USER'S GUIDE FOR TOMLAB /GENO[3]

Marcus M. Edvall[1] and Isaac Siwale[2]

November 6, 2006

[1] Tomlab Optimization Inc., 855 Beech St #121, San Diego, CA, USA, medvall@tomopt.com.
[2] Ike's Research Ltd

Contents

1 Introduction

1.1 Overview

Welcome to the TOMLAB /GENO (General Evolutionary Numerical Optimiser) User's Guide.

This document describes the usage of a program called GENO. GENO is an acronym for **General Evolutionary Numerical Optimiser**: the word general is here used not in the sense of GENO being "able to solve all problems", but rather in the sense that it is effective on a relatively wide range of problems as compared to most existing algorithms. GENO is a real-coded genetic algorithm that can be used to solve uni- or multi-objective optimization problems. The problems presented may be static or dynamic in character; they may be unconstrained or constrained by equality or inequality constraints, coupled with upper and lower bounds on the variables. The variables themselves may assume real or discrete values in any combination. In fact, except for the relatively benign requirement that, if present, all equation constraints should preferably be affine in the current control, the algorithm does not require the problem presented to have any other special structure. Although the generic design of the algorithm assumes a multi-objective dynamic optimization problem, GENO may be "specialized" for other classes of problems such as the general static optimization problem, the "mixed-integer" problem, and the two-point boundary value problem, by mere choice of a few parameters. Thus, not only can GENO compute different types of solution to multi-objective problems, it may also be set to generate real or integer-valued solutions, or a mixture of the two as required, to uni-objective static and dynamic optimization problems of varying types. These properties are easily pre-set at the problem set-up stage of the solution process. The design of GENO includes a quantization scheme that significantly enhances the rate of convergence, as well as the quality of the final solution.

The following sections describe the algorithm and TOMLAB format in more detail. There are several test problem included with the TOMLAB distribution that illustrates the use.

1.2 Contents of this Manual

- Section 1 provides a basic overview of the GENO solver.
- Section 2 shows how to access the solver.
- Section 3 describes all the fields used by the solver as well as the options to set.
- Section 4 illustrates how to solve a simple test case.
- Section 5 shows the screen and file output.
- Section 6 contains information on how to access the test set.
- Section 7 provides algorithmic details about the solver.

1.3 More information

Please visit the following links for more information:

- http://tomopt.com/tomlab/products/geno/

1.4 Prerequisites

In this manual we assume that the user is familiar with nonlinear programming, setting up problems in TOMLAB (in particular constrained nonlinear (**con or glc**) problems) and the Matlab language in general.

2 Using the Matlab Interface

The GENO solver is accessed via the *tomRun* driver routine, which calls the *genoTL* interface routine. The solver itself is located in the MEX file *geno.dll*.

Table 1: The interface routines.

Function	Description	Section	Page
genoTL	The interface routine called by the TOMLAB driver routine *tomRun*. This routine then calls the MEX file *geno*	3.1	6
glcAssign	The routine that creates a problem in the TOMLAB format. This is for problems with a single objective.		
clsAssign	The routine that creates a problem in the TOMLAB format for multi-objective optimization.		

3 TOMLAB /GENO Solver Reference

A detailed description of the TOMLAB /GENO solver interface is given below. Also see the M-file help for *genoTL.m*.

3.1 genoTL

Purpose
Solve global optimization problems as described below.

$$
\begin{array}{cccccc}
\min\limits_{x} & f(x) & & & & \\
 & & & & & \\
 & x_L & \leq & x & \leq & x_U, \\
s/t & b_L & \leq & Ax & \leq & b_U \\
 & c_L & \leq & c(x) & \leq & c_U
\end{array} \qquad (1)
$$

where $x \in \mathbb{R}^n$, $f(x) \in \mathbb{R}$, $A \in \mathbb{R}^{m_1 \times n}$, $b_L, b_U \in \mathbb{R}^{m_1}$ and $c(x) \in \mathbb{R}^{m_2}$.
The variables $x \in I$, the index subset of $1, ..., n$, are restricted to be integers.

or the general format for multi-objective optimization:

$$
\begin{array}{cccccc}
\min\limits_{x} & J(x) = r(1)r(2)r(3)... & & & & \\
 & & & & & \\
 & x_L & \leq & x & \leq & x_U, \\
s/t & b_L & \leq & Ax & \leq & b_U \\
 & c_L & \leq & c(x) & \leq & c_U
\end{array} \qquad (2)
$$

where $x, x_L, x_U \in \mathbb{R}^n$, $r(x) \in \mathbb{R}^M$, $A \in \mathbb{R}^{m_1 \times n}$, $b_L, b_U \in \mathbb{R}^{m_1}$ and $c_L, c(x), c_U \in \mathbb{R}^{m_2}$.
The variables $x \in I$, the index subset of $1, ..., n$, are restricted to be integers.

