University of Alberta

GIGA Program Description and Operation

by

Joseph Culberson

Technical Report TR 92–06
June 1992

DEPARTMENT OF COMPUTING SCIENCE
The University of Alberta
Edmonton, Alberta, Canada

GIGA Program Description and Operation

Joseph C. Culberson [*] [†]

June, 1992

Abstract

This document describes the gene invariant genetic algorithm (GIGA) program. This program represents a unique approach to designing GAs with many interesting results. The primary distinguishing feature is that when a pair of offspring are created and chosen as worthy of membership in the population, they replace their parents. In the absence of mutation, this has the effect of maintaining the original genetic material over time, although it is reorganized, and hence the "invariant" in the name.

The source code for the GIGA program, written in the programming language C, is available on line. This document explains how to use this program and describes the inputs. It also discusses the design philosophy and indicates several possibilities for future extensions and variations.

1 Foreword

This program is a prototype, and is not intended for any commercial application. It consists of C source code, and has been designed to run under UNIX, although it should be easy to port it to other environments. Its primary purpose is to allow readers of [2] to perform variations and extensions to the experiments cited there.

To obtain the program source and this document ftp thorhild.cs.ualberta.ca and cd to directory pub/GIGA. The README file will contain information on obtaining these and the following related documents.

[*] Supported by Natural Sciences and Engineering Research Council Grant No. OGP8053. Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, T6G 2H1. email:joe@cs.ualberta.ca

[†] This paper is available via ftp thorhild.cs.ualberta.ca

An accompanying paper [2] describes some of the interesting results and contrasts this approach with traditional genetic algorithms (TGAs). The paper is written to be read in conjunction with the experiments available with this program, and is designed to be used in an interactive fashion.

Lewchuk's master's thesis [11] presents research results on a special case of GIGA not currently available in this program. It is a particularly interesting approach in that nowhere does it explicitly select for or bias toward superior individuals, but it nevertheless is effective on several functions.

The intent of this program, documentation and the accompanying papers [2, 11] is to convince the reader that GIGA is a neat idea and deserves exploration. It is hoped the reader will find many new intriguing mysteries concerning evolutionary processes and discover that this is a unique tool that may help achieve some insight.

The program and this document should be considered preliminary and non-static. I hope to update both as new ideas and old bugs come to light.[1] To this end I welcome suggestions for improvements and extensions, and look forward to many controversial comments both pro and con.

In section 2 a quick and superficial over view of GAs is presented, followed by a short summary of the basic design principles of GIGA and how it contrasts with the GA. Section 4 explains the operation of the program, and its inputs. Section 5 discusses some of the design philosophies and lists various improvements and features that might be tried in future versions.

2 Introduction

For a thorough introduction to genetic algorithms the reader is referred to [7, 9, 4, 3]. Here we outline the basic features of genetic algorithms (GAs) for purposes of comparison to and description of gene invariant genetic algorithms(GIGAs).

A genetic algorithm maintains a *population* of n strings, called *individuals* or *members*. Each string is of length l and is drawn from a set of α characters, called the *alphabet*. Often the alphabet is binary, that is $\alpha = 2$, but larger character sets are allowed, and in some applications [4] the alphabet is the set of real numbers, and thus not even finite. The population varies over time under the control of the genetic algorithm as members are replaced by new strings. These new strings are generated from previous members through the action of various genetic operators to be discussed shortly.

[1] Why are both bugs and ideas attracted to light?

The members of a population are evaluated by the *environment*, which just means that the program has some means of assigning a value to a string. These values may be modified in some way, through *scaling* for example[7]. This modified value is referred to as the *fitness* of the string.

We designate the population by the matrix notation P_{ij}^t, which refers to the jth character of the ith member of the population at time step t. To refer to a particular member as a whole, we drop the second subscript, and when no confusion arises we drop the superscript indicating time.

