 0.4$

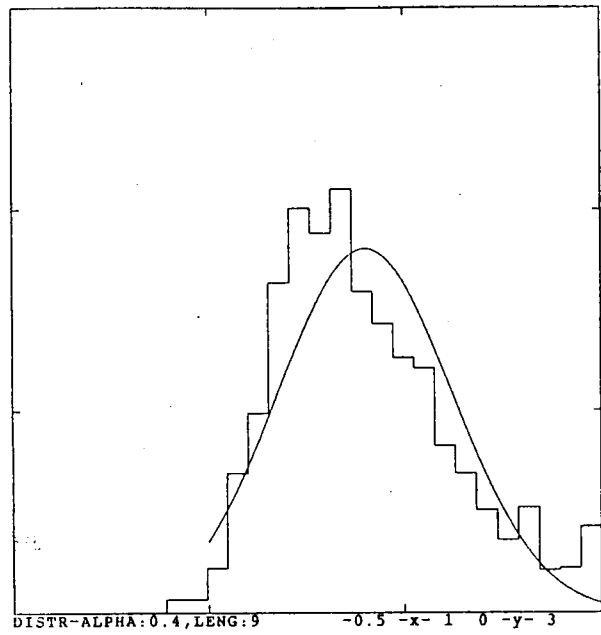


Figure 61: Problem Distribution for $\ell = 9, \alpha = 0.4$

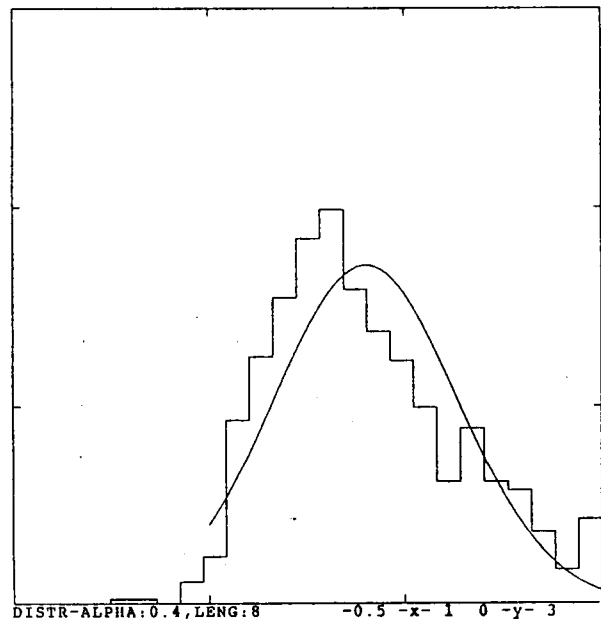


Figure 62: Problem Distribution for $\ell = 8, \alpha = 0.4$

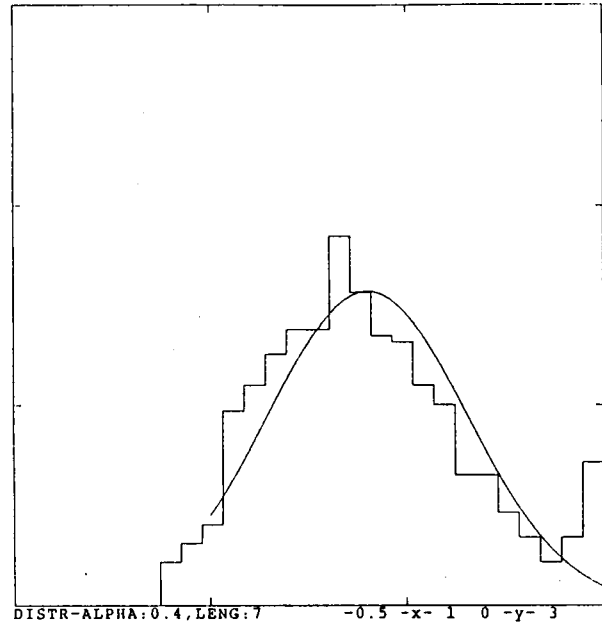


Figure 63: Problem Distribution for $\ell = 7, \alpha = 0.4$

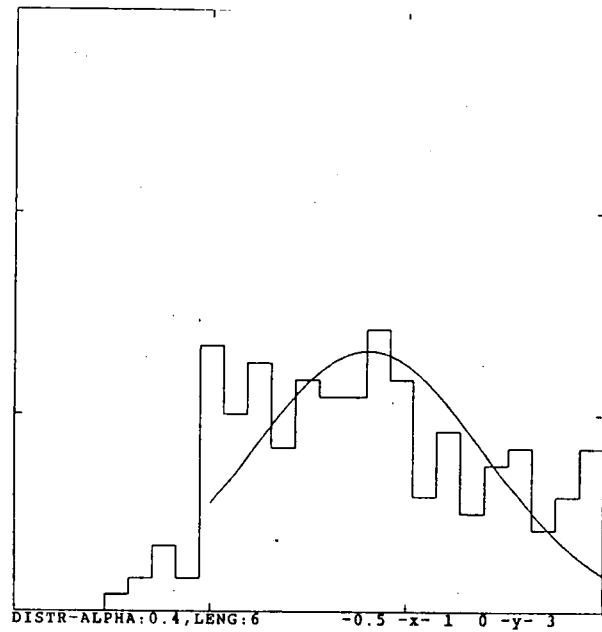


Figure 64: Problem Distribution for $\ell = 6, \alpha = 0.4$

Density mass seems to move towards the mean as string length increases for a constant performance value. We attempt to verify this trend by graph string length (x axis) versus standard deviation (y axis) of these densities for performance

levels 0.4, 0.6, and 0.8, figure 65. Note that there is a general decreasing standard deviation as string increases for problem distribution associated with a performance value.