Calling Syntax
Prob = clsAssign(...);
Result = tomRun('GENO',Prob,...)

Description of Inputs

Prob Problem description structure. The following fields are used:

A Linear constraints coefficient matrix.

x_L, x_U Bounds on variables.

b_L, b_U Bounds on linear constraints.

Prob Problem description structure. The following fields are used:, continued

c_L, c_U Bounds on nonlinear constraints.

MIP Structure with fields defining the integer properties of the problem. The following fields are used:

IntVars Vector designating which variables are restricted to integer values. This field is interpreted differently depending on the length.

If length(IntVars) = length(x), it is interpreted as a zero-one vector where all non-zero elements indicate integer values only for the corresponding variable.

A length less than the problem dimension indicates that IntVars is a vector of indices for the integer variables, for example [1 2 3 6 7 12]

A scalar value K restricts variables x(1) through x(K) to take integer value only.

PriLevOpt Print level for the solver.

GENO Structure with GENO solver specific fields.

PrintFile Name of file to print progress information and results to.

GENO.options Structure with special fields for the GENO solver:

adj_mode This parameter is problem-dependent. It should be set to 's' for uni-objective optimization problems, or if one seeks a Nash equilibrium solution of a multi-objective problem; it should be to 'g' in all other cases.
Default: None

bm_rate This parameter is the probability of boundary mutation and is returned from a simple function. It is problem-dependent but the default value is usually efficient.
Default: 0.005

constraints_check This parameter allows one to choose whether or not to display (1) values of the constraints at the end of the program run.
Default: 0

d_factor This is a weighting factor on the direction component of the differential cross-over operator. The parameter is problem-dependent but the default range is usually efficient.
Default: 0.15 - 0.8

Prob Problem description structure. The following fields are used:, continued

m_rate This parameter is the probability of ordinary mutation and is returned from a simple function. It is problem-dependent but the default value is usually efficient.
Default: 0.05

maximise This parameter is problem-specific: set this to true (1) if the problem is about maximization.
Default: 0

p_a_xover This parameter is problem-dependent: it specifies the probability threshold for the arithmetic cross-over operator. The default value is usually efficient.
Default: 0.55

p_agents This parameter is problem-specific: it declares the number of sub-objective in a multi-objective problem.
Default: None

p_b_xover This parameter is problem-dependent: it specifies the probability threshold for the boundary cross-over operator. The default value is usually efficient.
Default: 0.005

p_d_xover This parameter is problem-dependent: it specifies the probability threshold for the differential cross-over operator. The default value is usually efficient.
Default: 0.55

p_eqms A constant equal to 1.
Default: 1

p_h_xover This parameter is problem-dependent: it specifies the probability threshold for the heuristic cross-over operator. The default value is usually efficient.
Default: 0.55

p_maxgens This parameter specifies the maximum number of generations that the algorithm will execute. The most efficient value is dependent on the problem and the population size but it would be safe to assume that GENO solves most problems within 500 generations.
Default: None

p_mingens This parameter should always be 2.
Default: 2

p_order This parameter is problem-specific: it specifies the total number of variables in the problem.
Default: None

Prob Problem description structure. The following fields are used:, continued

p_plan This parameter is problem-specific: it specifies the length of the control vector along the time dimension.
Default: None

p_popsize This parameter specifies the population size. The most efficient value is problem-dependent but most likely to be within the range shown.
Default: 10 - 30

p_s_xover This parameter is problem-dependent: it specifies the probability threshold for the simple cross-over operator. The default value is usually efficient.
Default: 0.55

p_shuffle This parameter is problem-dependent: it specifies the probability threshold for shuffling the population. The default value is usually efficient.
Default: 0.55

p_u_xover This parameter is problem-dependent: it specifies the probability threshold for the uniform cross-over operator. The default value is usually efficient.
Default: 0.55

pos_orth This parameter is problem-specific: set this to false (0) if the static constraints are of the 'less than' type.
Default: 1

quantum_0 This parameter is problem-dependent: is specifies the initial size of quanta. In setting this parameter, the object should be to ensure that the initial population is sufficiently diverse on all dimensions. In this regard, a choice of the smaller between 0.1 and 10% of the smallest variable range is normally efficient. But if one seeks an integer solution, then this parameter should be set to 1.
Default: None

rand_seed This is a seed value for the random number generator.
Default: None

solution_type This parameter is problem-dependent: it defines the type of solution sought.
Default: None

timer This parameter declares whether to display (1) GENO's loop time at the end of the program run.
Default: 0

vars This parameter is problem-specific: it is an 'incidence matrix' that shows what variables are in each sub-problem of the multi-agent optimization problem.
Default: None