The genetic operator most often used to produce new strings is *crossover*. Two members of the population $\mathbf{P}_1$ and $\mathbf{P}_2$ (called the *parents*) are selected, and a pair (usually) of new strings C_1 and C_2 (called the *children*) is formed in which for each j either

$$C_{1j} = \mathbf{P}_{1j} \text{ and } C_{2j} = \mathbf{P}_{2j}$$

or

$$C_{1j} = \mathbf{P}_{2j} \text{ and } C_{2j} = \mathbf{P}_{1j}$$

That is, the characters of the children are the same as those of the parents, but may be switched between them. Various crossover types are described in the literature. In *one point* crossover an integer k, $1 < k < l$ is chosen, and for $j < k$ the first condition holds, while for $j \geq k$ the second condition holds. In *multipoint* crossover a number of such points are chosen and the intervals alternate between the two conditions. In *uniform* crossover, the choice between the two conditions is made independently character by character with some fixed probability p.

Other genetic operators include transpositions, reordering, reversal and mutation[7]. These are not currently available in the program described in this paper.

Gene invariant genetic algorithms (GIGAs), the subject of this paper, form a subclass of genetic algorithms, principally distinguished by the notion that the total genetic makeup of the population does not change with time. In particular, the multiset of characters of any column of the population $\mathbf{P}_{*j}^t$ *does not change with time.*[2]

To maintain this invariance, it is only necessary that a pair of children produced by the crossover operation replace their parents in the population. The invariance rule then follows trivially provided there is no mutation or other genetic operator in use.

[2] We will undoubtedly wish to relax this rule in future research, for example by adding operators such as mutation and transposition. But the current program enforces this rule.

In the remainder of this section, we will look at some of the design ideas in the accompanying program.

A *family* is a set of pairs produced by a set of crossover operations performed on a single pair of parents. In GIGA when a pair of parents is selected, they are used to generate a family. The best pair is selected from the family and replaces the parents. If *elitism* is invoked, the selection of the best pair includes the parents as part of the family. Many different notions of what constitutes a "best" pair can be defined. Several are implemented in the program. The sequence of a selection, production of a family and replacement of parents is called a *mating cycle* or *mating*.

For many problems, crossing parents which differ widely in value is likely to produce offspring of intermediate value, and so no progress will be made either in the minimum or maximum values. This observation is based on the assumption that similar values are reflective of string similarities, an assumption we must make if there is to be any use made of crossover.[3] Assuming we want to maximize or minimize some function, we are more likely to make local progress in terms of increased fitness if we mate strings of similar value. At the other extreme, crossing identical strings, or strings with Hamming distance less than two will produce strings identical to their parents. The approaches examined by Lewchuk [11] can be seen as taking these arguments to the extreme. He only allows the pair closest in value to mate, with some restrictions to eliminate mating of overly similar individuals.

In this implementation, we always select parents in adjacent rows of the population. When replacing parents, we always put the child with the larger value in the row of higher index. The effect, for well behaved functions, is that the population will become (nearly) sorted by value. The efficacy of the sorting (and consequently the search) will depend on the selection criteria for the parents. Several mechanisms are available in the program, others are suggested in section 5 and readers are encouraged to develop their own. In future versions we may wish to allow mating between non-adjacent individuals, perhaps with a probability inversely proportional to the distance between them in the population.

To further improve the efficiency of the program, the user can specify that the population be pre-sorted and maintained in sorted order. The user can also select various crossover operators and parameters, as well as other

[3] In light of the results in [2] similarities may mean the absence of certain characters or patterns, not just the presence of them. This discussion is necessarily vague until we can determine formal and encompassing definitions for these concepts.

specifications. A full discussion of these options is presented in section 4.

The GIGA algorithm is outlined in figure 1.

Initialize population $\mathbf{P}^0$.
For some number of matings do begin
    Select a pair of parents for mating.
    Produce a family of offspring pairs using crossover.
    Select the best offspring pair.
    Replace the parents with the best pair.
    Adjust the population (e.g. sort) if requested.
end.

