GENETIC: a Modular Genetic Algorithm

David L. Battle

Michael D. Vose

This is essentialy the masters thesis of David Battle. It describes a program which supplies a unifying framework for the efficient implementation of simple genetic algorithms. Design philosophy, algorithms, and particulars of the implementation are discussed and a performance comparison with an existing system (GENESIS) is presented.

C Code is available from vose@cs.utk.edu. This work was supported by the National Science Foundation (IRI-8917545).

TABLE OF CONTENTS

CHAPTER		PAGE
1.	**INTRODUCTION**	**1**
1.1	General Description of Genetic Algorithms	1
1.2	Types of Genetic Algorithms	2
1.3	Components of Genetic Algorithms	2
1.3.1	Selection	2
1.3.2	Crossover	3
1.3.3	Mutation	3
1.3.4	Fitness Functions	4
1.3.5	Change of Representation or Transformation	4
1.4	Unifying Genetic Algorithms	5
2.	**THE PROGRAM**	**6**
2.1	General Overview	6
2.2	Detailed Overview	8
2.2.1	Initialization	8
2.2.2	Determining Number and Distriubtion of Strings	12
2.2.3	Creating New Strings	13
2.2.4	Forming the Next Generation	13
2.2.5	Saving Statistics	14
2.2.6	Saving Final State	14
2.3	Module Classes	15

2.4 Algorithms 16
2.4.1 Matrix Multiplication 16
2.4.2 Generalized Crossover 18
2.4.3 Multisymbol Mutation 18
2.4.4 Fast Mutation 19
2.4.5 Generating a Unique Collection of Strings 20
2.4.6 Computing Parity 21
2.5 Probability Distributions and Random Number Generation 22
2.5.1 The Uniform Distribution 22
2.5.2 The Binomial Distribution 23
2.5.3 The Geometric Distribution 25
2.5.4 The Hypergeometric Distribution 26
2.5.5 Random Number Generation 27
2.6 Using the Program 27
2.7 Comparison with GENESIS 29

3. MODULE INTERFACES AND EXTENDING THE GA 31

3.1 Population Initialization Modules 31
3.2 The Number of New Strings Module 33
3.3 Selection Modules 34
3.4 The Selection Distribution Module 36
3.5 Crossover Modules 38
3.6 The Crossmask Module 40
3.7 The Reorder Module 43
3.8 The Fitness Function Interface Module 44

3.9 The Fitness Function 46
3.10 The Replacement Module 48
3.11 The Mutation Module 51
3.12 The Statistics Module 54
3.13 The Transform Module 58
3.14 The Matrix Generation Module 60
3.15 Adding a New Parameter 62
3.16 Adding a New Module Class 65

4. FUTURE WORK 68
4.1 Graphical User Interface 68
4.2 Additional Modules 68
4.2.1 Multi-Symbol I/O 68
4.2.2 Whitley-Style Replacement 69
4.2.3 Roulette-Wheel Selection 69
4.2.4 Elitest Replacement Module 70
4.2.5 Module to Read User-Supplied Distributions 70
4.3 Improving Memory Efficiency 70
4.4 Saving/Restoring Random Number Generator State 70

5. CONCLUSION 72

BIBLIOGRAPHY 73

APPENDICES 75

A. Proof of Mutation Methods 76

B. Proof of Matrix Multiplication Optimization **80**

C. Proof of Matrix Gray Code Theorem **81**

D. Manipulating Primary Data Structures **86**

D.1 Manipulating The String Data Structure 86

D.2 Manipulating the Mask Data Structure 90

D.3 Manipulating the Population Data Structure 91

D.4 Manipulating Matrices 92

E. Sample Files **96**

E.1 Configuration File 96

E.2 Paramater File 104

E.3 Population File 116

E.4 Mask File 116

E.5 Matrix File 117

E.6 Fitness Table 119

E.7 GENESIS-Style Statistics Output File 119

CHAPTER 1

INTRODUCTION

1.1 General Description of Genetic Algorithms

Genetic algorithms are optimization methods which attempt to mimic the way in which biological genetics and survival of the fittest have produced solutions to the incredibly complex problem of creating organisms that are capable of thriving in the intricate and ever-changing environment we call Earth. Genetic algorithms operate on populations of strings of symbols which are taken to be from the domain of some function for which one wishes to find an extreme. In this thesis, it is assumed that this function (called the fitness function, or sometimes the objective function) is to be maximized. On each generation some of the strings are paired off and "crossed over" to produce offspring which then replace some of the old strings in the population. Crossover involves swapping some subset of the symbols of the two parent strings. Strings may also be "mutated" from time to time. This involves randomly changing one or more symbols in the string to some other symbol. The mechanism for encouraging improvement in the population is that, in general, strings with a higher fitness function value are more likely to reproduce (possibly multiple times) in loose analogy with the principle of survival of the fittest.

1.2 Types of Genetic Algorithms

There are many types of genetic algorithms in use which are distinguished by variations in how the mechanics of the process are carried out. Two main categories of GAs are Steady State and Generational. In Steady State GAs a single pair of strings are crossed at each time step, the offspring are mutated and then immediately join the population (sometimes subject to the constraint that they are unique or better) and are available to reproduce at the next timestep. In generational GAs much or all of the population is involved in crossover and mutation at each timestep (or generation).

1.3 Components of Genetic Algorithms

1.3.1 Selection

The process of selecting strings for crossover is another source of differentiation between different types of GAs. The two primary methods employed here are Ranking and Proportional. In Ranking selection, the strings are sorted by fitness or some other criteria and the string which falls in the the ith slot has probability $p[i]$ of being selected as a parent in each round of crossover where p is a fixed probability distribution. In proportional selection, each string is given a probability of selection which is proportional to (some monotonic function of) its fitness.

1.3.2 Crossover

The mechanics of crossover are also varied. Three primary types of crossover are one point, two point, and uniform. In one point crossover a single point is chosen between symbols of the parent strings. Both parents are cut at this point and the right hand portions are interchanged. Two point crossover is similar except that the strings are thought of as circles which are snipped at two points to form two segments from each parent. One of the two segments from one of the strings is swapped with the corresponding segment from the other string to form the children. Uniform crossover selects some set of symbols from one of the parents (with uniform probability over all such sets) and swaps these symbols with the corresponding symbols from the other parent. Crossover is also limited by some probability called the crossover rate. At each time when crossover is to be done, a Bernoulli trial is made and if it is successful, the parent strings are crossed over; if it is unsuccessful, they are simply cloned.

1.3.3 Mutation

Mutation is generally implemented such that each symbol in each string has equal probability of changing to a symbol different from what it currently is with uniform probability over all such possibilities on each generation. The probability of a symbol's changing in this way is called the mutation rate. Other special-purpose versions of mutation could be devised which differ from this method for special applications.

1.3.4 Fitness Functions

The fitness function is the most variable portion of a genetic algorithm because it encodes the problem to be solved. A trivial example of a fitness function would be one which assigns a fitness to each binary string which is the number of 1 symbols which it contains. A more complex example would be a function which returned the autocorrelation of a binary string argument.

1.3.5 Change of Representation or Transformation

Some researchers have investigated modification of the fitness function by applying a matrix transformation to the strings before the fitness function is applied in an effort to make the problem easier for the GA to solve. Although there are known classes of deceptive functions which are made non-deceptive by a matrix transformation [3], in general there is no known method for finding a matrix transformation which will result in a net improvement in the performance of the GA when the time to find the matrix is included in the runtime of the GA.

A matrix transformation can also be used to introduce Gray coding. A matrix with a band of ones on the main diagonal and on the subdiagonal and zeros elsewhere will result in a reflective Gray code. A proof of this is presented in Appendix C. The matrix which corresponds to the transformation matrix used by this program is actually the inverse of the Gray coding matrix. The inverse of the Gray Code matrix is a matrix which has

1's on and below the main diagonal and 0's above. The form of the inverse is proven in Appendix C. Gray coding is useful because it avoids "Hamming cliffs" where adjacent integers differ in every bit position, for example 63 (011111) and 64 (100000) [1].

To see why the inverse of the Gray coding matrix is appropriate, consider the two strings given in the example above. These are the strings which we would like the fitness function to see, because it presumably interprets what is sees as integers. If we transformed these two strings by the Gray coding matrix and placed them in the population, they would only differ in one bit position; a desirable feature for the "internal format" strings which are used by crossover and mutation. When the time comes to evaluate their fitness, however, they should be transformed by the inverse of the Gray coding matrix to return them to their original form.

1.4 Unifying Genetic Algorithms

With all the variability in GAs, it would be useful to have a unifying framework. This thesis describes a GA implementation which provides such a framework. There are modules for implementing all of the above-described options as well as others. An effort has been made to provide a efficient and reliable GA implementation which can be extended by the user. The design is modular with well-defined interfaces between modules. Extra speed has been attained at the cost of using more memory (building pre-computed tables for certain functions); however, such tables have been kept to a reasonable size so that there is memory left for large populations.

CHAPTER 2

THE PROGRAM

2.1 General Overview

In a traditional GA, a generation is carried out by first selecting a "gene-pool" of strings from the old population, then crossing over and mutating some of these while cloning and mutating others. The decision whether to crossover a pair of strings or clone them is typically based on whether a Bernoulli trial with a probability of success equal to the crossover rate succeeds or fails, respectively.

In this implementation, the decision of whether to crossover or clone is removed from the loop by deciding in advance how many strings will be crossed over based on a geometric distribution. Later, the vacant slots in the new population can be filled in the replacement phase by cloning and mutating strings from the old population. Doing things in this way also allows distributions other than the geometric distribution. For example, steady-state GA's (number of strings to be produced from crossover is the constant 1 (or 2)) can be implemented within the same framework.

Figure 1 illustrates the top level flow in the `genetic` program. First the initialization routines of all the different module types are called. Then in the main loop the probability distribution which will be used in selection

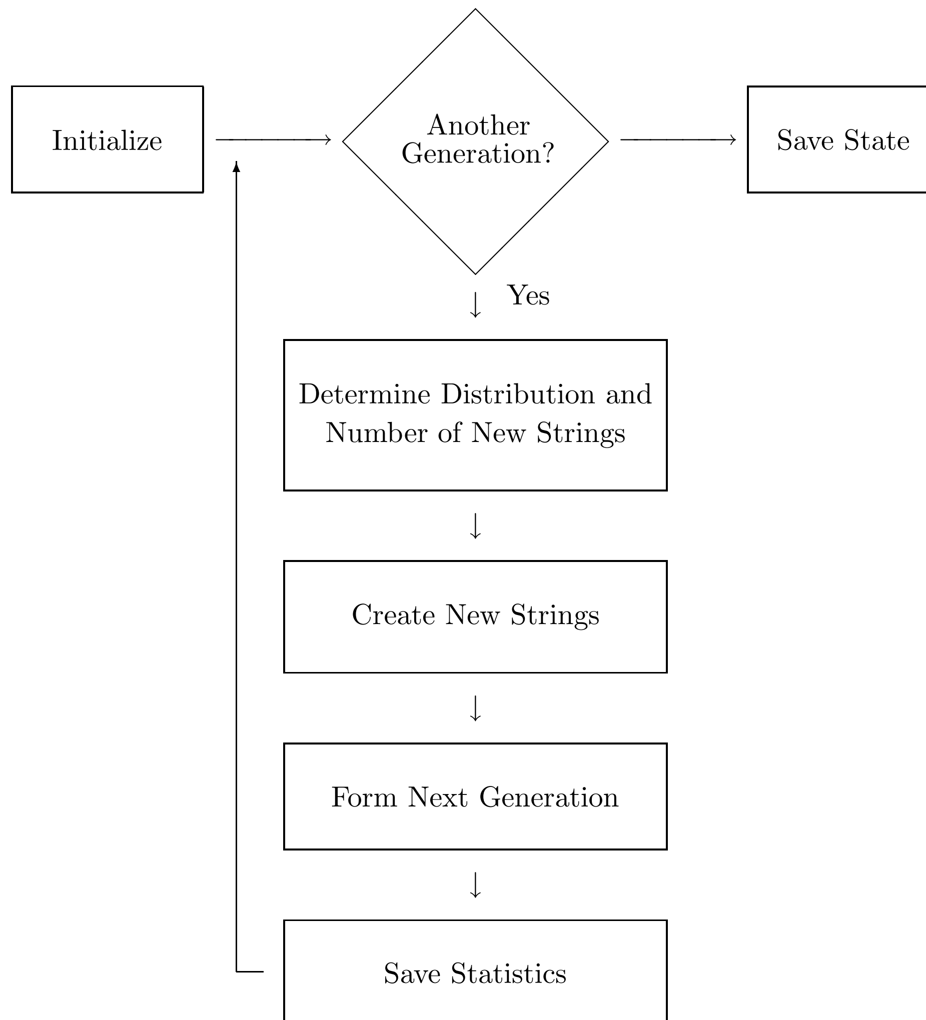

Figure 1. Top Level Flow Diagram

and the number of strings which will be generated from crossover in the current generation are determined. The new strings are created by selection and crossover. The next generation is formed from the newly generated strings and the old population. Finally statistical information about the population is saved and the loop is repeated. When the main loop is finished, state information to allow a restart is saved.

Figure 2 illustrates this process in more detail. Boxes with rounded corners are replaceable modules which are called, usually by the main program. Square cornered boxes are tasks which are performed by the main program or by required (non-replaceable) sub-modules. Specific module interfaces are described in the next chapter.