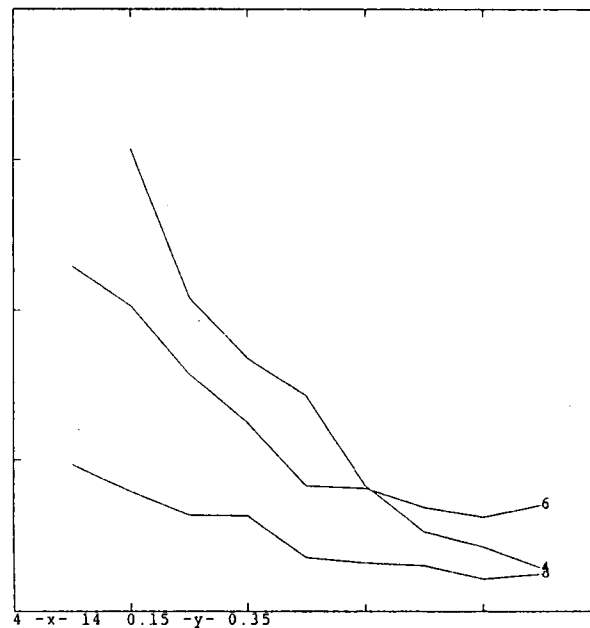


Figure 65: Standard Deviation Decreases As ℓ Increases.

We do not put high confidence on estimating problem distribution using this particular property of decreasing standard deviation because trends for performance level 0.4 and performance level 0.6 actually crosses in this graph, and there is no reason to suspect the standard deviation trends should cross as string length increases. We graph inverse of standard deviation versus string length and predicts standard deviation estimates for string length 13 problem distributions, figures 66 through 68. With consideration to experimental noise, the normal distribution prediction is close to the normal distribution curve from the actual data.