Prob Problem description structure. The following fields are used:, continued

vdu_output This parameter declares whether to display (1) progress of the best chromosome or its fitness.
Default: 0

view_vars In conjunction with vdu_output, this parameter allows one to choose between viewing (1) the variables in the best chromosome or its fitness.
Default: 0

Description of Outputs

Result Structure with result from optimization. The following fields are set:

f_k Function value at optimum.

x_k Solution vector.
x_0 Initial solution vector.

c_k Nonlinear constraint residuals.

xState State of variables. Free == 0; On lower == 1; On upper == 2; Fixed == 3;
bState State of linear constraints. Free == 0; Lower == 1; Upper == 2; Equality == 3;
cState State of nonlinear constraints. Free == 0; Lower == 1; Upper == 2; Equality == 3;

Ax Values of linear constraints.

ExitFlag Exit status from GENO (TOMLAB standard). 0 = Optimal solution found.
ExitText Exit text from GENO.
Inform GENO information parameter.
0 = Optimal: found an optimal solution.
Other = No optimal solution found.

FuncEv Number of function evaluations.
ConstrEv Number of constraint evaluations.
QP.B Basis vector in TOMLAB QP standard.

Solver Name of the solver (GENO).
SolverAlgorithm Description of the solver.

GENO Subfield with GENO specific results.

4 A Simple Example

At least two parts are needed when solving a problem with GENO, a main file and a file defining the objective function.

The following text can be entered into any Matlab m-file (save as genotest_f):

```
function f = genotest_f(x, Prob)

A = [4 4 4 4;1 1 1 1;8 8 8 8;6 6 6 6;3 7 3 7];
c = [.1 .2 .2 .4 .4]';
f=0;
for i = 1:5
   f = f - 1./( (x-A(i,:)' )'*( x-A(i,:)' ) + c(i) );
end
```

This defines the objective function to minimize.

The main file will define all static information about the problem, such as bounds and linear constraints:

```
Name = 'GENO TEST';
x_L = [ 0  0  0  0]';  % Lower bounds for x.
x_U = [10 10 10 10]';  % Upper bounds for x.

Prob = glcAssign('glbQG_f', x_L, x_U, Name);

Result = tomRun('GENO', Prob, 1);
```

In order to run this *glbQG* may be opened and the solver name switched to GENO.

5 Program Output

The results of a successful GENO execution are collected in the standard *Result* structure. It is also possible to print solution information to a text file (and to the Matlab screen). The full set of outputs (i.e. when all options are turned on) is shown below; a set of explanatory notes follow the listing.

```
A. GENO Parameters
   ===============
   optimisation period:                    1
   adjustment mechanism:                   s
   solution type:                          e

   population size:                        20
   random seed:                            240657
   initial quanta:                         0.100000

   mutation rate:                          0.050000
   boundary mutation rate:                 0.005000

   probability of simple crossover:        0.550000
   probability of arithmetic crossover:    0.550000
   probability of boundary crossover:      0.000000
   probability of heuristic crossover:     0.550000
   probability of differential crossover:  0.550000
   probability of shuffling population:    0.000000

   differential operator factor:           0.800000
   -------------------------------------------------


B. GENO Evolution
   ==============

   Generation
   Number                Objective1           Objective2

     0               2571734.743855            48.019520
    20                   685.227954             0.000000
    40                   682.883463             0.000000
    50                   682.807954             0.000000
    60                   680.814853             0.000000
    70                   680.787579             0.000000
    80                   680.702849             0.000000
    90                   680.692505             0.000000
   100                   680.654102             0.000000
   110                   680.651367             0.000000
   120                   680.645037            0.000000
   130                   680.639011            0.000000
   140                   680.638547            0.000000
   150                   680.637505            0.000000
   160                   680.631026            0.000000
   170                   680.630309            0.000000
   180                   680.630139            0.000000
   190                   680.630115            0.000000
   200                   680.630094            0.000000
   210                   680.630065            0.000000
   220                   680.630058            0.000000
   230                   680.630057            0.000000
   240                   680.630057            0.000000
   250                   680.630057            0.000000
   260                   680.630057            0.000000
   270                   680.630057            0.000000
   280                   680.630057            0.000000
   290                   680.630057            0.000000
   300                   680.630057            0.000000

   --------------------------------
C. Loop Time:  95.708000 seconds
   --------------------------------



D. GENO Optimal Solution
   =====================

   Best Vectors for Agent 1:

   Control1:       2.330499
   Control2:       1.951372
   Control3:       4.365727
   Control4:      -0.624487
   Control5:       1.594227
   State1:         0.000000       2.330499
   State2:         0.000000       1.951372
   State3:         0.000000       4.365727
   State4:         0.000000      -0.624487
   State5:         0.000000       1.594227

   Best Function Value:     680.630057


   Best Vectors for Agent 2:
```

```
   Control6:       0.000000
   Control7:       0.000000
   State6:         0.000000       0.000000
   State7:         0.000000       0.000000

   Best Function Value:       0.000000


E. Equations Vector at Solution
   ============================

      -0.477541
       1.038131


F. Inequalities Vector at Solution
   ===============================

     252.561724
     144.878178


G. Contents of Solution Matrix
   ===========================

       2.330499        0.000000
       1.951372        0.000000
       4.365727        0.000000
      -0.624487        0.000000
       1.594227        0.000000
       0.000000        2.330499
       0.000000        1.951372
       0.000000        4.365727
       0.000000       -0.624487
       0.000000        1.594227
     680.630057      680.630057
       0.000000        0.000000
       0.000000        0.000000
       0.000000        0.000000
       0.000000        0.000000
       0.000000        0.000000
```