Figure 3 shows the different classes of replaceable modules and the currently implemented instances of these classes. There are two to five instances of each class which, in different combinations, provide a wide variety of GA's from which to choose.

2.2 Detailed Overview

2.2.1 Initialization

The program consists of several collections of modules together with a controling main program which implements the framework into which the modules fit. Some modules also rely on functionality provided by other modules.

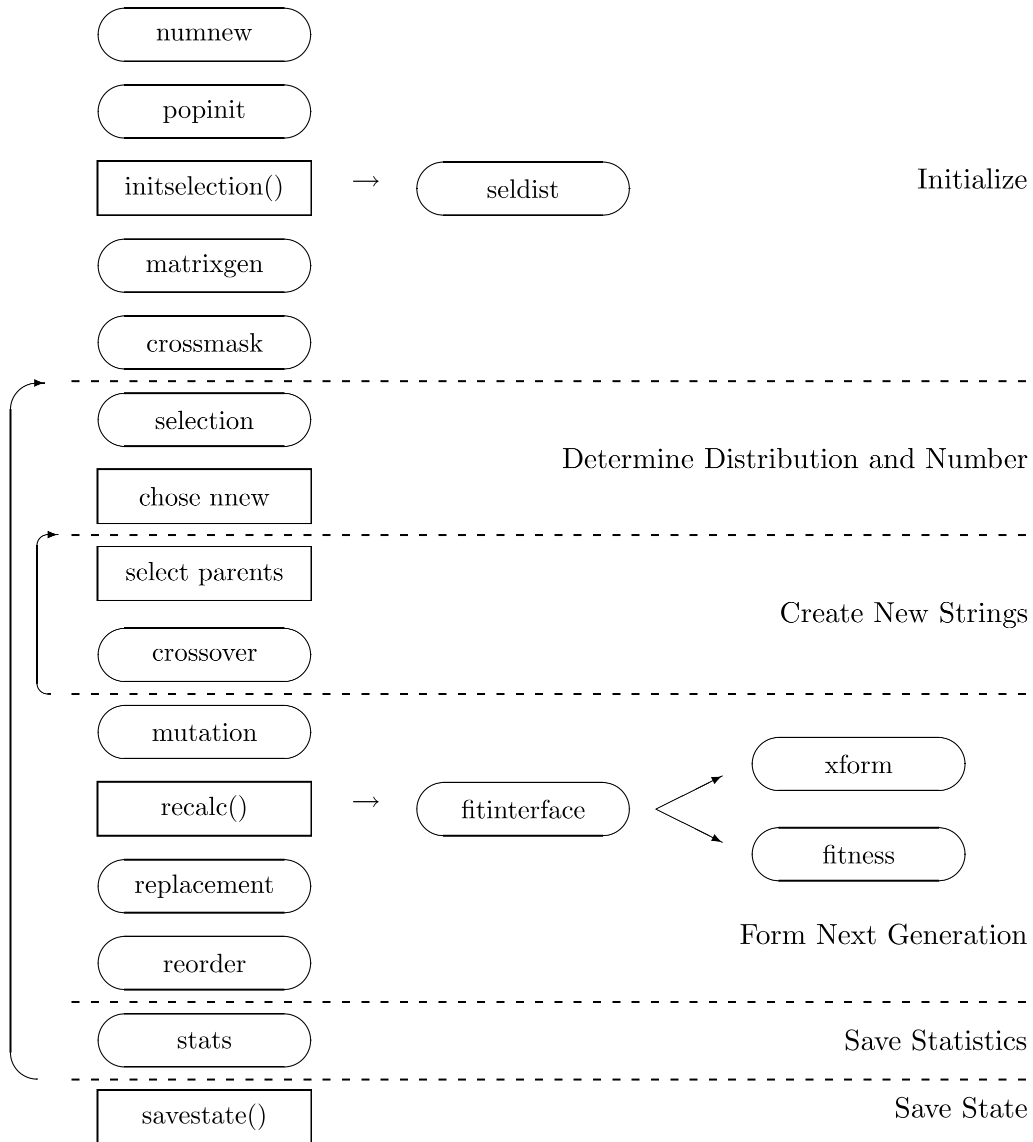

Figure 2. Structure Chart

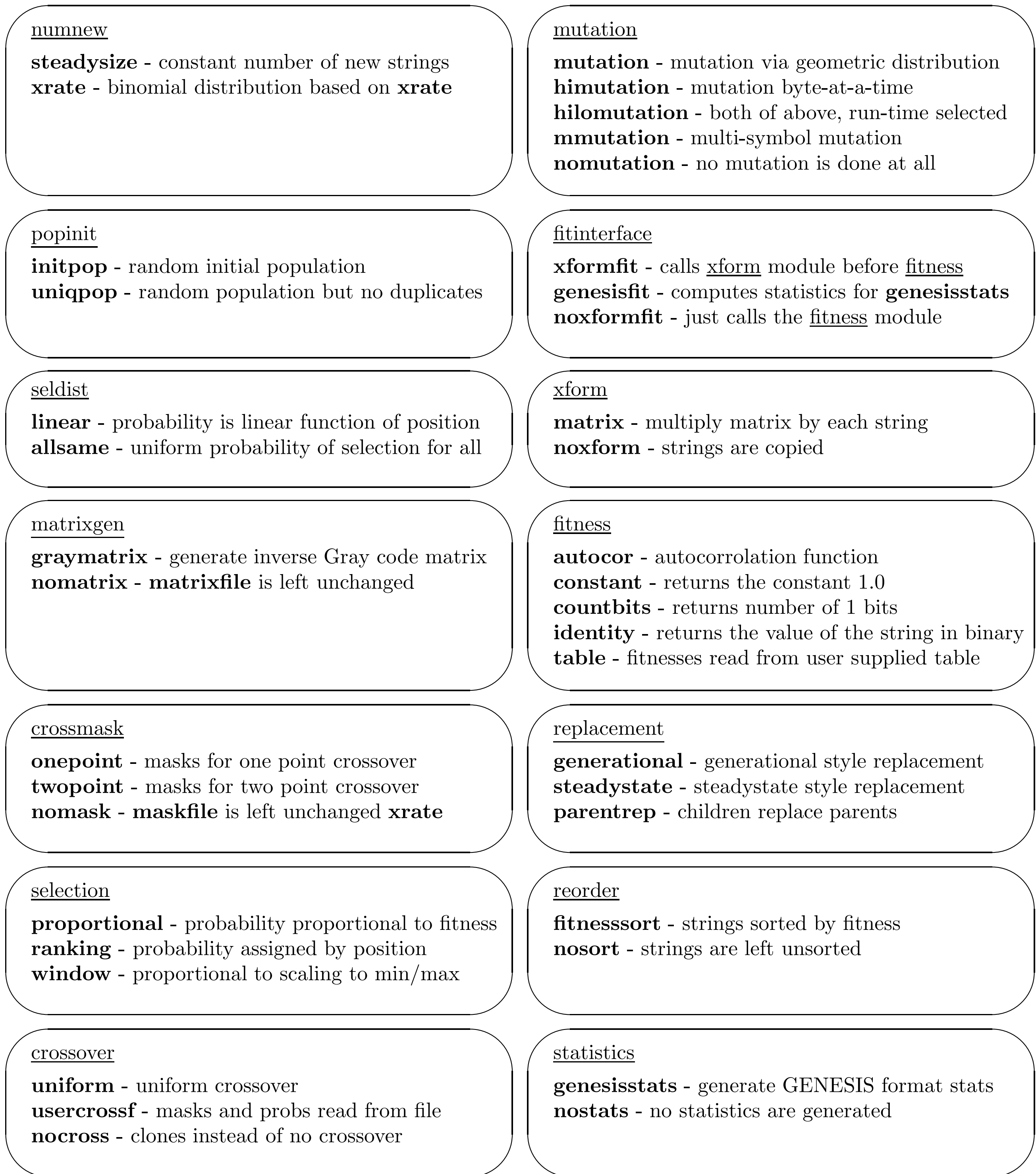

Figure 3. Module Classes and Instances

The main program first sets up a routine to trap interrupts so that the state of the system can be saved and the program suspended at the user's request. The parameter file specified on the command line or the default parameter file `generic.prm` is read in. The random number generation package is then initialized with the seed supplied in the `randseed` parameter, or the system time seed if this parameter is 0 or if it is not specified.

The first module to be called is the `numnew` module which determines the distribution of the number of new strings to be generated on each generation. This can either be a constant (`steadysize`) or a binomial distribution on the `xrate` parameter, or some user-supplied distribution.

The next module to be called is the `popinit` module. This module is responsible for setting the initial population to some reasonable starting point such as randomly generated strings or strings read from the population input file (`popinfile` parameter).

The initialization functions for the `xform` module (`initxform()`) the `fitness` function (`initfunc()`), the `fitinterface` module (`initfit()`), and the `statistics` module (`initstats()`) are called to allocate any memory or set up any tables needed by these modules. In addition, the `initxform()` function (which takes a population as an argument) will transform the initial population by the inverse of the transformation to be used during the run. This is because the `fitness` function will see strings which are transformed and it is a desirable that the strings seen by the fitness function and those on the population input and output files match.

The `selection` module's initialization function (`initselection()`) is then called. Only when `selection` is `ranking` does the `selection` module's

initialization function call the `seldist` module to determine the constant user-supplied probability distribution for selection.

This done, the fitness function is called for each string in the initial population and the results passed to the `reorder` module so that they are in the correct order to start the run.

The `matrixgen` module, if present, is called at this point to generate a matrix to be used as the transform matrix during the run.

Masks for crossover are then generated by calling the `crossmask` module. These are written to the file named in the `maskfile` parameter to be read by the `crossover` module (currently only the `usercrossf` module reads the mask file).

The population is then saved to the filename supplied in the `initpopfile` parameter and displayed on the standard output if the `displayinit` parameter is `TRUE`.

2.2.2 Determining Number and Distriubtion of Strings

Next the main loop is entered. During each iteration the selection distribution is determined by calling the `selection` module. When `selection` is `proportional` the distribution depends on the relative fitnesses of the strings involved and thus needs to be recomputed each generation.

Still within the main loop a random number is generated according to the distribution supplied by the `newsize` module. This is the number of new strings to be generated.

2.2.3 Creating New Strings

For each new string to be generated (or for every pair of children to be generated if the parameter `nchild` is 2 indicating that both children from crossover should be kept) two strings are selected by selecting a random number according to the selection distribution and using it to index into the population (which is kept in the order dictated by the `reorder` module).

These strings are crossed over by calling the `crossover` module. If `nchild` is 1 then only one of the two children is saved in the next generation, if it is 2 then both are saved. The parents of the string(s) are noted for future reference.

2.2.4 Forming the Next Generation

The loop on the number of children to generate being finished, the `mutation` module is called on the collection of new strings. The `urate` parameter is used by the `mutation` module to mutate the strings appropriately. Only when the `mutation` module is `mmutation` (multi-symbol mutation) is `alphabetsize` consulted to determine the number of bits per symbol.

Next the fitnesses of the newly generated strings are calculated. This involves calling `recalc()` which calls the `fitinterface` module for each string which differs from both its parents. When `fitinterface` is `xformfit` the `xform` module is called to transform each string prior to computing its fitness. When the `xform` module is `matrix`, then the matrix from the file named by the `matrixfile` parameter is multiplyed by each string and the

result of the multiplication is passed to the `fitness` function.

The `fitinterface` module can also be used for gathering statistics which cannot be computed in the `statistics` module. For an example of this, see the `genesisfit` and `genesisstats` modules in Appendix F.

The `replacement` module is then called to replace some of the strings from the old population with selected strings from the newly generated strings.

The resulting population is then ordered by calling the `reorder` module (and subsequently displayed depending on the value of the current generation counter and the `displayfreq` parameter).

2.2.5 Saving Statistics

The `statistics` module is called at the end of each iteration of the main loop to save statistical information about the population.

2.2.6 Saving Final State

The main loop continues until the generation counter reaches the value specified by the `endgen` parameter or until the user hits `^C`. At this time, the state of the program is saved. The population is written to the file specified by `popoutfile` and parameters necessary for restarting are written to the file named by the `paramoutfile` parameter. The final population is displayed if the value of the `displayfinal` parameter is `TRUE`. Note that when the population is written out, it is first transformed by the current

transformation. This is to maintain consistency between the strings seen by the fitness function and those kept in files.

2.3 Module Classes

Each module from within each of the collections is interchangeable with the other modules from the same collection. To build a specific configuration of the GA, the user selects one module from each collection. For example: one of the collections of modules is for replacement. Replacement modules determine which strings from the previous generation are replaced by freshly created strings. There are three replacement modules which provide generational, steadystate, and parent style replacement. In generational replacement, all the new strings are added to the next generation and the remaining slots are filled in by cloning strings chosen by the current selection module from the old population. This can be used to implement a simple generational GA by using it with a module which determines the number of new strings by using the crossover rate (`xrate`) parameter. Steadystate replacement throws out the bottom of the old population to make room for any new strings which are different from all the strings in the old population. Parent replacement causes each new string to replace its parent. Note that parent replacement is only useful when it can be guaranteed that each pair of parents produce a single pair of children, otherwise children may overwrite their parents only to be overwritten by later children.

Another example of a module collection is the population initialization modules. Initial populations may be read from a file, generated randomly,

or generated randomly without duplicates. Initial populations without duplicates are useful in conjunction with steady-state replacement which preserves the property of uniqueness among strings.