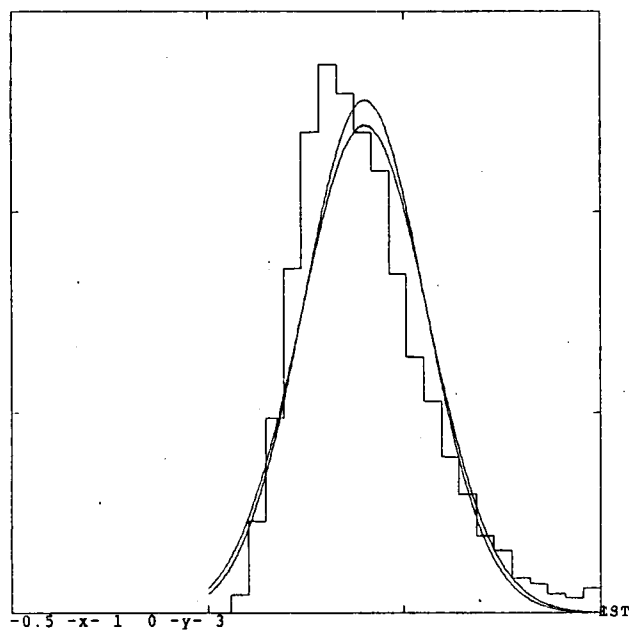


Figure 66: Problem Distribution for $\ell = 13, \alpha = 0.4$

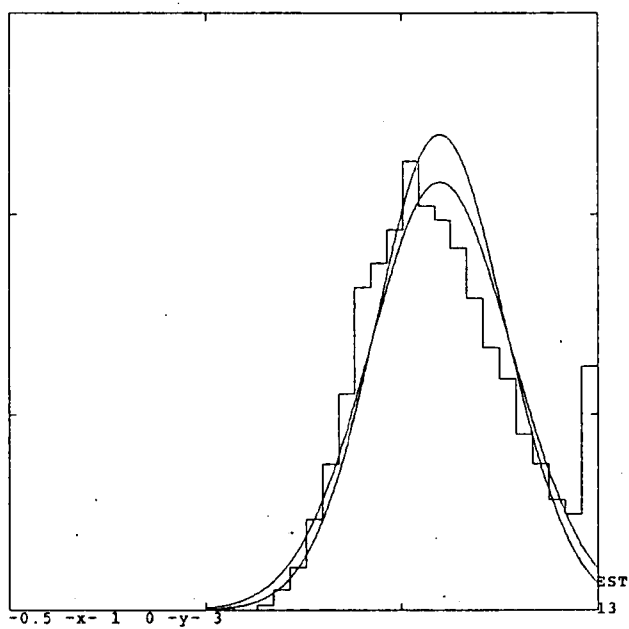


Figure 67: Problem Distribution for $\ell = 13, \alpha = 0.6$

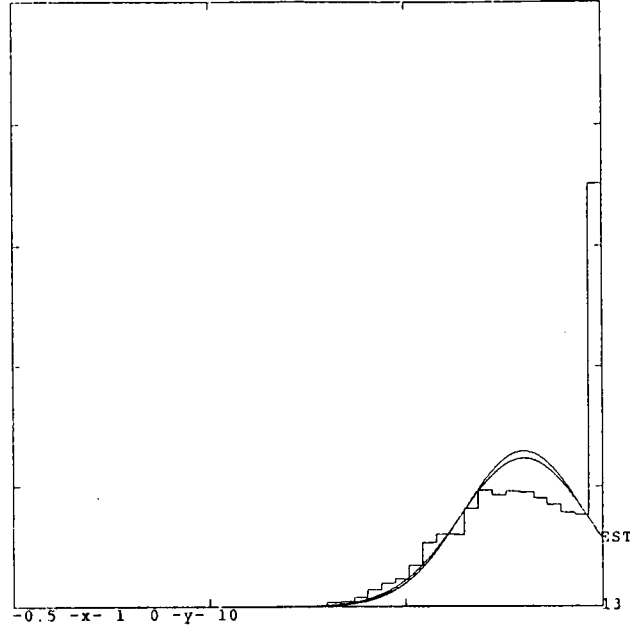


Figure 68: Problem Distribution for $\ell = 13, \alpha = 0.8$

4.3 Conclusion

With a model of time, space, performance, and problem size of SGA, $\Phi(t, s, \alpha, \ell) = 0$. We are able to answer questions such as “Given string length, population size and number of generations allowed for a SGA, What is the probability that the SGA will performance better than β (a performance value) ?” We first calculate the performance value α that corresponds to the given t , s , and ℓ using model $\Phi()$. Each α has a characteristic probability density $\mathcal{D}$ derived from problem distributions along the α level curve. The probability that the particular SGA may perform better than β is simply

$$1 - \int_{-\infty}^{\beta} \mathcal{D}$$

We have used second and third moments about mean which indicate that the problem distribution along a level curve is constant as problem sizw ℓ increases, and this characteristic made answering the above question possible. Constant problem distribution along a level curve means that time and space is an even trade for SGA, which is interesting. We have shown that our model, $\Phi()$, predicts fairly well for string length 13 level curves, and we are able to predict trends of problem distribution for a particular performance value as string length increases.