Figure 1: Outline of GIGA Program

3 Starting Up

This program is designed to be run under UNIX and has been tested primarily on SUN workstations. The program and sample inputs are packaged as a compressed shar file, created under UNIX by the shell command **shar**. The user should uncompress the file, then use the shell command **unshar** to obtain the source code and experiment input files. Directories **Source** and **Experiments** will be created for these files. **Experiments** contains subdirectories **Simple, Decept** and **DeJong** which contain the files from the three experimental sections of [2]. In **Source** there is a **makefile**. Issuing the command **make** will create an executable file called **giga**.

All input to the program is taken from the standard input, and all output is to the standard output. These features obviously require improvement, and users are encouraged to modify the program to suit their needs. The current program is intended primarily for interactive use.

The various input files contain settings for experiments described in [2]. For example, **goldberg.1** contains the input for an experiment on Goldberg's 3-bit deception. To run this experiment simply type

```
giga <goldberg.1
```

You may wish to pipe the output through **more**, or redirect it to an output file for later editing as sometimes the quantity of data generated can be quite large.

4 Program Inputs

In this section the input parameters required for the operation of this program are explained. Further discussion and possibilities for future improvements are presented in the final section.

Random Seed

This is required to initialize the random number generator. The default random number generator used is the one available in the C library under UNIX. Use of a different generator may cause some experiments to behave differently than described.

Population Parameters

GIGAs, like GAs, use a population of strings. This program allows the strings to be defined over an alphabet with up to 256 different characters in it. The following parameters allow the user to choose the characteristics of the population.

Population Size An integer (n) specifies the number of strings in the population. The population size remains fixed throughout the program run. It must be at least 2 and no more than MAXPOP.[4]

String Size An integer (l) specifies the number of characters in the strings with an upper limit of MAXSTRING. All strings are of this fixed length.

Alphabet Size An integer (α) determines the number of characters in the alphabet. This must be at least 2 and no more than 256. Populations may be printed only if the alphabet has fewer than 89 characters. The characters are stored internally as byte length integers ranging from 0 to $\alpha - 1$.

[4]Terms defined in capitals can be found in the file giga.h

Initialization A choice of only 1 or 2, this determines how the population is to be initialized. The first method simply generates n strings at random; that is, each character of each string is chosen at random.

The second method generates *sets* of α strings at a time. The first (i.e. zeroth) string of a set is chosen at random. The remaining strings of the set are produced in rotation from the zeroth string; specifically

$$\mathbf{P}^0_{ij} = (\mathbf{P}^0_{0j} + i) \bmod \alpha$$

This ensures that each character is (as nearly as possible for given α and n) equally represented in each column of the population. This is very important for GIGAs, and *is the recommended method for initializing the population.*

Keep Sorted An input of 'y' will ensure that after each replacement of a parent pair, the population will be sorted by value. This usually improves the performance. This also implies the initial population is sorted before any mating takes place.

Initial Sort If 'n' was the response to the previous parameter, then the user still has the option of sorting the initial population.

Mating Parameters

Mating parameters control how many pairs of children should be generated and which pair of children should replace the parents, if any.

Elitism Replying 'y' will cause the program to use elitism when deciding whether any pair of the family should replace the parents. If the parents are judged better than the best pair of children, no replacement will occur. If 'n' is entered, then the best pair of children will always replace the parents. This allows for greater stochastic behavior in the exploration, but usually performance is somewhat inhibited. A possible exception occurs with De Jong function f_2 as discussed in [2].

Equality Epsilon A real value used by GIGA to test the values of the parents to see if the parents might be equivalent. If the absolute difference is no greater than this epsilon value, then GIGA tests whether the parents are equal strings. If the strings are equal, then no crossover operations are performed, and the remainder of the mating cycle is aborted. The only effect is one of program efficiency, since no new strings can be produced by crossing identical parents.[5] If a negative value is entered, then no equivalency test will be performed. Usually a value of 0 (zero) is sufficient, but for noisy functions such as De Jong's function number 4, a higher value might be useful. Note that the equality test itself takes time proportional to string length l, so epsilon should usually be small.