Module collections are also provided for:

- Determining the number of new strings generated each generation (which can be a constant, or a probability distribution over several possible values)
- Determining the selection method
- The probability distribution used in ranking selection
- The ordering of strings in the population
- Crossover
- Including a matrix (or other) transformation
- Mutation type
- Statistics gathering

2.4 Algorithms

2.4.1 Matrix Multiplication

Since one of the initial purposes of this system was to investigate the effects of matrix transformations on GA performance, much attention has been given to optimizing the process of multiplying a binary matrix M by a

binary string (vector) using mod 2 arithmetic. The optimization is done by initializing a two-dimensional array of strings which is $\lceil$`strlenbits`/8.0$\rceil$ by 256 and is called `xtab`. This initialization is done only once after which a fixed matrix can be multiplied by any number of strings by simply XOR'ing together $\lceil$`strlenbits`/8.0$\rceil$ items from the table.

Each entry in `xtab` is a vector of `strlenbits` bits. The table is initialized by iterating through the 8 bit blocks of each vector entry. If `strlenbits` is not a multiple of 8, then strings are padded with 0's. The variable `i` used to index through the 8 bit blocks (there are $\lceil$`strlenbits`/8.0$\rceil$ of these) is used as the first index into `xtab`.

Within this loop, each possible value for an 8 bit block is iterated through (binary 00000000 through 11111111). The value of this iteration variable `j` forms the second index of `xtab`. The values of `j` are imbedded in the `i`th 8 bit block of a separate vector which is otherwise all 0's. This vector is multiplied by the matrix M and the result is stored in `xtab[i][j]`. Once this initialization is complete, multiplication by M can be done rapidly by xor'ing together $\lceil$`strlenbits`/8.0$\rceil$ vectors from `xtab`.

To obtain the appropriate vectors from `xtab` we iterate through the 8 bit blocks of the string to be transformed using `i` as the index and access `xtab[i][j]` where `j` is the `i`th 8 bit block of the string being transformed. The xor of all these vectors forms the result of multiplying the original matrix by the string in question. A more mathematical description together with a proof is presented in Appendix B. The table `xtab` corresponds to $z_{i,u}$ in the proof. Note that because of the space taken up by the tables for this module strings of length longer than about 1000 bits shold not be used.

2.4.2 Generalized Crossover

A generalized version of crossover has been provided which allows the user wide flexibility in how crossover is done. Crossover is implemented by first creating a mask of the same length as the string. The 1 bits in the mask indicate which bits will be crossed over and the 0 bits indicate which bits will be left in place. For example: given the parent strings 01010000 and 000001010 and the mask 11110000 the first child would be 00000000 and the second child would be 01010101. Once this implementation decision is made, the only thing left to vary is the probability distribution by which masks are to be selected.

Generalized crossover allows maximum flexibility in this regard by allowing the user to supply a list of masks and associated probabilities with which they are to be chosen. It is a simple matter to implement one point and two point crossover as well as a multi-symbol version of crossover which only crosses strings between certain sized groups of bits using this method. However, because uniform crossover gives equal probability to each of a large number (exponential in the string length) of masks, it is implemented separately.

2.4.3 Multisymbol Mutation

Unlike crossover, there is little variation in the net effect of mutation in various GAs. However there is one instance where an alternative form of mutation is required. This is when multi-symbol alphabets are being

simulated with binary strings. Here, a block of bits is treated as a single symbol which can take on one of several values. However, it may be the case that the number of symbols in the target alphabet is not a power of two. In this case it must be guaranteed that mutation does not leave an invalid symbol within a string. To accomplish this, a special version of mutation is provided which treats blocks of bits as single symbols, effectively doing a Bernoulli trial for each block of bits to determine whether it will be mutated and then choosing a random symbol different from its current value and changing it to that symbol.

2.4.4 Fast Mutation

Since it is time consuming to generate random numbers it is impractical to implement mutation on binary strings in the obvious way, which is to perform a Bernoulli trial for each bit of each string in the population and toggle those for which the trial succeeds. Implementing mutation by considering each bit would require `strlenbits` random number generations every time a population member is mutated. However since this implementation was designed to be a research tool (and should therefore be consistent with other GA implementations) it is desirable that whatever realization of mutation is chosen be probabilisticly identical to the slow but sure one-Bernoulli-trial-per-bit technique.

Thus, since experience as well as a few experiments have shown that good mutation rates are fairly low (my experience has shown that a good rule of thumb is roughly .1/`strlenbits`), it would be nice if a method could allow

one to "jump over" all long runs of bits which will not be mutated. The distribution which describes the number of failures before a single success in a sequence of Bernoulli trials is called the geometric distribution. As discussed in the next section, a sample can be made from an arbitrary (finite) distribution using a single random number generation. The geometric distribution, being infinite, requires two random number generations to sample using the method described in Appendix A. Thus, if the above-mentioned rule of thumb is applied for choosing mutation rates, the number of random number generations needed to mutate a single string can be reduced from `strlenbits` to an expected value of .2.

2.4.5 Generating a Unique Collection of Strings

Sometimes, for the purpose of maximum diversity, it is desirable to have an initial population which does not contain any duplicate strings. The best way of doing this depends on the `popsize` and `strlenbits` parameters. Suppose that `popsize` were large enough that generating `popsize` strings would require generating the majority of the strings of length `strlenbits`. For example, if `popsize` were 100 and `strlenbits` were 7, then there would only be 28 strings which would not be included in the population. In this situation, we can enumerate all strings of length `strlenbits` and choose which ones to throw out. On the other hand, suppose that `popsize` is small compared to the number of strings of length `strlenbits`. For example, if `popsize` were 50 and `strlenbits` were 32. In this case we can simply generate strings randomly and check them against our list to make sure they

haven't already been generated. The probability of generating a duplicate will be fairly small since `popsize` is small compared to $2^{\texttt{strlenbits}}$. A good breakpoint for the two above methods turns out to be when the ratio of `popsize` to $2^{\texttt{strlenbits}}$ is 1/2. In this case, either method may be used without generating more than 2*`popsize` strings, on average. This is obviously true in the first method because all strings of length `strlenbits` are generated, but only when `popsize` is more than half of these strings. In the second method, note that the population, when filled, will contain less than half of all the strings of length `strlenbits`. Thus when the population is being filled and a string is generated randomly, there is never more than a 50% chance of generating a duplicate of a string already in the population. Thus, there is always at least a 50% chance of adding a generated string to the population. If the chance was exactly 50% that a generated string would be added, then the expected number of strings generated before `popsize` had been added would be $2*$`popsize`. Note, however, that collision detection requires time and may therefore increase the time complexity of the algorithm.

Although when actually using a GA to solve some real-world problem it is the case that `popsize` $\ll 2^{\texttt{strlenbits}}$, larger populations are sometimes used for research purposes, thus the two algorithms.

2.4.6 Computing Parity

For mod 2 matrix multiplication, each entry in the output vector is computed by taking the bitwise AND of a row of the matrix with the input

vector and then computing the parity of the result (which amounts to the sum of the bits in the result, mod 2). To compute parity quickly, a table is made which contains the parity of all strings of length 8. Then, since parity is just the sum of the component bits mod 2, one first XOR's together all the longwords that make up the string, then XOR's the upper and lower half of this result. The process of XOR'ing together two halves continues until the remaining chunk is smaller than or equal to 8 bits. This result is then looked up in the table.

2.5 Probability Distributions and Random Number Generation

In implementing GA's efficiently it is often necessary to use various probability distributions. Among the useful distributions are the uniform distribution, the binomial distribution, the geometric distribution and sometimes the hypergeometric distribution.

2.5.1 The Uniform Distribution

The uniform distribution is the most basic distribution and is usually provided in some form by the run-time library of the implementation language. In the case of C, the run-time library provides a function `rand()` which returns with uniform probability an integer in the range from 0 through $2^{31} - 1$. However, in generating random strings, it is useful to have a 32 bit random number generator. For this purpose a random number generator has been adapted from Knuth [2]. It is a type of additive generator

which was devised in 1958 by G. J. Mitchell and D. P. Moore [unpublished], who suggested the sequence defined by

$$X_n = (X_{n-24} + X_{n-55}) \bmod m, \quad n \geq 55,$$

where m is even, and where $X_0, \ldots, X_{54}$ are arbitrary integers and not all even. The constants 24 and 55 in this definition were not chosen at random, they are special values that have the property that the least significant bits $\langle X_n \bmod 2 \rangle$ will have a period of length $2^{55} - 1$. Therefore the sequence $\langle X_n \rangle$ must have a period at least this long. By contrast the least significant 12 bits of the C runtime library random number generators have a period of 2^{12}. Note also, that m in the above definition can be, for example, 2^{32} thus in C on a typical machine, arithmetic mod 2^{32} can be done by simply using unsigned integers without the need for an expensive mod operation. This generator also provides extra resolution and better randomness for generating the uniform distributions necessary for other purposes within the program.

2.5.2 The Binomial Distribution

The binomial distribution is useful in the efficient implementation of genetic algorithms. It can be used, for example, to decide how many strings in a population will be crossed over rather than cloned given the crossover rate. The binomial distribution arises when one cares only about how many times in a sequence of Bernoulli trials success will occur. In the case of the number of strings to be crossed over whether or not each string will

be crossed is an independent Bernoulli trial with the probability of success equal to the crossover rate. But suppose we wanted a more efficient way to decide how many strings are to be crossed. Instead of doing a Bernoulli trial for each string in the population one needs but to look at a single sample from the appropriate binomial distribution to determine how many strings to cross. The strings to cross can then be selected according to the selection distribution and the total number of times the random number generator must be called is drastically reduced, especially for low crossover rates.

Efficient calculation of the binomial distribution requires care. The probability density function (p.d.f.) of the binomial distribution contains factorials which can easily overflow beyond the capacity of integer variables if one attempts to do the calculation naively. Also, it is often the case that one wants to calculate the values of the p.d.f. for all domain elements, which could be expensive. A technique which solves both of these problems is to use incremental calculation. The p.d.f. of the binomial distribution is

$$f(x) = \frac{n!}{x!\,(n-x)!}\, p^x\, (1-p)^{n-x} \quad ,\ x = 0, 1, 2, ..., n$$

We note that $f(0) = (1-p)^n$ and that

$$\frac{f(x)}{f(x-1)} = \frac{p(n-x+1)}{x\,(1-p)}$$

so we need only initialize a variable to $f(0)$ and increment through the $x's$ repeatedly multiplying by $f(x)/f(x-1)$ in order to generate the value of f at all points from 0 through n.

2.5.3 The Geometric Distribution

The geometric distribution is useful for producing "failure gap" sizes in dealing with a sequence of Bernoulli trials (mutating a string, for example). The p.d.f. of the geometric distribution is:

$$f(x) = (1-p)^x p, \quad x = 0, 1, 2, \ldots$$

Unlike the binomial distribution, computing this p.d.f. does not present overflow problems. The problem with the geometric distribution, however, is that it is open-ended; with vanishingly small probability, a geometric random variable can take on any non-negative integer value. Since the program employs tables of probabilities for various values of random variables to facilitate rapid computation, the geometric distribution must be truncated at some point. This truncation need not prevent accurate simulation of the geometric distribution, however. An extra element is added to the table after the truncated geometric distribution which contains the remaining probability mass. In particular we compute the first n elements $(0 \ldots n-1)$ of p.d.f. shown above plus an additional element which has the value $(1-p)^n$. Now, to simulate a sample of our geometric random variable we select an integer i according to the probability in the ith position of the table. If i is not n it is the value of our random variable. Otherwise we select from the table again and sum the integers chosen. We continue in this fashion until an integer other than n is chosen. A further improvement can be made by noting that the number of times the random number generator is called

before it returns something other than n is its self a random variable and thus can be simulated with a single call. We will see that the samples from the geometric distribution needed by a GA can be simulated with exactly 2 random number generator calls (see Appendix A).

2.5.4 The Hypergeometric Distribution

The hypergeometric distribution is not used in the final version of the program. However, it was used in an earlier version and its potential for use in GAs has earned it brief mention here. The p.d.f. of the hypergeometric distribution is:

$$f(x) = \frac{\binom{s}{x}\binom{n-s}{r-x}}{\binom{n}{r}}$$

Suppose that strings were mutated with a certain probability (derived, perhaps, from the mutation rate) and crossed over with another probability. We use a binomial distribution to determine the number of strings which should be mutated. An application of a second binomial distribution would give the number of strings to be crossed over. But if we want to do both of these operations in one pass, we need to know how many strings will be both crossed and mutated. This is where the hypergeometric distribution comes in. The hypergeometric distribution would give probabilities for the number of strings which were both crossed and mutated given the number r to be mutated, the number s to be crossed, and the total population size n. To use this one would compute r and s using appropriate binomial distributions

and then compute the size of the overlap using the above distribution.

2.5.5 Random Number Generation

Vose has shown that given a vector of probabilities for a discrete distribution one may construct in linear time a table of probabilities and aliases which allow generating a sample from that discrete distribution with a single call to a random number generator which provides a uniform distribution [4]. His is a modification of a technique discovered by Walker [5] and discussed by Knuth [2]. This technique is used repeatedly throughout this GA.