4.4 Future Investigation

It would be interesting to further investigate behavior of hard and easy problems distribution with respect to changing string length. For example, characterize the rate of convergence as string length increases. It would be interesting to investigate the limiting distribution in greater detail, since our investigation indicates that a normal approximation is probably not a good choice.

References

References

- [1] Ball, N. R., *Adaptive Signal Processing via Genetic Algorithms and Self Organising Neural Networks*, The Institute of Electrical and Electronics Engineers' Genetic Algorithms, Neural Networks and Simulated Annealing Workshop proceedings, United Kingdom, 1990.
- [2] Bowen, P., Fuhrer, D., El Jazzar, M., *A Genetic Algorithm Approach to the "Blind" Traveling Salesman Problem*, University of Tennessee, Department of Computer Science, 1989.
- [3] Goldberg, D. E., *Genetic Algorithms in Search Optimization & Machine Learning*, Addison-Wesley Publishing Co. Inc., 1989, 59pp.
- [4] Hoel, P. G., *Introduction To Mathematical Statistics*, John Wiley & Sons, Inc., 1954, 73pp.
- [5] Holland, J. H., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.
- [6] Rényi A., *Probability Theory*, North-Holland Publishing Co., 1970, 387pp.
- [7] Press W. H., Flannery B. P., Teukolsky S. A., Vetterling W. T., *Numerical Recipes, The Art of Scientific Computing*, Cambridge University Press, 1987, 203pp, 305pp.
- [8] Vose, M. D., *Generating Random Numbers*, University of Tennessee, Department of Computer Science, 1989

Appendix

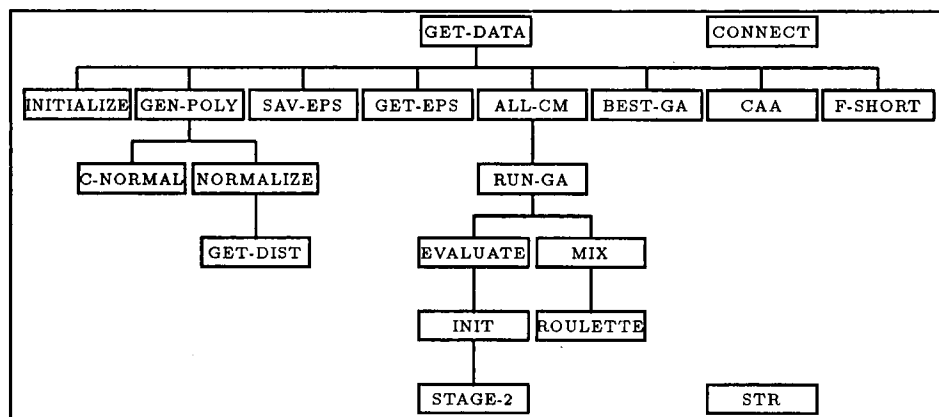


Figure 69: Hierarchical Diagram.

Program Modules

Figure 69 gives an overview of parts of program.

sav-eps Writes a eps (environment polynomial state) structure to a file. Structure saves environment variables related to a particular polynomial. File name is concatenation of the following 7 items:

- “l”string length
- “p”number of polynomials
- “s”population size
- “r”rating function
- “a”stopping value
- coefficient distribution type
- “-”polynomial number

For example, file name

17p198s4rR1a80U-104

contains environmental information for 9 SGAs operating on a length 7 polynomial and performance ratings for all 9 SGAs by rating function R1 after each generation, and this file is 104th polynomial out of a set of 198 polynomials.