Family Size An integer(s) controls the number of pairs of children produced from a single pair of parents. Notice that the number of evaluations done by the program is $n + 2sm$ where m is the number of matings not aborted due to equality considerations.

Definition of Best An integer 1 to 4 that selects one of the four definitions of best pair. If elitism is activated, then the definition is applied to the parents as a pair also, and if the parents are best, then no replacement occurs. In the remainder of this item, "replacement" is deemed to include this option.

Option 1 selects the pair in which the maximum value of the pair is largest over all pairs. This pair replaces the parents. This is most effective when we wish to maximize a function.

Option 2 selects the pair in which the minimum value of the pair is smallest over all pairs. This pair replaces the parents. This is most effective when we wish to minimize a function.

Option 3 selects the pair in which the absolute difference of the values of the pair is maximum over all of the pairs. This pair replaces the parents. This treats maximization and minimization symmetrically. The idea behind using this option is to encourage the population to diverge as much as possible.

Option 4 selects the pair in which the absolute difference of the values of the pair is minimum over all of the pairs. This pair replaces the parents. This also treats maximization and minimization symmetrically. However, it is unlikely to optimize anything, as the name SQUISH implies. The population is encouraged to converge in value as much as possible. This option is supplied mostly because of its symmetry to option 3, and because it might be interesting for some functions to see how similar the values can be made under the condition of enforced invariance.

[5] If mutation is added, then invoking this test could affect program results.

Crossover Parameters

The crossover types allowed in this program are one point, uniform and a mixture of uniform and multipoint, with various parameters controlling rates and probabilities.

Crossover Type An integer 1 to 4 selects the desired crossover type.

Option 1 selects one point crossover as described in section 2. Each pair of the family is generated by an independently chosen crossover point.

Option 2 selects uniform crossover as described in section 2. An additional parameter, Swap Bias will be requested when this option is selected. This value should be an integer between 0 and 100. It represents the probability (expressed as a percentage) of the character in the jth position of child 1 coming from parent 1. The further this value is from 50, the more similar the offspring will be to their parents. A value of 50 maximizes the shuffling of the two strings during crossover. It makes every string in the Hamming closure of the parents equally probable as a child.[6]

Option 3 selects one point crossover as in option 1, but guarantees that each pair of offspring in a family will be produced using a different crossover point. By setting $s = l - 1$, and using elitism, a deterministic hill climbing search will be created (in crossover space, not Hamming space).[7]

Option 4 allows for a mixed set of crossover operations. If this option is selected, then the user will be asked for a Swap Bias, as described in option 2, and its application is the same in those instances when uniform crossover is used. The user will then be asked for a sequence of crossover rates and crossover numbers indicating the mixture of crossover types to use. A crossover number of 0 indicates uniform crossover, while any positive integer k indicates a k-point crossover. k point crossover is implemented by selecting k points at random from the range 1 to $l - 1$. The selection is not exclusive, and so it is possible

[6] C is in the Hamming closure of $\mathbf{P}_1$ and $\mathbf{P}_2$ if for each $j, 1 \leq j \leq l$, $C_j = \mathbf{P}_{2j}$ or $C_j = \mathbf{P}_{1j}$.

[7] In crossover space, two points are adjacent if one can be produced from the other by a single crossover operation. It only makes sense when points are populations (in this case of size 2) of strings. In Hamming space, points represent single strings, and two points are adjacent when they differ in exactly one character position.

that fewer than k crossover points will be generated, although at least one crossover point is guaranteed.