2.6 Using the Program

To use the program, two files must be customized. These are the configuration file (which has an extension `.cfg`) and the parameter file (`.prm`). The configuration file determines which modules will be linked together to form your customized GA program. It consists of several lines of the form:

$$task = module$$

where *task* is one of: `popinit`, `numnew`, `selection`, `seldist`, `reorder`, `crossover`, `crossmask`, `fitinterface`, `fitness`, `replacement`, `mutation`, `statistics`, `xform`, or `matrixgen`, and *module* is a module of the appropriate type for that task. Choices for modules as well as what they do are listed as comments in `generic.cfg` (in Appendix E). Comment lines are lines which begin with the `"#"` character.

Once the `.cfg` file is configured as you want it, run the program `cfgparam` to build the GA executable. If you called your configuration file something besides `generic.cfg`, you will need to specify the name of the file on the command line with the cfgparam command. The executable will be called `genetic`.

The second file you need to configure is the parameter file. The default parameter file is called `generic.prm` (listed in Appendix E) and it contains comments describing what the various options do. The parameter file is a sequence of lines of the form

$$\mathit{option} = \mathit{value}$$

where *option* is one of: `strlenbits`, `popsize`, `startgen`, `endgen`, `nchild`, `steadysize`, `xrate`, `urate`, `popinfile`, `initpopfile`, `popoutfile`, `paramoutfile`, `matrixfile`, `maskfile`, `alphabetsize`, `displayinit`, `displayfinal`, `displayfreq`, `fittable`, `reportfile`, `reportfreq`, or `randseed`. Values can be either integer, floating point, string, or boolean depending on the parameter. Not all options are needed for a particular configuration. For example, the `alphabetsize` parameter does not need to be specified unless the `mmutation mutation` module (or some other user-created module which uses this parameter) is being used. Default values are provided for any needed parameters which are not specified.

You may also need to implement a fitness function if the fitness function you need is not provided. The table fitness module can be used for tabular fitness functions when working with fairly short strings (up to about 16 bits). To use this module simply select it in your `.cfg` file using:

```
fitness = table
```

Give the name of the file with the fitness values in your `.prm` file, for example:

```
fittable = mytable.fit
```

and in this file provide $2^{\texttt{strlenbits}}$ fitness values. These will be assigned to the string whose value when treated as a binary number gives the line number on which the fitness value appeared. Values can be any valid floating-point number, one per line. Note that when strings are printed out they are printed least significant bit first, so don't be surprised when the string 10000000 has the second number in the file associated with it.

If your strings are longer or you need to compute fitnesses dynamically for some reason, you will have to write your own fitness function. See the section in Chapter 3 describing the `fitness` modules.

2.7 Comparison with GENESIS

Since such attention has been spent on optimization detail in this GA implementation, a performance comparison with an existing system is in order. Here are some execution times on a Sun 4/480 for the autocorrelation fitness function (see Appendix F) for `genetic` (the implementation described in this thesis) and GENESIS (a popular system developed by John J. Grefenstette of the Naval Research Laboratory):

Program	Gens	Trials	Lost	Conv	Bias	Best	Average	Time
genetic	6000	588677	0	0	0.579	29	22.29	74.6s
GENESIS	5973	590049	0	0	0.536	3	9.96	161.2s

`Gens` is the number of generations or passes through the population made by each algorithm. `Trials` is the total number of fitness function evaluations. These two are slightly different because the GENESIS run length is specified in trials while `genetic` is specified in generations. Also, both programs have (slightly different) code for eliminating unnecessary fitness function evaluations. `Lost` and `Conv` give the number of bit positions which are constant (`Lost`) and more than 80% the same (`Conv`). Bias is a statistic which varies between 0 and 1 and gives a rough idea of how "converged" the population is. `Best` is the fitness of the best string found. Note that `genetic` gives 29 while GENESIS gives a 3. These are actually the same becuase GENESIS does minimization while `genetic` does maximization. Since lower is better for autocorrelation, `genetic` was given a fitness function which returned the 32 minus the autocorrelation. `Average` is the average fitness of the population at the end of the run. `Time` is the time in seconds for the test run. For this test run, the population size was 100, the string length was 32, the crossover rate was 1.0, the mutation rate was 0.1 and the report interval was roughly 100 generations (GENESIS's report interval is specified in trials; `genetic` produced 61 reports while GENESIS produced 60). The data shown above is from the last report, and the times were measured with Unix `time` command. Note that `genetic` is more than twice as fast in this particular instance. Both GA's found a string with autocorrelation 3, which is the best that exists for string length 32. Both GA's used two-point crossover.

CHAPTER 3

MODULE INTERFACES AND EXTENDING THE GA

From time to time, the ambitious user may find that the modules included do not provide all the functionality that is necessary. This may require writing a special purpose module to add functionality. This chapter describes the interfaces of all the module types in sufficient detail to allow a experienced C programmer to add a new module. This chapter may also be useful to the advanced user who desires a better understanding of the inner workings of the program. Also, adding a new parameter and a new module class are discussed.

3.1 Population Initialization Modules

Here is a sample population initialization module:

```
/*
 * initpop.c
 *
 * Module Class: popinit
 *
 * Generate population randomly or read in if popinfile set.
 */
```

```
#include <stdio.h>
#include "strings.h"
#include "population.h"
#include "globals.h"

initpop(p)
struct population *p;
{
    if(popinfile == (char *)NULL) {
        rndpop(p);
    }
    else {
        readpop(p, popinfile);
    }
}
```

The job of the population initialization module is to set the data fields of the strings which make up the population which it receives as an argument to a suitable initial value. Generally speaking, population initialization modules should call `readpop()` (or some other custom file reading module) if a `popinfile` was specified by the user. If no file name was specified, the `initpop()` module is responsible for filling in the population in some other way. This particular module calls `rndpop()` to initialize the strings of `p` to random bits.

3.2 The Number of New Strings Module

The `numnew` module is responsible for filling the probability distribution which will be used to determine the number of new strings to be formed on each generation. Here is an example module:

```
/*
 * xrate.c
 *
 * Module Class: numnew
 *
 * Determine the distribution for the number of new strings
 * each generation based on the crossover rate (xrate)
 * parameter.
 */
#include "rand.h"
#include "globals.h"
void numnew(dnnew)
struct dist *dnnew;
{
    double *nnew;
    nnew = binomial(popsize, xrate);
    dnnew->p = nnew;
    initdist(dnnew, 1.0);
}
```

This module calls `binomial()` to initialize an array (`nnew`) with probabilities from the binomial distribution. It will be an array with `popsize`+1 entries which will sum to one. This array is passed along with the distribution descriptor (`dnnew`, which is provided as parameter to the `numnew` module) to the `initdist()` function. When the time comes to determine how many strings to generate in a particular generation, a random numbers will be generated according to the distribution supplied to the `initdist()` function.

3.3 Selection Modules

Selection modules must also initialize a probability distribution. The proportional selection module is given as an example:

```
/*
 * proportional.c
 *
 * Module Class: selection
 *
 * Selection probabilities proportional to fitness.
 */
#include <stdio.h>
#include "population.h"
#include "globals.h"
#include "constants.h"
#include "rand.h"
```

```
/* array of selection probabilities for each string */
static double *selprob;

initselection(dsel)
struct dist *dsel;
{
    selprob = (double *)malloc(popsize * sizeof(double));
    dsel->p = selprob;
}

getseldist(p, dsel)
struct population *p;
struct dist *dsel;
{
    int i;
    double totfitness = 0.0;
    struct string **members = p->members;
    /* calculate probability of selection for each string */
    for(i = 0; i < popsize; i++) {
        totfitness += (selprob[i] = members[i]->fitness);
    }
    /* initialize distribution descriptor for selection this
       generation.  Note: dsel->p is filled in above */
    initdist(dsel, totfitness);
}
```

The selection module is divided into to portions. One is called at the beginning of the run (`initselection()`) and one is called each generation (`getseldist()`). The `initselection()` function should allocate any memory required and, if the distribution is constant, as is the case in ranking selection, initialize the distribution `dsel`. The `getseldist()` function is provided a population (`p`) and a proability distribution structure (`dsel`) to be filled in . This example sums the fitness fields of all the strings in the population and then assigns a probability of selection to each string which is the ratio of its fitness to the sum of the fitnesses of the entire population. Each time a string is selected for crossover by the main program, it will be selected according to this distribution. Note that distribution passed to `initdist()` need not sum to 1.0. The second parameter provided to `initdist()` should be the sum of the distribution.

3.4 The Selection Distribution Module

If the ranking selection module is used, another module (the `seldist` module) which provides a constant selection distribution must be provided. Here is an example:

```
/*
 * linear.c
 *
 * Module Class: seldist
 *
```

```
 * Generate a selection distribution which gives each slot
 * in the population probability of being chosen proportional
 * to its distance from the end of the population.
 *
 * Note: distribution returned sums to 1.0.
 */

#include <stdio.h>



double *seldist(n)
unsigned int n;
{
    double *pdf;
    double sum;
    int i;

    pdf = (double *)malloc(n * sizeof(double));
    sum = (n*(n+1))/2;
    for(i = 0; i < n; i++) {
        pdf[i] = (n - i)/sum;
    }
    return pdf;
}
```

It has a single parameter (`n`) which is usually equal to the global parameter `popsize`. It must allocate an array of double's of length `n` and return a pointer to it. Before returning, however, it should initialize the array to some probability distribution. In this case each slot is assigned a probability which is the ratio of its distance from the bottom end of the population to the sum of the integers from 1 to `n`. This distribution prefers strings near the beginning of the population to those nearer the end. `seldist` modules are called by the `initselection()` function of the `ranking` selection module and thus are called only once per run.

3.5 Crossover Modules

Many forms of crossover can be handled by the usercrossf module, however it is sometimes necessary to write a customized crossover module. The `uniform` crossover module is one example:

```
/*
 * uniform.c
 *
 * Module Class: crossover
 *
 * Implements uniform crossover.
 */
#include <stdio.h>
#include <math.h>
```

```
#include "rand.h"

#include "mask.h"

#include "population.h"

#include "globals.h"

#include "constants.h"

static struct mask *mask;

initcrossover()

{

    mask = newmask();

}

crossover(s1,s2,o1,o2)

struct string *s1, *s2, *o1, *o2;

{

    unsigned long *m, *a, *b, *ca, *cb, d1;

    int i;

    m = mask->data;

    rnddata(m);

    a = s1->data;  b = s2->data;

    ca = o1->data; cb = o2->data;

    for(i = 0; i < strlenlongs; i++) {

        d1 = (a[i] ^ b[i]) & m[i];

        ca[i] = a[i] ^ d1;

        cb[i] = b[i] ^ d1;

    }

}
```

Crossover modules also have an initialization procedure for allocating memory and other initialization. It is called `initcrossover()`. Crossover modules have 4 parameters; all 4 are strings. The first two (`s1` and `s2`) are are the input strings; the parents. The second pair (`o1` and `o2`) are the output string; the offspring. The only thing that usually differs from one type of crossover to another is the was the mask is generated. For uniform crossover, the mask is just a random binary mask (selected with uniform probability over all binary masks of length `strlenbits`).

3.6 The Crossmask Module

Often, custom crossover operators can be implemented using the `usercrossf` module. The `usercrossf` crossover module can be used in conjunction with a mask-generating module called a `crossmask` module. One example of a `crossmask` module is the one-point crossover mask generating module:

```
/*
 * onepoint.c
 *
 * Module Class: crossmask
 *
 * Generates a masks file to implement one point crossover
 * when used with the usercrossf crossover module.
 */
```

```
#include <stdio.h>

#include "strings.h"
#include "globals.h"
#include "mask.h"

onepointmask(mask, xpoint)
struct mask *mask;
{
    int i;

    cleardata(mask->data);
    for(i = xpoint; i < strlenbits; i++) {
        setstrbit(mask, i);
    }
}

genmasks()
{
    int i;
    struct mask *mask;
    FILE *fp;

    if((fp = fopen(maskfile, "w")) == (FILE *)NULL) {
        fprintf(stderr, "genmasks: can't write on maskfile");
```

```
        perror("");
        exit(1);
    }
    mask = newmask();
    fprintf(fp, "%d\n", strlenbits-1);
    for(i = 1; i < strlenbits; i++) {
        fprintf(fp, "%18.16f ", 1.0/(strlenbits-1));
        onepointmask(mask, i);
        writedata(fp, mask->data);
        fprintf(fp,"\n");
    }
    freemask(mask);
    fclose(fp);
}
```

These modules should have a function called `genmasks()` which write a file which contains the number of masks generated on the first line, followed by the masks together with their probabilities one mask per line (see file formats in Appendix D). The file should be written to the file named in the global parameter `maskfile`. The above example generates each of the possible masks for one-point crossover and writes them to a file. The macro `setstrbit()` can be used to set bits when building masks. The `writedata()` function can be used to output the masks to the file. Be sure to close the file because it will be re-opened for reading by the `usercrossf` module. The `crossmask` module is called only once per run.

3.7 The Reorder Module

When using ranking selection, it is very important to keep the strings of the population in some reasonable order, the strings nearer the beginning of the population being “better” in some sense than the strings near the end. This is accomplished by using a `reorder` module. An example is the `fitnesssort` module which sorts the population by fitness on each generation:

```
/*
 * fitnesssort.c
 *
 * Module Class: reorder
 *
 * Orders strings in population in decreasing order by
 * fitness.
 */

#include "population.h"

reorder(p)
struct population *p;
{
    sortpop(p);
}
```

The `reorder` module has a single parameter which is the population to be reordered. This module simply calls the `sortpop()` function to sort the population based on fitness. The strings of the population could be reordered in some other way (based, for example, on how well their offspring were doing) or not reordered at all when using proportional selection. See Appendix E for specific information on manipulating data structures such as the population data structure.