Points to Note: GENO Output

- Viewing the Output - The screen/file output is controlled by the parameter Prob.PriLevOpt whose default value is 0; this output may be turned on (1).

- Timing Program Run - The evolution loop timer is controlled by the parameter timer whose default value is false (0); the timer may be turned on by including the assignment timer = 1 (Prob.GENO.options.timer).
- Checking Feasibility - GENO allows the user to check the feasibility of the computed solution via the parameter called constraint_check whose default value false (0); the values of the various constraints at the solution may be viewed by including the assignment constraint_check = true (1) (Prob.GENO.options.constraint_check).
- Program Output File - GENO's output is directed to a text file (Prob.GENO.PrintFile). The text file output cannot be turned off.

6 GENO Test Set

There are 21 example problems supplied by the authors of GENO included with the TOMLAB distribution. The problems are assembled in 7 separate files (geno_prob, geno_f, geno_g, geno_H, geno_c, geno_dc and geno_d2c). GENO itself will only utilize the information in _f and _c, but first and in some cases second order derivatives are supplied to enable smooth execution of other solver.

To create and run one of these problem the following code may be executed:

```
probNumber = 1;          % Any number from 1 to 21.

Prob = probInit('geno_prob', probNumber);

Result = tomRun('GENO', Prob, 1);
```

Of course the algorithm may also be tested using any of the example problems in the general TOMLAB test suite.

7 GENO Algorithmic Details

7.1 Introduction

The Genetic Algorithm (or GA for short) is a recent development in the arena of numerical search methods. GAs belong to a class of techniques called *Evolutionary Algorithms*, including Evolutionary Strategies, Evolutionary Programming and Genetic Programming. One description of GAs is that they are stochastic search procedures that operate a population of entities; these entities are suitably coded candidate solutions to a specific problem; and the GA solves the problem by artificially mimicking evolutionary processes observed in natural genetics.

Naturally, terminology from the field of natural genetics is used to describe genetic algorithms. In a biological organism, the structure that encodes how the organism is to be constructed is called the *chromosome*. One or more chromosomes may be required to completely specify the organism; the complete set of chromosomes is called a *genotype*, and the resulting organism is called a *phenotype*. Each chromosome comprises a set of genes, each with a specific position or locus on the chromosome. The loci in conjunction with the values of the genes (which are called the alleles) determine what characteristics are expressed by the organism.

In the GA analogy, a problem would first be coded and in this regard, genetic algorithms have traditionally used a binary representation in which each candidate solution is coded as a string of 0's and 1's. The GA's "chromosomes" are therefore the strings of 0's and 1's, each representing a different point in the space of solutions. Normally each candidate solution (or "organism") would have only one chromosome, and so the terms *organism*, *chromosome* and *genotype* are often used synonymously in GA literature. At each position on a chromosome is a *gene* that can take on the *alleles* 0 or 1; the phenotype is the *decoded* value of the chromosome. The population of candidate solutions represent a sample of different points of the search space, and the algorithm's genetic processes (see below) are such that the chromosomes evolve and become better and better approximations of the problem's solution over time.