The crossover rate numbers give the frequencies at which the different crossover types will be selected. The input to this set of values is terminated by the entry 0 0. For example, if the user enters

10 0
15 1
5 2
0 0

then uniform crossover will be selected on average 10 times out of 30, one point crossover 15 times out of 30 and two point crossover 5 times out of 30. The selections are made randomly and independently for each pair of each family.

Selection Parameters

This parameter determines the method of selecting parents in the first step of the mating cycle.

Selection Type An integer 1 to 3 chooses the desired selection type.

Option 1 selects adjacent pairs at random from the population.

Option 2 selects adjacent pairs starting from the bottom, and proceeding in sequence until the top most pair has been selected. It then restarts at the bottom on the next selection request.

Option 3 behaves like option 2 until the top most pair is selected, then instead of restarting at the bottom, it proceeds instead from the top to the bottom. When the bottom is reached, the pattern is repeated.

Termination Parameters

Maximum Matings The number of matings before termination.

Report Parameters

The program prints out the number of matings, total string evaluations, minimum value of the population and the maximal value of the population exactly when a new minimum or maximum has been found.

The user has the option of having the population printed out at various times. These options are; before the initial population is sorted, after the initial population is sorted (if sorting is specified), during execution, in which case the user specifies the frequency of printing, and after execution is complete. These options are selected by a sequence of yes or no responses.

Experiment Parameters

More than one run of an experiment may be requested, and if so the program will keep track of averages and other information and print a summary report.

Number of Experiments A positive integer indicates the number of experiments to perform. If this is greater than the constant MAXEXPER defined in **giga.h** the minimum number of evaluations for finding the minimum and maximum values will not be summarized for all experiments. A warning will be printed in this case.

Collection Frequency A positive integer indicates how often to collect data. If the number of collection points (computed as the number of iterations divided by the frequency) is greater than MAXCOLLECT then some information will be lost. A warning will be printed in this case.

Function Parameters

Which Function An integer 1 to 17 selects the desired function.

Function's 1 to 5 are the De Jong functions described in [5], and adapted from the GENESIS system[8] (files **fdj1.c** to **fdj5.c**).

Option 6 selects the *one max* function, sometimes called the ones counting function (file **f00.c**). This function returns the number of '1' characters in the string.

Option 7 selects a function that adds together the character values of the string (file **f01.c**).

Option 8 selects the *majority* function which computes the absolute difference in the number of 1's and 0's in the string (file **f02.c**).

[8] Any errors that have been introduced should not be attributed to GENESIS, but to the adapter.

Options 9 to 12 are various attempts at deceptive functions and are discussed in [2] (files **f03.c** to **f06.c** in that order).

Option 13 interprets a binary string as a binary number (file **fbin.c**).

Option 14 is Goldberg's 3-bit deceptive function [8]. String length must be a multiple of three, and $\alpha = 2$. The loosely ordered version can be selected by using the parameter passing option described under the heading "parameters to functions" (file **fdgb.c**).

Option 15 is Liepin's deceptive function [12] as described in [6]. This function requires the string length to be a multiple of the order of deception. The default order of deception is 5, but can be changed using the parameter passing mechanism (file **fdl.c**).

Options 16 and 17 are deceptive functions for GIGA that have the optimal points not at the opposite pole from the suboptimal points in Hamming space. These are discussed in [2] and are found in files **f07.c** and **f08.c**.

Gray Code Interpretation Entering 'y' or 'n' sets a flag that can be accessed by the function to determine whether to interpret a binary string as a reflected Gray code instead of the usual binary number encoding. Only the De Jong functions currently access this flag.

Parameters to Functions A chance to enter real values to be passed to the function being evaluated. Up to ten real values may be passed to the function, provided the function elects to make use of them. Currently only the De Jong functions, **fdgb.c** and **fdl.c** make use of this. In De Jong functions, a value called "center" can be set by entering a value in this array. If no values are entered, then the default center values are used. For each of the functions 3 and 4 one other parameter may be set. Function 3 adds an offset to the final sum, and this can be altered by entering a second value in the parameter list. However, this is rather pointless since GIGA is insensitive to linear scaling. Function 4 uses a noise function, and the degree of the noise can be altered by entering a second number in the parameter list.