3.8 The Fitness Function Interface Module

The `fitinterface` module sits between the main program and the `fitness` function. It is used, for example, to call the `xform` module, if desired. If an `xform` module is not being used, then the `noxformfit` fitness interface module should be used for better execution speed. Here is an example `fitinterface` module:

```
/*
 * xformfit.c
 *
 * Module Class: fitinterface
 *
 * Calls the transformation function
 */
#include "strings.h"
#include "globals.h"
#include "constants.h"
```

```
static struct string *tmp;

initfit()
{
    tmp = newstring();
}


/*
 * fit(s)
 *
 * The program's interface to the user's fitness function.
 * String undergoes a transformation first before evaluation.
 *
 */

double fit(s)
struct string *s;
{
    double fitness();

    xform(s, tmp);
    return fitness(tmp);
}
```

Fitness interface modules have an initialization function (`initfit()`) which can be used for allocating memory or setting up tables. The main procedure of a `fitinterface` module is called `fit()` and has a single parameter which is a string. It should return a double precision floating point number which represents the fitness of that string. This module transforms its string argument (`s`) by calling the `xform` module and places the result in a temporary string (`tmp`) which it passes to the `fitness` function.

3.9 The Fitness Function

The `fitness` function is the central part of the GA. It is responsible for assigning a "fitness" value to any string that is presented as an argument. These fitnesses are used by the other modules to determine which strings survive. Here is an example fitness function:

```
/*
 * countbits.c
 *
 * Module Class: fitness
 *
 * Returns the number of 1 bits in its binary string
 * argument.
 */
#include "strings.h"
#include "globals.h"
```

```
initfunc()
{
}

double fitness(s)
struct string *s;
{

    int i, sum = 0, w;

    for(i = 0; i < strlenlongs; i++) {
        w = s->data[i];
        while(w != 0) {
            w &= (w-1);
            sum++;
        }
    }
    return (double)sum;
}
```

The first two lines should be in any fitness function. They include header files which describe the format of strings and the global variables which give information such as the string length and alphabet size. There first function defined should be called `initfunc()` and is called only once at the

beginning of the run. If needed you may put code here which initializes tables or allocates memory needed to compute your fitness function. If no such code is needed, leave the function definition empty. The second function defined should be called `fitness()` and should return a double precision floating point number. It has a single argument which is a pointer to a string structure. Strings are implemented as an array of unsigned longwords. The symbol `LONGLEN` is defined in `strings.h` and gives the number of bits in each longword; usually 32. The least-significant bit in the first longword is printed as the first bit in the string, and the most significant bit of the last longword is the last bit in the string. The global variable `strlenlongs` gives the number of longwords in the string and the global variable `strlenbits` gives the length of the string in bits. This module counts the 1 bits in its argument. The construct `w &= (w-1);` deletes the most significant 1 bit in w.

3.10 The Replacement Module

The `replacement` module is responsible for incorporating the newly generated strings into the population. It can be used as a place to filter out unwanted strings (for example duplicates, or strings with unusually low fitness) or to implement special-purpose replacement styles such as replacing parents or replacing strings of lowest fitness. Here is an example `replacement` module:

```c
/*
 * generational.c
 *
 * Module Class: replacement
 *
 * Does generational-style replacement.
 */


#include <stdio.h>
#include "strings.h"
#include "population.h"
#include "globals.h"
#include "rand.h"
#include "constants.h"


/*
 * replacement(p, newp, newsize)
 *
 * Build next generation in newp from things already there
 * (at the end) plus selected (and mutated) old strings
 * from p.
 */
```

```
replacement(p, newp, newsize)
struct population *p, *newp;
int newsize;
{
    int i;
    struct string **members = p->members,
      **nmembers = newp->members;

    /* flesh out newp with selected members from p */
    for(i = 0; i < popsize-newsize; i++) {
        copystring(nmembers[i], members[drand(dsel)]);
    }

    /* mutate the cloned old strings */
    mutation(newp, 0, popsize-newsize);

    /* and recalculate their fitness if necessary */
    recalc(newp, p, 0, popsize-newsize, CHECKFITINVALID);
}
```

The `replacement` module receives 3 parameters; the old population (`p`), the newly generated strings (`newp`) and the number of new strings (`newsize`). Note that the newly generated string are generated at the end of newp, not

the beginning. This version clones `popsize` − `newsize` selected members of the old population, mutates these using the current `mutation` module. This is the `generational` replacement module.

3.11 The Mutation Module

The mutation module is responsible for introducing random fluctuations in the population to prevent premature convergence. It uses the `urate` parameter to determine the probability with which each position in each string will be modified. Here is the default mutation module:

```
/*
 * mutation.c
 *
 * Module Class: mutation
 *
 * Use the geometric distribution to calculate gap sizes
 * between mutated strings and bits.
 */

#include "strings.h"
#include "population.h"
#include "rand.h"
#include "globals.h"
#include "constants.h"
```

```
static struct dist *dskip, *djump;

initmutation()
{
    double *nskip, *njump, probsum;
    nskip = geometric(strlenbits, urate);
    probsum = 1.0 - nskip[strlenbits];
    dskip = allocdist(strlenbits);
    dskip->p = nskip;
    initdist(dskip, probsum);
    njump = geometric(popsize, probsum);
    djump = allocdist(popsize+1);
    djump->p = njump;
    initdist(djump, 1.0);
}

/*
 * mutation(p, start, end)
 *
 * Does mutation using a geometric distribution to skip
 * over strings which are not mutated and a conditional
 * geometric distribution to find which bit(s) within a
 * selected string to mutate.
 */
```

```
void mutation(p, start, end)
struct population *p;
int end;
{
    int string = start, bit = 0;
    struct string **members = p->members;

    while(1) {
        string += drand(djump);
        bit += drand(dskip);
        if(bit >= strlenbits) {
            bit -= strlenbits;
            string++;
        }
        if(string >= end)
            break;
        togglebit(members[string], bit);
        bit++;
        members[string]->fitness = FITINVALID;
    }
}
```

Mutation takes three parameters, a population (`p`) the first string to be mutated (`start`), and the stopping point (`end`). This version accomplishes

its task by using the distribution descriptors `djump` and `dskip` to jump over strings and bits, respectively, which will not be mutated. These have been initialized to appropriate distributions for this purpose as discussed in Appendix A.

3.12 The Statistics Module

The statistics module can be used to gather whatever statistics about the population the user wishes. For example, it could be used to remember the generation numbers of generations in which an improvement was made in the best string found so far. Currently the only statistics module provided is one which mimics the statistics generated by Grefenstette's GENESIS program.

```
/*
 * genesisstats.c
 *
 * Module Class: statistics
 *
 * Generates GENESIS style static output.
 */

#include <stdio.h>
#include "constants.h"
#include "population.h"
```

```
#include "globals.h"

/*
 * genesisstats.c
 *
 * Produce GENESIS style statistics information.
 *
 */

static FILE *fp;

initstats()
{

    if((fp = fopen(reportfile, "w")) == NULL) {
        fprintf(stderr,"stats: Can't open report file %s.\n",
          reportfile);
        exit(1);
    }
}



int Lost, Conv;

double Bias;
```

```
stats(p)
struct population *p;
{
    if(!reportfile)
        return;
    if(reportfreq && (gen % reportfreq) == 0) {
        compute_stats(p);
        fprintf(fp, "%5d %5d %2d %2d %5.3f %e %e %e %e\n",
          gen, Trials, Lost, Conv, Bias, Online, Offline,
          Best, Average);
        fflush(fp);
    }
}
/************************************************************/
/*                                                          */
/* Copyright (c) 1986                                       */
/* John J. Grefenstette                                     */
/* Navy Center for Applied Research in AI                   */
/* Naval Research Laboratory                                */
/*                                                          */
/* Permission is hereby granted to copy all or any part of */
/* this program for free distribution.   The author's name */
/* and this copyright notice must be included in any copy. */
/************************************************************/
```

```
compute_stats(p)
struct population *p;
{
    register int i, j, FEW = (popsize/20),
      MANY = popsize - FEW, SOME = popsize/2;
    register int ones;
    double performance;
    struct string **members = p->members;

    Bias = 0.0;
    Lost = Conv = 0;

    for (j = 0; j < strlenbits; j++) {
        ones = 0;
        for (i=0; i < popsize; i++) {
            if (bittest(members[i], j)) ones++;
        }
        Lost += (ones == 0) || (ones == popsize);
        Conv += (ones <= FEW) || (ones >= MANY);
        Bias += (ones > SOME) ? ones : (popsize - ones);
    }
    Average = 0.0;

    for (i=0; i < popsize; i++) {
        performance = members[i]->fitness;
```

```
        if (performance > Best)
            Best = performance;
        Average += performance;
    }
    Bias /= (popsize*strlenbits);    Average /= popsize;
    Online = Onsum / Trials;  Offline = Offsum / Trials;
}
```

Statics modules take a single argument which is a population (p). They are free to access this as well as the global parameters.

3.13 The Transform Module

The transformation module is used to transform a string before it is handed to the fitness function. It takes two arguments, an input string and an output string. Currently, the only example is matrix transformation module. It is too long to quote in detail, but here is the main procedure (see Appendix F for the full text of the matrix transformation module):

```
/*
 * xform(s, tmp)
 *
 * Transform string s under global matrix M using xtab.
 */
```

```
xform(s,tmp)
struct string *s, *tmp;
{
    int i;
    unsigned char *p;
    unsigned long *tdata = tmp->data;
    register unsigned long *d1, *d, *l;

    p = (unsigned char *)s->data;   /* alias */
    copydata(tdata, xtab[0][p[0]]->data);
    l = tdata + strlenlongs;
    for(i = 1; i < ntab; i++) {
        d = tdata;
        d1 = xtab[i][p[i]]->data;
        while(d < l)
            *d++ ^= *d1++;
    }
}
```

The parameter `s` is the input string and `tmp` is the output string. This module uses the technique described in the section on Algorithms in Chapter 1 and proved in Appendix B to perform the multiplication of a matrix by the input string. The result is stored in the output string.

3.14 The Matrix Generation Module

This module can be used to generate special purpose matrices for the matrix `xform` module. Currently the only `matrixgen` module is the `graymatrix` module used to generate matrices which produce a Gray code. Here is the gray-code `matrixgen` module:

```
/*
 * graymatrix.c
 *
 * Module Class: matrixgen
 *
 * Generate a matrix which produces a Gray code
 * transformation.
 */

#include <stdio.h>
#include "matrix.h"
#include "globals.h"

genmatrix()
{
    struct matrix *m;
    FILE *fp;
```

```
    m = newmatrix();
    if((fp = fopen(matrixfile, "w")) == NULL) {
        fprintf(stderr,
          "gengray: can't open file %s for output.\n",
          matrixfile);
        exit(1);
    }
    gray(m);
    writematrix(fp, m);
    fclose(fp);
    freematrix(m);
}
/*
 * gray()
 *
 * Called by the gengray() below.  Generates a matrix with
 * the lower triangle filled with 1's.  Thus for
 * strlenbits = 5:
 *
 * 10000
 * 11000
 * 11100
 * 11110
 * 11111
 */
```

```
static gray(m)
struct matrix *m;
{
    int i,j;

    identity(m);
    for(i = 0; i < strlenbits; i++) {
        for(j = 0; j < i; j++) {
            setstrbit(m->rows[i], j);
        }
    }
}
```

It has no paramenters and is responsible for generating a matrix and writing it to the file named by the global parameter `matrixfile`. It must close the file because it is later opened for reading by the matrix transformation module.

3.15 Adding a New Parameter

From time to time, new modules may need parameters other than those that are currently supplied. For example, suppose the user wanted to implement a statistics module which saved its information to a particular file but wanted the file name to be a settable parameter. The user might want

to add a parameter to supply this file name. To do this, four things must be done. First a global variable must be defined in the file `globals.c` to be used for the new parameter. For this example the following line would do:

```
char *reportfile;
```

Next a declaration for this global variable must be placed in the file `globals.h`. The declaration must be preceeded by the keyword `extern`. For example:

```
extern char *reportfile;
```

This file (`globals.h`) is included by each module which wishes to access parameters.