Before a GA can be run, a *fitness function* is required: this assigns a figure of relative merit to each potential solution. Before fitness values can be assigned, each coded solution has to be decoded and evaluated, and the module designed to do this is generally called the *evaluation function.* But whereas the evaluation function is a problem-specific mapping used to provide a measure of how individuals have performed in the problem domain, the fitness function, on the other hand, is a problem-independent mapping that transforms evaluation values into an allocation of *reproductive* opportunities. For any given problem therefore, different fitness functions may be defined. At each generation, the chromosomes in the current population are rated for their effectiveness as candidate solutions, and a process that emulates nature's survival-of-fittest principle is applied to generate a new population which is then "evolved" using some genetic operators defined on the population. This process is repeated a sufficient number of times until a good-enough solution emerges. The three primary genetic operators focused on in practice are *selection*, *crossover* and *mutation.*

7.2 Selection

This operator is sometimes called *Reproduction.* The reproduction operation is in fact comprised of two phases: the selection mechanism and the sampling algorithm. The selection mechanism assigns to each individual x, a positive real number, called the *target sampling* rate (or simply: fitness), which indicates the *expected* number of offspring reproduced by x at generation t. In the commonly used *fitness proportionate selection* method, an individual is assigned a target sampling rate equal to the ratio of the individual's evaluation to the average evaluation of all chromosomes in the population. This simple scheme however suffers from the so-called scaling problem where a mere shift of the underlying function can result in significantly different fitness values. A technique that has been suggested to overcome this is to assign target sampling rates according to some form of population ranking scheme. Here, individuals are first assigned a rank based on their performance as determined by the evaluation function; thereafter, the sampling rates are computed as some linear or non-linear function of the ranks.

After fitness values have been assigned, the sampling algorithm then reproduces copies of individuals to form an intermediate mating population. The most common method of sampling the population is by the *roulette wheel* method in which each individual is assigned a slice of a virtual roulette wheel which is proportional to the individual's fitness. To reproduce a population of size P, the wheel is spun P times. On each spin, the individual under the wheel's maker is selected to be in the mating pool of parents who are to undergo further genetic action. An alternative approach and one which minimizes *spread* is Stochastic Universal Sampling (SUS). Here, rather than spin the wheel P times to select P parents, SUS spins the wheel once but with P equally spaced pointers which are used to select the P parents simultaneously. Reproduction may also be done by a tournament selection. A typical implementation is as follows. Two individuals are chosen at random from the population and a random number r is chosen between 0 and 1. If r < k (where k is a tuning parameter, say 0.75), the fitter of the two individuals is selected to go into the mating pool; otherwise the less fit individual is chosen. The two are then returned to the original population and can be selected again.

Reproductive processes may be implemented in *generational* or *steady-state* mode. Generational reproduction replaces the entire population with a new population, and the GA is said to have a generation gap of 1. Steady-state reproduction on the other hand replaces only a few individuals at a time. *Elitism* is an addition to selection that forces the GA to retain some number of the best individuals at each generation. Such individuals can be lost if they are not selected to reproduce or if they are operated on by the genetic operators.

The selection operator is the driving force in GAs, and the *selection pressure* is a critical parameter. Too much selection pressure may cause the GA to converge prematurely; too little pressure makes the GA's progress towards the solution unnecessarily slow.

7.3 Crossover

The crossover operation is also called recombination. It is generally considered to be the main exploratory device of genetic algorithms. This operator manipulates a pair of individuals (called parents) to produce two new individuals (called offspring or children) by exchanging corresponding segments from the parents' coding. The simplest form of this operator is the single-point crossover, and this is as illustrated below where the crossover point is the position marked by the symbol, —.

```
Parent1: (a b c | d e)      Cross over at position 3     Child1: (a b c | 4 5)
                            ------------------------->
Parent2: (1 2 3 | 4 5)                                   Child2: (1 2 3 | d e)
```

Other binary-coded crossover operators which are variations of the above scheme have since been defined, e.g., two-point crossover, uniform crossover and shuffle crossover. For real-coded GAs, recombination is usually defined in a slightly different way. We mention three crossover operators that are employed by GENO:

- **ARITHMETIC CROSSOVER.** This operator produces two offspring that are convex combinations of the parents. If the chromosomes $c_k^v = (x_k^1, x_k^2, ..., x_k^N)$ and $c_k^w = (x_k^1, x_k^2, ..., x_k^N)$ are selected for crossover, the offspring are defined as:

  $c_k^1 = \alpha * c_k^v + (1 - \alpha) * c_k^w$ and $c_k^2 = \alpha * c_k^w + (1 - \alpha) * c_k^v$, where $\alpha \in [x\_L, x\_U]$

- **HEURISTIC CROSSOVER.** This operator combines two chromosomes and produces one offspring as follows: if c_k^v and c_k^w are two parent chromosomes such that the fitness of c_k^v is not worse than that of c_k^w, then the offspring is:

  $c_k^x = c_k^v + \alpha * (c_k^v - c_k^w)$, where $\alpha \in [x\_L, x\_U]$

  Here, the idea is to use the ”quasi-gradient” of the evaluation function as a means of directing the search process.