Function **fdgb.c** tests the first parameter and if it has been set to anything (other than MAXDOUBLE) the loosely ordered version of Goldberg's deceptive function is used, otherwise the tightly ordered version is used.

Function **fdl.c** implements Liepin's deceptive function with a default of order 5 deception. If the first parameter value is set, then this value is used as the order of deception.

The user is advised to look at the function encodings to see how these values are used. The values are passed to the functions in the array OPTIONS.parameters[];

Adding New Functions

New functions can easily be added to the program. First, the function must be given a name distinct from the functions already defined. It is a good idea to place the source in a file of a similar name. Functions take one argument, a pointer to a character string, which contains the individual to be evaluated. String length, alphabet size and optional arguments are passed through the OPTIONS data structure. See the source file **fdj3.c** for an example.

Functions must return a *double* value. Linkage is made through the evaluate routine in the file **evaluate.c** and you will need to add a case in the switch statement to call your function. You should also change the selection printout under the heading `**** FUNCTION ****` in the file **init.c** so the user will be aware of the existence of the function. Finally, add the source file name with the ".o" suffix to the **makefile** file and make the program again.

5 Design Philosophy and Future Options

The reader will undoubtedly have many ideas about what should be added to this program, or future versions of it. One obvious operation that should be allowed is mutation. This destroys the invariance property, and this loss of purity is the reason it is not included in this program. This program's purpose is to illustrate the effects of crossover and invariance, and allowing mutation could have distracted from that purpose. But some functions would be more readily optimized if the operation were available. If the approach is ever to become applicable, mutation should be selectable at the user's discretion.

In the **mating parameters**, options 1,2 and 3 for the definition of best pair are equivalent for linear functions of the character strings, in the sense that GIGA will produce the same results on these functions using any of these options. Many other functions also make these options equivalent. For example, if the options are equivalent for function $g(x)$, and f is a monotonic function, then they are equivalent for $f(g(x))$. It would be interesting to have a characterization of all functions yielding equivalence. One characteristic is "the difference between a pair of children may be greater than the parents if and only if the maximum increases and the minimum decreases".

One reason this is of interest is the desire to make GIGAs universal. That is, to optimize a function, we would like to have to specify as little special knowledge as possible. Admittedly, not deciding whether to minimize or maximize does seem a bit extreme. Another reason relates to the analysis of the attempted deception in [2]. There the production of suboptimal values leads to a refined search in minimal areas, which leads in turn to the optimal value. One wonders whether arbitrarily complex behavior can be obtained, and what criteria for "best" would abet such behavior.

Many other definitions of "best" are possible, including the pair with greatest or least total value. Such total value selection might have benefit in some cases by more quickly concentrating the search.

Future mating options could include more choices than the yes or no response to elitism. For example, we could mimic simulated annealing and select children to replace their parents with some probability when the parents are superior to their children. This probability would possibly decline over time.

Family size could be made variable, and perhaps be made greater at the appropriate end of the population, thus mimicing to some extent the TGA's tendency to concentrate search on the best values.

In **crossover parameters** option 1 is redundant and should be removed. Option 4 is an ad hoc approach to the perceived need for mixed crossover types noted in the analysis of the deceptive attempt in [2]. Research into more robust methods that take notice of previous success rates and are time dependent may lead to more general and effective results. Targeted crossover, in which a few bits are exchanged, with the exchanges all taking place where the strings differ, might be effective in many functions.

In **selection parameters** the motivation behind the options comes from viewing GIGAs as processes which sort high value schemata from low value schemata. Option 1 can be seen as a randomized sorting process. Recall that when the parents are replaced, the larger valued child always goes into the top most position. (If no replacement is done, then the parents will swap position if necessary). Thus, if the function is well behaved, sorting will gradually take place.