Next an entry must be added in the `params` table in the source file `param.c`. The table looks like this:

```
struct param params[] = {
    { "strlenbits",   INT,    FALSE, (void *)&strlenbits   },
    { "popsize",      INT,    FALSE, (void *)&popsize      },
    { "startgen",     INT,    FALSE, (void *)&startgen     },
    { "endgen",       INT,    FALSE, (void *)&endgen       },
    { "nchild",       INT,    FALSE, (void *)&nchild       },
    { "steadysize",   INT,    FALSE, (void *)&steadysize   },
    { "xrate",        DOUBLE, FALSE, (void *)&xrate        },
    { "urate",        DOUBLE, FALSE, (void *)&urate        },
    { "popinfile",    STRING, FALSE, (void *)&popinfile    },
```

```
    { "popoutfile",   STRING, FALSE, (void *)&popoutfile   },
    { "paramoutfile", STRING, FALSE, (void *)&paramoutfile },
    { "matrixfile",   STRING, FALSE, (void *)&matrixfile   },
    { "maskfile",     STRING, FALSE, (void *)&maskfile     },
    { "initpopfile",  STRING, FALSE, (void *)&initpopfile  },
    { "alphabetsize", INT,    FALSE, (void *)&alphabetsize },
    { "displayinit",  BOOL,   FALSE, (void *)&displayinit  },
    { "displayfinal", BOOL,   FALSE, (void *)&displayfinal },
    { "displayfreq",  INT,    FALSE, (void *)&displayfreq  },
    { "fittable",     STRING, FALSE, (void *)&fittable     },
};
```

There are 4 items in each parameter table entry. The first is the name by which the parameter will be know in the `.prm` file. The second is the type of the parameter. Parameters may be either integers (`INT`), double precision floating point numbers (`DOUBLE`), strings (`STRING`) or booleans (`BOOL`). `INT`s and `DOUBLE`s are are specified in the parameter file as decimal notation. `STRING`s are specified as sequences of non-blank characters. `BOOL`s are specified as strings `TRUE` or `FALSE`. The third item is used for error checking. It is a boolean field which says whether the variable has been seen yet when the parameter file is being read. It should always be initialized to `FALSE`. The final item is the address of the global variable where the value of the parameter read from the file should be stored. For our example, one might add the following entry on the line following the entry for `fittable` above:

```
    { "reportfile",   STRING, FALSE, (void *)&reportfile   },
```

Adding this line will cause the routine which reads the parameter file at startup to recognize the parameter named `reportfile`. It is up to the individual modules which parameters they reference.

Finally the parameter must be saved by the `savestate()` function in the file `param.c`. In the case of this example, adding the following code at the end of the function would suffice:

```
if(reportfile != (char *)NULL)
    fprintf(fp, "reportfile = %s\n", reportfile);
```

3.16 Adding a New Module Class

Very occasionally the user may find that he wants to add a type of module which is not provided for in the GA framework. As an example, consider the `xform` module class which performs a transformation on the strings of the population before their fitness is evaluated. The first thing that must be done is to locate a point in the code at which a call to the newly created module class can be most easily inserted. In the case of the `xform` module, the most obvious place is the point where the `fitness` function is called; namely, the `fitinterface` module. A call to the module is inserted and a module with an appropriate interface is constructed. The new module class may require initialization. If this is the case, a call to its initialization function should be added in the main program in `genetic.c`. Finally, the program which builds the GA must be informed of the existence of new module class. In the file `cfgparam.c` is a table with a list of module classes:

```
struct param params[] = {
    { "popinit",      STRING, FALSE, (void *)&popinit      },
    { "numnew",       STRING, FALSE, (void *)&numnew       },
    { "selection",    STRING, FALSE, (void *)&selection    },
    { "seldist",      STRING, FALSE, (void *)&seldist      },
    { "crossover",    STRING, FALSE, (void *)&crossover    },
    { "crossmask",    STRING, FALSE, (void *)&crossmask    },
    { "reorder",      STRING, FALSE, (void *)&reorder      },
    { "fitinterface", STRING, FALSE, (void *)&fitinterface },
    { "fitness",      STRING, FALSE, (void *)&fitness      },
    { "replacement",  STRING, FALSE, (void *)&replacement  },
    { "mutation",     STRING, FALSE, (void *)&mutation     },
    { "statistics",   STRING, FALSE, (void *)&statistics   },
};
```

A new entry must be added for our new module. The first and last field should be the only ones which change. The last field gives the address of a `char *` variable (which should be declared before this table) to be used for temporarily storing the name of the module of that class when it is read from the `.cfg` file:

```
    { "xform",        STRING, FALSE, (void *)&xform        },
```

Finally, an entry in the `.cfg` file being used should be added which names the new module class on the left of a `"="` and a module of that class (the name of a `.c` file with the `.c` extension omitted) on the right. For example:

```
xform = matrix
```

Now simply run `cfgparam` with the new `.cfg` file as its only argument. A new copy of the GA program `genetic` will be created which includes the new module class.

CHAPTER 4

FUTURE WORK

4.1 Graphical User Interface

Although this GA implementation is very flexible, there are a lot of details which the user must be careful about lest he make careless errors which result in the GA doing something unanticipated (or, perhaps worse, something undetectably different from what was intended). To alleviate this problem a Graphical User Interface with a built-in front-end to create input files would be extremely useful.

4.2 Additional Modules

4.2.1 Multi-Symbol I/O

Although there is a multi-symbol version of mutation and a facility which could be used to construct a multi-symbol crossover (the `usercrossf crossover` module), simulating multi-symbol GA's fully would require adding a new `popinit` module to read input files with multi-character alphabets and also adding a new module class for writing population output files. A `maskgen` module which produces an appropriate `.msk` file given the values of the `alphabetsize` and `strlenbits` parameters is also needed.

4.2.2 Whitley-Style Replacement

Darrell Whitley has proposed a style of replacement which rewards strings not for their fitness, but for the fitness of their children. It works by maintaining the population in an order which is dictated by moving strings nearer the beginning of the population when they produce offspring of above average fitness and displacing the strings near the end of the population when newly generated strings are inserted. Because strings have parent fields which tell which strings in the old population produced them, a replacement module could be written which implemented this strategy.

4.2.3 Roulette-Wheel Selection

Some GA implementations, in particular GENESIS, do proportional selection in such a way the the number of offspring a particular string produces is exactly the expected number, plus or minus 1. This is known as roulette-wheel selection. Currently in this GA implementation, there is no way to accomplish this task by simply writing a new module; however a new module class could be added to allow this. The current interface is inadequate for this purpose because it assumes the existence of a probability distribution over the strings in the population which describes the probability with which each will be selected as a parent in each round of crossover.

4.2.4 Elitest Replacement Module

A replacement strategy which never replaces the n best strings is called an elitest replacement strategy. A module which implements this strategy would be useful for investigating the efficacy of this type of replacement.

4.2.5 Module to Read User-Supplied Distributions

Rather than requiring the user to provide a module which constructs a probability distribution, it would be useful to provide a module which reads a fixed probability distribution from a file. However, this module would be of limited usefulness because distributions often vary with population size, string length, and other parameters.

4.3 Improving Memory Efficiency

In designing this GA implementation, an effort was made, at least initially, to maintain consistency in data structures, often at the expense of memory efficiency. Improvement could be made in the way memory allocation is handled in the GA by removing unnecessary levels of indirection.

4.4 Saving/Restoring Random Number Generator State

When the user interrupts a run with `^C` and later restarts the run, the random number generator is currently re-initialized to the same point it

was initialized to at the beginning of the original run, rather than the to the state it was in at the time of the interruption. By adding code to save and restore all the state information maintained by the random number generator, however, it is possible to resume the random number generator at the proper point.

CHAPTER 5

CONCLUSION

I believe that this GA implementation is a useful system and at worst contains many useful ideas for both optimization and organization which can be applied to future GA implementations.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Grefenstette, J. J. [1987] *A User's Guide to GENESIS*, Navy Center for Applied Research in Artificial Intelligence, Washington, D. C.

[2] Knuth, D. E. [1981]. *The Art of Computer Programming* , Addison Wesley, Reading Massachusetts.

[3] Liepins, G. E. and M. D. Vose [1990]. "Representational Issues in Genetic Optimization," *J. Expt. Theor. Artif. Intell.* **1990:2**, 101-115.

[4] Vose, M. D. [1991]. "A Linear Algorithm for Generating Random Numbers With a Given Distribution," *IEEE Transactions on Software Engineering*, v. 17, no. 9 (1991) 972-974.

[5] Walker, A. J. [1974]. "New Fast Method for Generating Discrete Random Numbers With Arbitrary Frequency Distributions," *Electronic Letters* **10:8**, 127-128.

APPENDICES

APPENDIX A

Proof of Mutation Methods

Suppose we are interested in implementing mutation on binary strings such that the probability of mutation for each bit in every string of the population is p. One way to accomplish this would be to do a Bernoulli trial with probability of success p for each bit. However, we desire a method which will require a minimal number or calls to the random number generator. For low mutation rates we can reduce the number of calls needed by determining the size of the gap between bits which are to be changed. For the purpose of this algorithm we treat the population as one long string of bits. Consider the probability that the $(x + 1)$st bit is the first bit which will be changed. For this to happen there must be x failures followed by a success. Since all the trials are independent, the probability of this happening is $(1 - p)^x p$. This is the p.d.f. of the geometric distribution. Note that there is a non zero probability of x taking on any value, no matter how large. However, our stock random number generation technique requires that the probability for all values of the random variable be pre-computed.

We break down the geometric distribution into manageable-sized chunks by truncating the geometric distribution after the sth bit. To balance out the distribution so that it again sums to 1 we add a term which is the sum

of the probabilities of all $x \geq s$. Note that:

$$\sum_{x=s}^{\infty}(1-p)^x p = (1-p)^s \sum_{x=0}^{\infty}(1-p)^x p = (1-p)^s$$

So our extra term will be $(1-p)^s$. Now, consider what would happen if we do "chaining" by the following method: generate integers according to the truncated distribution (with the extra term) until some integer other than the one corresponding to the extra term is chosen, summing all the integers generated as we go along. The probability that the sum of all these intergers is x is:

$$((1-p)^s)^{\lfloor x/s \rfloor}((1-p)^{x \bmod s} p)$$

because the event of choosing the extra term would have to happen $\lfloor x/s \rfloor$ times followed by the choice of the (x mod s)th term. Note however that for any s:

$$((1-p)^s)^{\lfloor x/s \rfloor}((1-p)^{x \bmod s} p) = (1-p)^{s\lfloor x/s \rfloor + x \bmod s} p = (1-p)^x p$$

which is the same probability assigned by the ordinary geometric distribution. By choosing s to be a suitably large constant, we can make the probability of choosing the last slot (and therefore being forced to generate another random number) arbitrarily close to 0. Since the bits are independent, after the first bit is mutated, the same method can be used to find the size of the gap to the next mutated bit, and so on until the end of the population is reached.

A further optimization can be made by noting that for low mutation rates many samples from the truncated distribution may be required before some term other than the last is chosen. However, since the probability of choosing

the last term is by definition $(1-p)^s$ the probability that the last term will be chosen k times before something else is chosen is $(1-p)^{sk}(1-(1-p)^s)$ which is a geometric distribution. So, the number of calls to the random number generator that would have been required can be determined with a single call. However, this requires that the last call cannot choose the last slot. So, the probabilities of choosing first s slots must be normalized to sum to 1. To show that the probability that $(x+1)$st bit is the first to be mutated is correctly computed by this method, note that with the modified method, we must first generate $\lfloor x/s \rfloor$ as the number of times the last term would have been chosen with probability $(1-p)^{s\lfloor x/s \rfloor}(1-(1-p)^s)$ and then we must choose the $(x \bmod s)$th bit on the last call with probability

$$\frac{(1-p)^{x \bmod s} p}{1-(1-p)^s}$$

Multiplying these probabilities together and simplifying gives $(1-p)^x p$ which is the desired probability.

On the other hand, for high mutation rates a different approach should be taken. Note that the above method uses exactly 2 calls to the random number generator for each bit mutated. When the mutation rate is larger than about 1/16, the average number of calls to the random number generator for each byte (8 bits) in the population exceeds 1. However, with one call per byte, we can handle an arbitrarily high mutation rate as follows. Make a list of the 256 bit patterns within a byte that could be mutated. The probability of a particular pattern of bits being mutated is $(1-p)^{8-i}p^i$ where i is the number of 1 bits in the pattern. We construct a discrete distribution descriptor with these probabilities, and then with a single call

to the random number generator we can find which bits within a byte to mutate. We simply XOR this mask into the byte from the string and then continue with the next byte until the entire population has been mutated.

Note that since strings are not always a multiple of 8 bits long, we need to calculate the point along the mutation rate axis at which to change our mutation algorithm. Suppose that in the first method, we choose s to be the block size. Then the expected number of calls to the random number generator per string is $2pl$ where l is the string length. For the second method, the number per string is $\lceil l/s \rceil$. Setting these equal and solving for p yields

$$p = \frac{\lceil l/s \rceil}{2l}$$

For $s = 8$ p is approximately $1/16$ as noted above. In the `hilomutation` module, this breakpoint is calculated and the appropriate mutation algorithm selected automatically.

APPENDIX B

Proof of Matrix Multiplication Optimization

Definition 1. *The projection operator:* $\pi_i(v) = \langle v_{si}, \ldots, v_{s(i+1)-1} \rangle$ *where* $i \in \{0, \ldots, k-1\}$, $v \in \{0,1\}^n$, *and* $n = ks$ *for some* $k, s \in \{1, 2, \ldots\}$. *Thus,* $\pi_i(v)$ *is the* ith *block of* s *bits in* v.

Definition 2. *The injection operator:*

$$\iota_i(u) = \langle \overbrace{\underbrace{0, \ldots, 0}_{si\ 0's}, u_0, \ldots, u_{s-1}, 0, \ldots, 0}^{n \text{ components}} \rangle$$

where $i \in \{0, \ldots, k-1\}$ *and* $u \in \{0,1\}^s$.

Note. *We can write a vector* v *as:*

$$v = \sum_{i=0}^{k-1} \iota_i(\pi_i(v))$$

Definition 3. *For each* $i \in \{0, \ldots, k-1\}$ *and each* $u \in \{0,1\}^s$ *define* $z_{i,u} = M\iota_i(u)$ *where* M *is a* n *by* n *matrix with* $M_{ij} \in \{0,1\}$.