- **DIFFERENTIAL CROSSOVER.** This operator uses three parents: one parent is taken as the ”base”, and the other two are used to generate the search direction. Thus, if $\bar{u}_T^B$, $\bar{u}_T^V$ and $\bar{u}_T^W$, are the parent chromosomes with $\bar{u}_T^B$ as the ”base”, then the offspring are:

  $\bar{u}_T^1 = \bar{u}_T^B + \alpha * (\bar{u}_T^W - \bar{u}_T^V)$ and $\bar{u}_T^2 = \bar{u}_T^B + \alpha * (\bar{u}_T^V - \bar{u}_T^W)$, where $\alpha \in [x\_L, x\_U]$

In GENO, the factor α is pre-selected from the unit interval, although random variable may also be used.

7.4 Mutation

By modifying one or more of the gene values of an existing individual, mutation creates new individuals and thus increases the variability of the population. The mutation operator ensures that the probability of reaching any point in the search space is never zero. The mutation operator is applied to each gene of the chromosome depending on whether a random deviate drawn from a certain probability distribution is above a certain threshold. Usually the uniform or normal distribution is assumed. Again, depending on the representation adopted, variations of the basic mutation operator may be defined.

7.5 Why Genetic Algorithms Work

Currently, there are several competing theories that attempt to explain the macroscopic behavior of GAs. The original description of GAs as *schema processing* algorithms by John Holland (1975) has underpinned most of the theoretical results derived to date. However, other descriptive models based on Equivalence Relations, Walsh functions, Markov Chains and Statistical Mechanics have since been developed. A survey of these models is beyond the scope of this introductory exposition. Instead, we provide a sketch of the logic leading up to one of the main explanatory models, namely *The Building Block Hypothesis.* We begin by stating some definitions.

- **DEFINITION A1:** [Schema; Schemata] A *schema* is a template that defines the similarities among chromosomes which is built by introducing the *don't* care symbol (*) in the alphabet of genes. It represents all chromosomes which match it at every locus other than the '*' positions. For example, the schema (1 0 * * 1) represents of four chromosomes, i.e.: (1 0 1 1 1), (1 0 1 0 1), (1 0 0 1 1) and (1 0 0 0 1). A collection of schema is a *schemata.*
- **DEFINITION A2:** [Order] The *order* of a schema S (denoted by o(S)) is the number of non-*don't care* positions in the schema. For example, the schemata S1 = (1 0 * * 1), S2= (* 0 * * 1), S3= (* * 1 * *) are of orders 3, 2 and 1, respectively.
- **DEFINITION A3:** [Defining Length] The *defining length* of a schema S (denoted by $\delta(S)$) is the *positional distance* between the first and last *fixed* positions (i.e., the non-don't care sites) in the schema. It defines the compactness of the information contained in the schema. For example, defining lengths of the three schemata S1 = (1 0 * * 1), S2= (* 0 * * 1), S3= (* * 1 * *) are $\delta(S1) = 4$, $\delta(S2) = 3$ and $\delta(S3) = 0$, respectively.
- **DEFINITION A4:** [Schema Fitness] The *schema fitness* is the average of the fitness of all chromosomes matched by the schema in a given population. That is, given the evaluation function *eval(.)* defined on a population of chromosomes x_j of size P, the fitness of schema S at generation t is:

$$eval(S,t) = \sum_{j=1}^{P} eval(x_j/P) \tag{3}$$

The evolutionary process of GAs consists of four basic steps which are consecutively repeated, namely:

```
  t <- t+1
  select P(t) from P(t-1);
  recombine and mutate P(t);
  evaluate P(t)
```

The main evolutionary process takes place in the *select*, *recombine* and *mutate* phases. After the selection step, we can expect to have $\xi(S, t+1)$ chromosomes matched by the schema S in the mating pool. For an average chromosome matched by the schema S, the probability of its selection is equal to eval(S,t)/F(t) where F(t) is the total fitness for the whole population. Since the number of chromosomes matched by schema S before selection is $\xi(S, t)$, and the number of single chromosome selections is P, it follows that:

$$\xi(S, t+1) = \xi(S, t) * P * eval(S, t)/F(t) \tag{4}$$

or, in terms of the average fitness of the population, $\bar{F}(t)$:

$$\xi(S, t+1) = \xi(S, t) * eval(S, t)/\bar{F}(t) \tag{5}$$

In other words, the number of chromosomes grows as the ratio of the fitness of the schema to the average fitness of the population. This means that an above-average schema receives an increasing number of matching chromosomes in the next generation, a below-average schema receives a decreasing number of chromosomes, and an average schema remains the same.