Option 2 behaves like bubblesort [1], and in fact would degenerate to bubblesort if no children were ever produced.

Option 3 behaves like alternating bubble sort, which is called the cocktail shaker sort [10].

However, these analogies are not quite sufficient. Unlike sorting, the basic operation of crossover does not guarantee that anything moves upwards. In fact, if we consider the function **f00.c**, the ones counting function with the sorting options turned off, then we may observe that when using option 2 and uniform crossover there is a limit to how much exchange is likely to occur per mating. If the mating is between $\mathbf{P}_i$ and $\mathbf{P}_{i+1}$, and $\mathbf{P}_i$ has many more ones than its upper neighbor, then the two children will each obtain about 1/2 of the excess ones. If we have elitism, with high probability the next mating cycle will result in swapping the parents. If elitism is turned off, then somewhat over 1/2 of the excess ones will be moved up on each mating cycle.

If option 3 is used on the ones counting function, the upward passes sweep excess ones towards the top, and the downwards passes sweep excess zeroes towards the bottom. This process is repeated until the ones and zeroes are separated.

If sorting is turned on, then we tend to mate strings with more similar numbers of ones, and the sweeping actions are more efficient. For a particular mating, we get ones moving upwards depending on the variance due to the randomness of crossover. But these ones are moved further upwards by the sorting action, and the next mating will again be between strings of more or less equal value.

Future strategies might implement something more akin to parallel sorting. Instead of mating adjacent pairs, we would mate alternating adjacent pairs, then on the second pass those at distance two, then at distance four and so on. The basic approach would be much like the parallel merge sort described in [1]. With the sorting option available, it is not clear that this technique would be of benefit.

The selection method in [11] could be mimiced in this program by adding a selection type that would only select the adjacent pair with minimal value difference. Using this selection method, the sorting option, setting family size to one and turning elitism off would be equivalent to Lewchuk's method[11]. The definition of "best" would be irrelevant in this case.

In the current program, option 3 is recommended as it appears to be the best option on all functions we have tried.

Selection of non-adjacent elements as parents may also help in many cases, in particular on functions such as the De Jong functions. The program may be used as a tool to explore the interaction of function value and crossover space. With the current limitation this interaction is more tightly coupled, and so may be more tractable from a theoretical stand point.

Termination is currently determined only by exceeding the maximum number of matings. Future options might include measures of progress, or measures of population diversity and base termination on these measures. A system that would allow user intervention, changing various parameter settings and continuation of a run would greatly extend the usefulness of this program as a research tool.

References

[1] Sara Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley Publishing Company, Inc., 1978.

[2] Joseph C. Culberson. Genetic invariance: A new paradigm for genetic algorithm design. ftp thorhild.cs.ualberta.ca, April 1992.

[3] Lawerence Davis, editor. *Genetic Algorithms and Simulated Annealing*. Research Notes in Artificial intelligence. Morgan Kaufmann, 1987.

[4] Lawerence Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.

[5] K.A. DeJong. *Analysis of Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, The University of Michigan, 1975.

[6] Larry J. Eshelman. The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. In Gregory J. E. Rawlins, editor, *Fondations of Genetic Algorithms (FOGA I)*, pages 265–283. Morgan Kaufmann, 1991.

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

[8] David E. Goldberg, Bradley Korb, and Kalyanmoy Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3:493–530, 1989.

[9] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[10] D. E. Knuth. *Sorting and Searching*, volume III of *The Art of Computer Programming*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1973.

[11] Michael Lewchuk. Genetic invariance: A new approach to genetic algorithms. Master's thesis, University of Alberta, Edmonton Alberta, April 1992. Technical Report TR 92-05 "Genetic Invariance: A New Type of Genetic Algorithm" ftp thorhild.cs.ualberta.ca.

[12] Gunar E. Liepins and Michael D. Vose. Representational issues in genetic optimization. *Journal of Experimental and Theoretical AI*, May 1991.