Theorem.

$$Mv = \sum_{i=0}^{k-1} z_{i,\pi_i(v)}$$

Proof.

$$Mv = M\sum_{i=0}^{k-1} \iota_i(\pi_i(v)) = \sum_{i=0}^{k-1} M\iota_i(\pi_i(v)) = \sum_{i=0}^{k-1} z_{i,\pi_i(v)}$$

APPENDIX C

Proof of Matrix Gray Code Theorem

Definition 1. *The* $k+1$ *by* $k+1$ *matrix* M_{k+1} *is:*

$$M_{k+1} = \left[\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 1 & & & \\ 0 & & M_k & \\ \vdots & & & \\ 0 & & & \end{array}\right] \quad ; \quad M_1 = [\,1\,]$$

Definition 2. *The* $k+1$ *by* 2^{k+1} *matrix* C_{k+1} *is:*

$$C_{k+1} = \left[\begin{array}{ccc|ccc} 0 & \cdots & 0 & 1 & \cdots & 1 \\ \hline & C_k & & & C_k & \end{array}\right] \quad ; \quad C_1 = [\,0 \quad 1\,]$$

Definition 3. *The* $k+1$ *by* 2^{k+1} *matrix* A_{k+1} *is:*

$$A_{k+1} = \left[\begin{array}{ccc|ccc} 0 & \cdots & 0 & 1 & \cdots & 1 \\ \hline & A_k & & & A_k^* & \end{array}\right] \quad ; \quad A_1 = [\,0 \quad 1\,]$$

where the "*" superscript means that the first row of the matrix to which it is attached should be complemented.

Theorem 1. M_k *is the* k *by* k *identity matrix with ones down the subdiagonal.*

Proof.

Base case $k = 1$: $M_1 = [\,1\,]$.

Induction:

$$M_{k+1} = \left[\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 1 & & & \\ 0 & & & \\ \vdots & & M_k & \\ 0 & & & \end{array}\right] = \left[\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 1 & 1 & & 0 \\ 0 & 1 & \ddots & \\ \vdots & & \ddots & 1 \\ 0 & 0 & & 1 \end{array}\right]$$

Theorem 2. *C_k has as columns the binary representation of the integers from 0 through $2^k - 1$, most significant bit at the top.*

Proof.

Base case $k = 1$: $C_1 = [\,0 \quad 1\,]$.

Induction:

$$C_{k+1} = \left[\begin{array}{ccc|ccc} 0 & \cdots & 0 & 1 & \cdots & 1 \\ \hline & C_k & & & C_k & \end{array}\right]$$

By induction, the left half of the matrix C_{k+1} is the integers from 0 to $2^k - 1$; prepending a 0 to each does not change them. Note that in the right half, prepending a 1 to each column of C_k effectively adds 2^k to the value of each. So the values of the binary numbers in the right half of C_{k+1} range from $2^k + (0)$ through $2^k + (2^k - 1) = 2^{k+1} - 1$.

Theorem 3. *$A_k = M_k C_k$.*

Proof.

Base case $k = 1$: $[\,1\,][\,0 \quad 1\,] = [\,0 \quad 1\,]$.

Induction:

$$M_{k+1}C_{k+1} = \left[\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 1 & & & \\ 0 & & & \\ \vdots & & M_k & \\ 0 & & & \end{array}\right] \left[\begin{array}{ccc|ccc} 0 & \cdots & 0 & 1 & \cdots & 1 \\ \hline & C_k & & & C_k & \end{array}\right] =$$

$$\left[\begin{array}{ccc|ccc} 0 & \cdots & 0 & 1 & \cdots & 1 \\ \hline & M_kC_k & & & (M_kC_k)^* & \end{array}\right] = \left[\begin{array}{ccc|ccc} 0 & \cdots & 0 & 1 & \cdots & 1 \\ \hline & A_k & & & (A_k)^* & \end{array}\right] = A_{k+1}.$$

In the first step the definitions of the two matrices involved have been expanded. In the second step the matrix multiplication is carried out on the

partitioned matrices. In the third step, it is noted that adding a matrix with ones in the first row and zeros elsewhere has the effect of complementing the first row of the object begin added to it. This is the definition of the "*" notation. The fourth step follows by the inductive hypothesis, and the fifth step is by definition of A_{k+1}.

Theorem 4. *Two consecutive columns of A_k differ in exactly one bit.*

Proof.

Base case $k = 1$: $A_k = [\,0 \quad 1\,]$.

Induction:

$$A_{k+1} = \left[\begin{array}{ccc|ccc} 0 & \cdots & 0 & 1 & \cdots & 1 \\ \hline & A_k & & & A_k^* & \end{array}\right]$$

The induction hypothesis covers all adjacent pairs of columns except the two middle columns. Since they clearly differ in the first bit position, they must be identical elsewhere for the theorem to be true. This amounts to showing that the last column of A_k is identical to the first column of A_k^*. The first column of any A_k is all zeros because it is the image of the first column of C_k (all zeros) under M_k by Theorem 3. Thus, the first column of any A_k^* is a 1 followed by all zeros. The last column of any A_k is the image of all ones under M_k (Theorem 3), which is a one followed by all zeros because only the first row of M_k has a single 1, all other rows have 2 ones.

Definition 4. *The $k+1$ by $k+1$ matrix N_k*

$$N_{k+1} = \left[\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 1 & & & \\ \vdots & & N_k & \\ 1 & & & \end{array}\right] \quad ; \quad N_1 = [\,1\,]$$

Definition 5. *The k by k identity matrix I_k.*

Theorem 5. $M_k N_k = I_k$.

Proof.

Base case $k = 1$: $[1][1] = [1]$.

Induction:

$$M_{k+1}N_{k+1} = \left[\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 1 & & & \\ 0 & & & \\ \vdots & & M_k & \\ 0 & & & \end{array}\right] \left[\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 1 & & & \\ \vdots & & N_k & \\ 1 & & & \end{array}\right] =$$

$$\left[\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & M_k N_k & \\ 0 & & & \end{array}\right] = \left[\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & I_k & \\ 0 & & & \end{array}\right] = I_{k+1}.$$

APPENDIX D

Manipulating Primary Data Structures

When writing new modules for the GA, one needs a clear understanding of the data structures it uses. This appendix describes in some detail the data structures themselves as well as the macros and functions which can be used to manipulate them.

D.1 Manipulating The String Data Structure

The string data structure is defined as follows:

```
struct string {
    unsigned long *data;
    double fitness;
    int parent1, parent2;
};
```

The first field is a pointer to a block of unsigned longwords which contain the string data. The second field is the fitness which is calculated by the user-supplied `fitness` function. This field should almost always be valid. Any user-written function which modifies the data field of the string should recalculate the fitness before returning. The last two fields are used to store the index of parent strings in the old population. This can be used for

parent replacement or statistics gathering. These fields are only valid for newly generated strings, not for strings that have been incorporated into the main population. Fitness fields are invalid after crossover and/or mutation until `recalc()` is called. To get at the `i`th longword of a string's data field, use the C expression `s->data[i]` where `s` is a pointer to a string structure.

It is not usually necessary to manipulate the data field of strings directly. The following macros and functions are provided which take strings as arguments and operate on their data field:

`bittest(s,b)`

The first parameter to this macro is a pointer to a string structure and the second argument is an interger. It returns `TRUE` if the `b`th bit is 1, `FALSE` otherwise.

`setstrbit(s,b)`

This macro sets the `b`th bit of the data field of string `s` to 1.

`clrstrbit(s,b)`

This macro set the `b`th bit of the data field of string `s` to 0.

`togglebit(s,b)`

This macro toggles the `b`th bit of `s`.

`cleardata(s->data)`

This function clears all the bits in strings `s` to 0.

`notdata(s->data)`

This function toggles all the bits in string `s`.

`anddata(s1->data,s2->data,dest->data)`

This function forms the bitwise and of the data field of strings `s1` and `s2` and places the result in the string `dest`.

```
xordata(s1->data,s2->data,dest->data)
```

This function forms the bitwise exclusive or of the data field of strings `s1` and `s2` and places the result in the string `dest`.

```
ordata(s1->data,s2->data,dest->data)
```

This function forms the bitwise or of the data field of strings `s1` and `s2` and places the result in the data field of string `dest`.

```
copydata(s1->data,s2->data)
```

This function copys the data from string `s2` to string `s1`.

```
rnddata(s->data)
```

This function places random bits in all the positions of the data field of string `s`.

In addition, the following functions are useful in dealing with strings:

```
struct string *newstring()
```

This function returns a pointer to a newly allocated string.

```
freestring(s)
```

This function frees all memory used by string `s`.

```
cmpstring(s1,s2)
```

This function returns a non-zero value if strings `s1` and `s2` differ, 0 if they are the same.

```
writedata(fp,s->data)
FILE *fp;
```

This function writes the string `s` to file fp using ascii “0” and “1” characters.

```
printdata(s->data)
```

This function prints string `s` to the standard output.

D.2 Manipulating the Mask Data Structure

In doing matrix multiplication, crossover, and mutation it is often convenient to deal with masks instead of strings. Masks are just like strings except that they do not have associated fitness and parent fields:

```
struct mask {
    unsigned long *data;
};
```

All of the functions and macros which operate only on the data fields of strings may also be used with masks. In addition, the function:

```
struct mask *newmask()
```

is provided for allocating new masks.

D.3 Manipulating the Population Data Structure

A population is simply an array of `popsize` strings. Its definition is very simple:

```
struct population {
    struct string **members;
};
```

You can think of it conceptually as an array of pointers to strings. An array of pointers rather than an array of strings is used because strings often need to be reordered within the population. This alleviates the need to copy around large chunks of string data, particularly for long strings. To get at the `i`th string in a population, use the C expression `p->memebers[i]` where `p` is a pointer to a population structure.

There are several functions for dealing with populations:

```
struct population *newpop()
```

Allocates memory for a new population, including memory for the strings that make it up.

```
freepop(p)
```

Frees all memory associated with population `p`.

```
rndpop(p)
```

Calls `rnddata()` on all the strings in population `p`.

```
printpop(p)
```

Prints all the strings of population `p` as well as their fitnesses to standard output.

```
writepop(p, file)
char *file;
```

Write all the strings of population p to the named file.

```
sortpop(p)
```

Sort all the strings of population `p` in decreasing order of fitness.

```
readpop(p, file)
char *file;
```

Read the strings stored on the named file into population `p`.

D.4 Manipulating Matrices

The matrix data structure is defined as follows:

```
struct matrix {
    struct string **rows;
};
```

Unlike population, matrices are square; they have `strlenbits` rows and `strlenbits` columns. To access the `i`th row of matrix `m` use the C expression `m->rows[i]`. The following functions are available for dealing with matrices from within the `matrix.c` file (which implements the matrix `xform` module):

`struct matrix *newmatrix()`

Allocates memory for a new matrix and returns a pointer to it.

`freematrix(m)`

Frees all memory associated with matrix `m`.

`identity(m)`

Turns matrix `m` into an identity matrix.

`triangular(m)`

Turns matrix `m` into a random upper triangular matrix with 1's on the main diagonal. These are useful because they are invertable.

`gray(m)`

Turns `m` into a matrix with 1's on the main- and sub-diagonal and 0's elsewhere.

`rndop(m)`

Perform a random operation on matrix `m` such as swapping two rows or XOR'ing a row with another row and replacing the second with the result. This is used in generating random invertable matrices.

`randinv(m)`

Turn `m` into a random invertable matrix.

`multiply(m,s,p)`

Multiply matrix `m` by string `s` using $\bmod 2$ arithmetic and place the product in string `p`.

`copymatrix(m1,m2)`

Copy matrix `m2` to `m1`.

`printmatrix(m)`

Print the matrix `m` to the standard output.

```
writematrix(fp,m)
FILE *fp;
```

Write the matrix `m` to the name file.

`swaprows(m,i,j)`

Swap the `i`th and `j`th rows of matrix `m`.

`findrow(m,j)`

Find an row with index `>= j` in matrix `m` with the `j`th bit set and return its index.

`changetransform(p)`

Change the current global transformation matrix used on population p from the global matrix `M` to the global matrix `M1` (which should be filled in with an invertable matrix beforehand by the user). Transform each string in population p accordingly. This involves transforming each string by the old transformation matrix (as when they are written out) and by the inverse of the new matrix (as when they are read in).

```
initxtab(m)
```

Initialize the tables used by the `xform` function to cause it to multiply strings by the matrix `m`.

```
xform(s,tmp)
```

Transform string s under global matrix M by use of info stored in xtab table.

```
readmatrix(m,file)
char *file;
```

Read the matrix stored in the named file into matrix `m`.