String length	ℓ
Number of Polynomials	n
Population size	b
Rating function	$\{R1, R2\}$.
Stopping value	$\alpha \times 100$
Coefficient distribution	U-uniformly distributed, N-normally distributed.
Polynomial number	Polynomial identification number.

get-eps Restores structure saved by **sav-eps**.

get-gas Restores a **gap** (genetic algorithm parameter) from array that contains set of gaps. **gap** structure contains crossover rate, mutation rate, and population.

sav-gas Save current **gap** into array of gaps.

mix Replace current population by next population and ensures best string survives to the next generation.

evaluate Calculates statistics for current population.

roulette Returns a string in population with probability of the strings' fitness as a percent of the total fitness (roulette wheel selection scheme).

init Initializes roulette wheel selection routines.

stage-2 Sets up data structure for roulette wheel selection.

get-dist Fills fitness distribution table (**fd**). Fitness distribution table contains all unique fitness values and number of strings that have smaller fitness values.

normalize-fitness Modify randomly generated polynomial to a non negative function by subtracting from the polynomial its smallest value.

read-poly Reads in distribution files for generating random polynomials as described in Variations subsection of Random Polynomial section.

initialize Initialize global structures and variables for SGA. Additionally, sets up environment for collecting data from SGA.

caa Calculates accumulative average in an on going iterative averaging process and returns average performance rating at a particular generation.

get-data Calls initialization routines and implement iterative averaging method. This module also receives parameters from the user. For example, length of string, number of polynomials, population size, etc.

f-short Finds the polynomial with shortest extension in iterative averaging method.

all-cm Runs a combination of SGAP for number of generations. It retrieves each SGAP from array, runs requested generations, and saves result back to array.

run-ga Main SGA routine. It calls **evaluate** module, calculate SGA performance rating, and then call **mix** to form next generation.

comp-max2 Finds best result from a set of ratings from various SGAPs. It does component wise optimization and save result to **r-ary**.

comp-max Component-wise optimization of results from SGAPs.

get-smaller Return number of strings smaller than best fitness in population but are greater than population mean.

gen-poly Generates a full-term random polynomial.

rand Uses preset data structure and returns values with a set distribution.

str Returns string representation of passed in parameters.

connect Interface function for distributed processing. Informs control program that it is ready to receive commands, checks for command to execute, execute command, returns result, and repeat process until receiving the stop command.

Modules listed below are part of dispatcher program for distributed processing. Dispatcher reads in available processor file **processors.txt** and jobs need to be done from file **jobs.txt**.

get-mcp Get host processor name.

sg (start grunt), starts a worker on a processor.

init Initialize distributed processing process and grunt records.

read-file Reads in file specified in parameter until end of file.

start-all Starts workers on all available processors.

sap Sends stop signal to all workers and cleans up local file directory.

clean-all Coordinates clean up process by calling appropriate routines.

mcf Create a command file locally.

fap Find all workers that are ready to receive jobs.

pgsf Print all workers' status to a file.

save-result Copies results returned by workers into a single file.

assign-job Assign a job to a worker.

send-job Sends job command to worker.

ajf Test to see if all jobs are complete.

start Starts dispatcher.

Vita

Norman Ma was born in Taiwan and came to the states in 1975 when he was eleven years old. To his great amazement after arriving in this vast country, he could not believe he never traveled outside of a fifty square mile area for the previous eleven years! He decided he will eventually visit all popular tourist spots in this country. He spent his teenage years at Oak Forest and Palos Park, both are Southwest suburbs of Chicago. He graduated from the University of Illinois in 1985. While working as a software design engineer, he suffered a bout of *Academic Withdraw Syndrome* (AcaWiSy), pronounced ah-ka-wee-see, and decided to continue schooling and started graduate school at University of Tennessee in Fall of 1988. After graduation, he plans to get married, to continue attending school, and hopefully to live a productive live.

Permanent address: 16701 Phelps Lane, Huntington Beach, California.

This thesis was typeset with $\text{\LaTeX}$ in UNIX environment.