The schema growth equation 5 however has to be modified to take into account the destructive effects of recombination and mutation. For chromosomes of length m, the crossover site is selected uniformly from among (m - 1) possible sites. A schema S would be destroyed if the crossover site is located within its defining length. The probability of this happening is $p_d(S) = \delta(S)/(m-1)$ and, hence, the probability of a schema surviving the crossover operation is,

$$p_s(S) = 1 - (\delta(S)/(m-1)) \tag{6}$$

The crossover operator is however only applied selectively according to some probability (p_s, say). Furthermore, even when the crossover site is within the defining length there is always a finite chance that the schema may survive. These considerations dictate the modification of 6 to,

$$p_s(S) \geq 1 - p_c * (\delta(S)/(m-1)) \tag{7}$$

Thus, the combined effects of *selection* and *recombination* are summarized by:

$$\xi(S, t+1) \geq \xi(S, t) * (eval(S, t)/\bar{F}(t)) * (1 - p_c * (\delta(S)/(m-1))) \tag{8}$$

The mutation operator changes a single gene with probability p_m. It is clear that *all* the fixed positions of a schema must remain intact if the schema is to survive mutation. Since each mutation is independent of all others, the probability of a schema S surviving mutation is therefore:

$$p_s(S) = (1 - p_m)^{o(S)} \tag{9}$$

And for $p_m \ll 1$, $p_s(S)$ may be approximated by the first two terms of its binomial expansion, i.e.:

$$p_s(S) \approx 1 - p_m * o(S) \tag{10}$$

Therefore, ignoring higher-order terms involving products of p_c and p_m, the combined effects of selection, crossover and mutation is summarized by the following reproductive schema growth inequality:

$$\xi(S, t+1) \geq \xi(S,t) * (eval(S,t)/\bar{F}(t)) * (1 - p_m * o(S) - p_c * (\delta(S)/(m-1))) \tag{11}$$

Clearly, the disruptive effects of mutation and crossover are greater, the higher the order, and the longer the defining length of the schema. One can therefore expect that later generations of chromosomes would increasingly be comprised of short, low-order schemata of above-average fitness. This observation is captured by the *Schema Theorem* which states:

- **THEOREM A1** [Schema Theorem] Short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations of a genetic algorithm.

An immediate result of this theorem is the assertion that genetic algorithms explore the search space by short, low-order schemata which are used for information exchange during recombination. This observation is expressed by the Building Block Hypothesis which states:

- **HYPOTHESIS A1** [Building Block Hypothesis] A genetic algorithm seeks near-optimal performance through the juxtaposition of short, low-order, high-performance schemata called building blocks.

Over the years, many GA applications which support the building block hypothesis have been developed in many different problem domains. However, despite this apparent explanatory power, the hypothesis is not universally valid. In particular, it is easily violated in the so-called deceptive problems.

7.6 Setting GA Parameters

Before one can use a GA, one needs to specify some parameter values namely the selection pressure, the population size, and the crossover and mutation rates. Both theoretical and empirical studies show that ”optimal” values for these parameters depend on how difficult the problem at hand is. And since prior determination of the degree of difficulty a particular problem poses is hard, there are no generally accepted recipes for choosing effective parameter values in every case. However, many researchers have developed good heuristics for these choices on a variety of problems, and this section outlines some their recommendations.

7.6.1 Experimental Studies

De Jong (1975)

Kenneth A. De Jong tested various combinations of GA parameters on five functions with various characteristics including continuous and discontinuous, convex and non-convex, uni-modal and multi-modal, deterministic and noisy for his PhD Thesis. His function suite has since been adopted by many researchers as the standard test bed for assessing GA designs.

De Jong used a simple GA with roulette wheel selection, one-point cross-over and simple mutation to investigate the effects of four parameters namely: population size, crossover rate, mutation rate and the generation gap. His main conclusions were that:

- Increasing the population size resulted in better long-term performance, but smaller population sizes responded faster and therefore exhibited better initial performance.
- Mutation is necessary to restore lost alleles but this should be kept low at a low rate for otherwise the GA degenerates into a random search.
- A cross-over probability of around 0.6 worked best. But increasing the number of cross-over points degraded performance.
- A non-overlapping population model worked better in general.
- In summary, he concluded that the following set of parameters were efficient (at least for the functions that he studied): population size - 50 - 100; crossover probability - 0.6; mutation probability - 0.001; generation gap - 1.

De Jong's work was very important in that it provided practical guidelines for subsequent applications of GA's. His recommendations for the various parameters have been so widely adopted that they are sometimes referred to as "the standard settings". But subsequence research revealed that applying De Jong's parameter values cases can be a serious mistake in some cases.