APPENDIX E

Sample Files

E.1 Configuration File

Here is the default configuration file. Start by making a copy of this file and modifying only the things you understand and believe you want to change.

```
#
# popinit
#
# population initialization modules
#
# initpop - initialize it to random bit strings
# uniqpop - random bits strings, no two strings will be alike
# zeropop - initialize population to all zeros, for testing
#

popinit = initpop
```

```
#
# numnew
#
# modules to determine the distribution of number of new
# strings generated per generation:
#
# xrate - distribution is based on the crossover rate
# steadysize - generate steadysize (a parameter in the .prm
#   file) new strings each generation with probability 1.0
#

numnew = xrate



#
# selection
#
# selection modules:
#
# ranking - selection probability is determined by reorder
#   and seldist modules
# proportional - selection probability is proportional to
#   string fitness
# window - modified proportional selection
#
```

```
selection = ranking


#
# seldist
#
# selection distribution modules:
#
# linear - see file linear.c
# allsame - uniform selection from population
#
# Note: Selection distribution module only needed with
# ranking selection.

# seldist = linear


#
# reorder
#
# reordering modules:
#
# fitnesssort - reorders string by fitness each generation
# nosort - doesn't reorder strings at all
```

```
#

reorder = nosort


#
# crossover
#
# crossover implementation modules:
#
# uniform - does uniform crossover
# usercrossf - uses masks from maskfile parameter to do
#   crossover
# nocross - just copys parents; for testing
#

crossover = usercrossf


#
# crossmask
#
# crossover mask generation modules, writes mask file to the
# file name provided in the maskfile parameter.
#
```

```
# onepoint - generate masks and probabilities for one point
#   crossover
# twopoint - generate masks and probabilities for two point
#  crossover
# nomask - use when crossover is not usercrossf, or when
#  using your own masks
#

crossmask = twopoint



#
# fitinterface
#
# fitness interface modules:
#
# xformfit - calls the xform module on each string before
#   evaluating fitness
# noxformfit - doesn't call any transformation
# genesisfit - calculate parameters needed to produce
#   GENESIS-style statistics
#

fitinterface = noxformfit
```

```
#
# fitness
#
# fitness functions:
#
# countbits - counts the number of 1 bits in the string
# autocor - return autocorrelation of string
# table - return an entry from a table
# identity - return the binary value of the string (len<32)
# integer - same as above, unlimited length strings
# constant - return the constant 0 for every string
#

fitness = countbits



#
# replacement
#
# replacement modules:
#
# generational - all new strings are added, selected old
#   strings fill in
# steadystate - newly generated strings which differ from old
```

```
#   strings are added
# parentrep - each string replaces its parent
# fillin - fill in next generation with first stings from old
#

replacement = generational



#
# mutation
#
# mutation modules:
#
# mutation - normal mutation, for urate < ~1/16
# himutation - special module for use when urate > ~1/16
# hilomutation - run-time switched between above 2
# mmutation - multi-symbol version of mutation
# nomutation - does no mutation at all
#

mutation = mutation



#
# statistics
```

```
#
# statistics gathering modules:
#
# genesisstats - saves genesis-style statistics, use
#   fitinterface = genesisfit
# nostats - does nothing, use when no other statistics
#   module is used
#

statistics = nostats



#
# xform
#
# Population transform module.
#
# matrix - multiplies a matrix (uses parameter matrixfile)
# noxform - identity transformation
#

xform = noxform


#
# matrixgen
```

```
#
# Only useful with the matrix population transformation
# module. If you have code that generates your transformation
# matrix for you, you can link it into the GA here.  See
# graymatrix.c for an example.  This code is called at the
# beginning of the run and is expected to output an
# appropriately sized matrix in the file named by in the
# matrixfile parameter.
#
# graymatrix - generate a matrix which produces a gray-code
#   transformation
# nomatrixgen - don't generate a matrix, use existing
#   matrixfile.
#

matrixgen = nomatrixgen
```

E.2 Paramater File

Here is the default parameter file. Again, start with a copy and change the thing you need to be different.

```
#
# Lines which begin with # in the left-most column are
# comments.
#
```

```
#
# This file (paramdefs.prm) is read by the GA when no other
# parameter file is specified.  To specify another parameter
# file, use:
#
# genetic filename.prm
#

#
# strlenbits
#
# strlenbits is the length each string will be, measured in
# bits
#

strlenbits = 32

#
# popsize
#
# popsize is the number of strings in the population
#

popsize = 100
```

```
#
# startgen
#
# startgen is the generation number to begin the GA, mostly
#  used for restarting
#

startgen = 0



#
# endgen
#
# endgen is the number of the first generation which will
# not be done.  If startgen is 0, then endgen is the number
# of generations which will be done.
#

endgen = 1000



#
# nchild
#
```

```
# nchild is the number of children to keep from each
# crossover.  It must be either 1 or 2.
#

nchild = 2


#
# xrate
#
# xrate is the crossover rate.  it is used by certain
# configurations of the GA to determine what portion of the
# next generation will be the result of crossover.  It must
# be between 0.0 and 1.0.
#

xrate = 1.0

#
# urate
#
# urate is the mutation rate. It is (usually) the probability
# that each bit in the population will be mutated each
# generation.  For steady state, only the newly generated
# string(s) will be mutated.  For multi-symbol mutation
```

```
# mode, it is the probability that each symbol will be
# mutated.
#
# If you are unsure what you want the mutation rate to be,
# a good rule of thumb is (0.1 / strlenbits).
#

urate = 0.003125



#
# steadysize
#
# steadysize is the number of strings to be generated each
# generation when the GA is in steady-state mode.  It is
# ignored in generational mode.
#

# steadysize = 1



#
# popinfile
#
# popinfile is the file from which the initial population
```

```
# will be taken. The format is one string per line, 0's and
# 1's only. The length of each line must match the specified
# strlenbits and the number of lines must agree with the
# specified popsize or warnings will result. If the popinfile
# parameter is not specified, or is commented out with a the
# the initial population will be generated randomly.
#

# popinfile = paramdefs.pop


#
# initpopfile
#
# initpopfile is the name of a file where the initial
# population should be written for later reference.  This is
# only useful is the population was randomly generated rather
# than read from a file.
#

initpopfile = paramdefs.pop


#
# popoutfile
```

```
#
# popoutfile is the file where the population will be written
# at the end of the run, or if the user interrupts the run
# with ^C.
#

popoutfile = paramdefs0.pop



#
# paramoutfile
#
# paramoutfile is the place to write parameters for a
# continuation run when the user hits ^C
#

paramoutfile = paramdefs0.prm



#
# matrixfile
#
# matrixfile specifies a file containing an invertible binary
# matrix consisting of strlenbits lines and strlenbits
# columns of 0's and 1's with no spaces.  The inverse of this
```

```
# matrix is used to transform each population string on
# input and the matrix itself is used to transform
# each population string before being passed to the fitness
# function or being written to file or standard output.
# Commenting out or omitting the matrixfile parameter will
# result in no matrix transformation being done.  If you
# aren't going to be using matrixfile, you should use the
# noxformfit module to save execution time.
#

# matrixfile = paramdefs.mat



#
# maskfile
#
# maskfile is only used by the usercrossf module.  It is
# the name of a file containing a list of masks to be used
# by crossover.  The first line must contain the number of
# masks provided on the rest of the file. The masks should be
# one per line preceded by a probability.  The probabilities
# should sum to one.  Whenever crossover is done, a mask
# is selected according to the specified probabilities.  For
# each of the two strings involved in the crossover, the
# positions corresponding to the 1 bits from the mask are
```

```
# switched and the resulting two strings are returned as
# the result of the crossover. See paramdefs.msk for an
# example. Note: this file will be overwritten when using
# mask generating functions such as onepoint and twopoint.
#

maskfile = paramdefs.msk



#
# alphabetsize
#
# alphabet size is a parameter used by the mmutation
# (multi-symbol mutation) module. It is the number of
# distinct symbols which should be representable by a single
# position in the string. For alphabetsize = 2, mmutation
# behaves exactly like ordinary mutation, but slower.
#

alphabetsize = 2



#
# displayinit
#
```

```
# displayinit is a boolean flag specifying whether the
# initial population and associated fitnesses should be
# echoed to standard output at startup.
#

displayinit = FALSE



#
# displayfinal
#
# displayfinal is a boolean flag specifying whether the
# final population and associated fitnesses should be printed
# to standard output at termination.
#

displayfinal = TRUE



#
# displayfreq determines the interval (in generations)
# between printings of the population and fitnesses to
# standard output during the run.  If displayfreq
# is 0, no printings will be done.
#
```

```
displayfreq = 0


#
# fittable
#
# This parameter is used with the table fitness module.  It
# is the name of a file containing a list of 2**strlenbits
# floating point numbers.  These will be assigned as the
# fitnesses for the string corresponding to the binary
# representation of the number's position in the file (line
# number).  Example: if strlenbits = 2 and f(00) = 0,
# f(10) = 1, f(01) = 2 and f(11) = 3 then the file would be:
# 0
# 1
# 2
# 3
#
# Remember that when strings are printed out (as well as in
# description above) they are printed least significant bit
# *first*.
#

# fittable = twobit.fit
```

```
#
# reportfile
#
# Some stats modules (such as genesisstats) look at this
# parameter to determine where to write their output.
#

reportfile = out



#
# reportfreq
#
# The frequency with which reports are written to the above
# file, in generations.  It is left up to the stats module
# whether to pay attention to this parameter (genesisstats
# uses it).
#

reportfreq = 100
```

E.3 Population File

A population input file is just a list of fixed length binary strings. The length of each line should match the `strlenbits` parameter and the number of lines in the file should match the `popsize` parameter. This example works for `popsize=10` and `strlenbits=32`.

```
01100110001101011010101010100101
01011000011110011101010101000011
10010110010010111101010101000110
01100110010011110101010101001001
01101001001100111101010001011001
11001001011010110101010001101010
01101001001110110100010101011001
11001001001110011101010101001010
01001110010010111101010101000011
01101001001110100101010111000011
```

Strings on population input files are least significant bit first.

E.4 Mask File

The mask file provides not only masks for use in crossover but also the probability with which each mask will be used at each crossover operation. The first line of the file is the number of masks, then the masks one per line preceeded by their probability. The probabilities should sum to as

nearly floating point 1.0 as possible. However only a warning will be issued if they do not. To use a crossover mask file you must have `crossover = usercrossf` in your configuration (`.cfg`) file. The following file implements one point crossover on strings with two-bit-long blocks.

```
16
0.0625 00000000000000000000000000000011
0.0625 00000000000000000000000000001111
0.0625 00000000000000000000000000111111
0.0625 00000000000000000000000011111111
0.0625 00000000000000000000001111111111
0.0625 00000000000000000000111111111111
0.0625 00000000000000000011111111111111
0.0625 00000000000000001111111111111111
0.0625 00000000000000111111111111111111
0.0625 00000000000011111111111111111111
0.0625 00000000001111111111111111111111
0.0625 00000000111111111111111111111111
0.0625 00000011111111111111111111111111
0.0625 00001111111111111111111111111111
0.0625 00111111111111111111111111111111
0.0625 11111111111111111111111111111111
```

E.5 Matrix File

To use a matrix file, you must have the following in your `.cfg` file:

- `fitinterface=matrixfit`,
- `xform=matrix`
- `matrixgen = nomatrixgen` (unless you want a Gray code matrix)

Matrices are binary, square, and are `strlenbits` in each dimension. Each string is multiplied by the matrix before it is passed to the fitness function. The multiplication is done using mod 2 arithmetic. When a population is read in, each string is transformed with the inverse of the matrix, and when written out each string is transformed by the matrix itself. This allows successive runs to use different matrices seamlessly and also makes population files easier for users to read because the population files are always the same as what the fitness function sees. The matrix must have an inverse to be useful as a transformation matrix. Here is an example matrix for use when `strlenbits=10`.

```
1001010101
0110111010
0010101011
0001101010
0000101101
0000010110
0000001101
0000000110
0000000011
0000000001
```

E.6 Fitness Table

Fitness tables are useful when doing experiments with short strings and fitness functions which can be listed exhaustively. They are files which contain one floating point value per line and should have $2^{\texttt{strlenbits}}$ lines. Here is an example for two bit strings which assigns each string a fitness which is the number of 1 bits in that string:

```
0
1
1
2
```

E.7 GENESIS-Style Statistics Output File

Here is an example file produced by the `genesisstats` `statistics` module:

```
  0    200 0 0 0.552 -9.770e00 -5.125e00 -5.000e00 -1.009e01
100  10010 0 0 0.579 -9.585e00 -4.037e00 -4.000e00 -9.400e00
200  19797 0 0 0.568 -9.550e00 -4.019e00 -4.000e00 -9.140e00
300  29593 0 0 0.577 -9.545e00 -4.012e00 -4.000e00 -9.690e00
400  39375 0 0 0.598 -9.534e00 -4.009e00 -4.000e00 -9.270e00
500  49177 0 0 0.608 -9.531e00 -4.007e00 -4.000e00 -9.330e00
600  58994 0 0 0.587 -9.542e00 -4.006e00 -4.000e00 -9.580e00
700  68824 0 0 0.598 -9.537e00 -4.005e00 -4.000e00 -9.370e00
```

```
 800 78622 0 0 0.587 -9.535e00 -4.004e00 -4.000e00 -9.040e00
 900 88423 0 0 0.586 -9.532e00 -4.004e00 -4.000e00 -9.880e00
1000 98231 0 0 0.593 -9.540e00 -4.003e00 -4.000e00 -9.370e00
```

The first column is the generation number at which the report line was produced. The secon column is the number of fitness function evaluations which had been performed at that point in time. The third and fourth columns are the number of "Lost" and "Converged" bits. Lost bits are bit positions within the strings which are the same in every population member. Converged bits are bit positions in the strings which are the same in 80% or more of the population. The fifth column is called the "Bias" of the population. It is the sum over the bit positions of the number of strings which have the same bit in that position as the majority of the population, normalized to be between 0 and 1 by dividing by `popsize*strlenbits`. The next 4 columns are the Online, Offline, Best, and current Average performance. Online performance is defined as the average fitness of strings evaluated by the fitness function. Offline performance is the running average of the fitness of the Best string found so far, updated every time a string is evaluated. Best is the fitness of the best string seen during the run so far, and Average is the average fitness of the strings in the population at the time the report is generated.