Schaffer, Caruana, Eshelman and Das (1989)

Recognizing that parameter values can have a significant impact on the performance of a GA and that a more thorough investigation was needed, Schaffer et al. (1989) expanded De Jong's experimental setup. In addition to the five functions that he had studied, they introduced five more and performed a more exhaustive assessment of the direct and cross effects of the various parameters on a GA's performance. A notable observation they made was that good GA performance results from an inverse relationship between population size and the mutation rate, i.e. high mutation rates were better for small population sizes and low mutation rates were good for large populations. Whilst recognizing that their results may not generalize beyond the 10 functions in their test suite, they recommend the following parameter values:

- Population size: 20 - 30
- Mutation rate: 0.005 - 0.1

- Cross-over rate: 0.75 - 0.95

De Jong's work was very important in that it provided practical guidelines for subsequent applications of GA's. His recommendations for the various parameters have been so widely adopted that they are sometimes referred to as the "standard settings".

7.6.2 Theoretical Studies

Several researchers have theoretically analysed the dynamics of GAs. In his survey, Lobo (2000: chapter 3) reports the most notable of these as being the work on selection by Goldberg and Deb (1991); the work on mutation by Mûhlenbein (1992) and Bäck (1993); the research on population sizing by Goldberg, Deb and Clark (1992) and Harik et al. (1997); and the work on control maps by Goldberg, Deb and Thierens (1993) and Thierens and Goldberg (1993). The insights and practical implications afforded by these studies are summarized below.

- **On Selection.** In the absence of all other operators, repeated use of the selection operator would eventually result in a population comprised of the single chromosome with the highest fitness. Goldberg and Deb define the *takeover time* as the time it takes (as measured by the number of generations elapsed) for this event to occur. They derived takeover time formulae for various selection schemes and validated these using computer simulations. For fitness proportionate selection schemes, the takeover time depends on the fitness function distribution; for order-based selection, the takeover time is independent of the fitness function and is of the order O (log P), where P is the population size. Obviously the takeover time is increases in the presence of cross-over and mutation, but one must be careful not to exert too much selection pressure to cancel the diversifying effects of these operators.

- **On Mutation.** Independently of each other Mûhlenbein (1992) and Bäck (1993) analyzed the effects of mutation on a simple (1 + 1) evolutionary algorithm. They both concluded that for a chromosome of length L, the optimal fixed mutation rate is L^{-1}. Intuitively, it is easy to see why there should be this inverse relationship between chromosome length and mutation rate. Besides exploring the search space, the mutation (and to so extent, cross-over operation) can disrupt building blocks during the course of a GA run. And obviously, this is more likely to occur for long chromosomes than short ones since the operator is applied (with probability) to each gene. So in order to minimize building block disruption, one should decrease the mutation rate for relatively longer chromosomes.

- **On Population Size.** Studies on population size attempt to formulate schema growth equations similar to equation 5 that have population size as a variable. Unfortunately, population sizing equations are difficult to use in practice. Lobo notes:

  "In order to apply [the equation] the user has to know or estimate the selective advantage that a building block has over its closest competitors; he has to estimate the building block's fitness variance, he has to estimate the maximum level of deception in the problem at hand; and of course he has to hope that the building blocks are going to mix well, which may not occur in practice" (paraphrased from p.34)

  It is difficult to see how these population sizing studies can be used to further inform the choice of parameter values suggested by the empirical work of De Jong (1975) or Schaffer et al. (1989).

- **On Selection.** Increasing the population size resulted in better long-term performance, but smaller population sizes responded faster and therefore exhibited better initial performance.

7.6.3 Parameter Adaptation

We mention, in passing, parameter adaptation techniques. These methods change GA parameters as the search progresses. There are three main approaches: (1) centralized methods change parameter values based on a central learning rule; (2) decentralized methods encode the parameters values into the chromosomes themselves; (3) meta-GA's attempt to optimize parameter values by evolving these values for the actual GA that is run at a lower level using the parameters identified by the meta-level GA. The main advantage of a GA so designed is that the user is no longer required to specify parameter values prior to executing the search.

7.7 Concluding Remarks

Genetic Algorithms are simple and yet powerful search and optimization procedures that are widely applicable. Unfortunately, our current knowledge is such that one cannot rigorously predict whether a GA is going to efficient in any given instance due to the difficult in choosing the parameters of the algorithm. Nevertheless, the parameters recommended for GENO are efficient, at least on the examples reported. These parameters were arrived at after extensive "trial and error" experimentation guided by the empirical and theoretical work outlined above: they are summarized in Section 3.1.