Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing, ACM Press: New York (1993).

LIBGA: A USER-FRIENDLY WORKBENCH FOR ORDER-BASED GENETIC ALGORITHM RESEARCH*

Arthur L. Corcoran
Roger L. Wainwright

Department of Mathematical and Computer Sciences
The University of Tulsa

Abstract

Over the years there has been several packages developed that provide a workbench for genetic algorithm (GA) research. Most of these packages use the generational model inspired by GENESIS. A few have adopted the steady-state model used in Genitor. Unfortunately, they have some deficiencies when working with order-based problems such as packing, routing, and scheduling. This paper describes LibGA, which was developed specifically for order-based problems, but which also works easily with other kinds of problems. It offers an easy to use 'user-friendly' interface and allows comparisons to be made between both generational and steady-state genetic algorithms for a particular problem. It includes a variety of genetic operators for reproduction, crossover, and mutation. LibGA makes it easy to use these operators in new ways for particular applications or to develop and include new operators. Finally, it offers the unique new feature of a dynamic generation gap.

Introduction

Genetic algorithms (GAs) are based on the principle of natural evolution. A fundamental premise of genetic algorithms is that they can solve complicated problems by simulating evolution. GAs are able to prune a search space and generate plausible solutions within the problem specified constraints by emulating biological selection and reproduction. Over the years there has been several genetic algorithm packages developed by researchers. The majority of these use the 'generational' model first described by Holland [5] and later made more popularly known by Goldberg [3]. The first widely available implementation of a generational GA was in GENESIS [4]. A few GA packages use the newer 'steady-state' model first used in Genitor [9].

The problem with existing GA packages is that they are very difficult to use in one way or another for performing research into order-based problems. These problems include packing, routing, scheduling, and other 'NP-complete' problems. Consequently, the authors have developed a GA package called LibGA which incorporates many of the different options and variations of existing GA packages. It also specifically provides support for order-based problems. LibGA is a library of various GA functions and operations that have been packaged into an easy to use system for researchers to develop GAs for their particular problems.

This paper provides a brief introduction to genetic algorithms and surveys existing packages. It describes LibGA in detail and provides an example of how it can be used.

Genetic Algorithms

A genetic algorithm is an iterative procedure which borrows the ideas of natural selection and 'survival of the fittest' from natural evolution. By simulating natural evolution in this way, a GA can easily solve complex problems. Furthermore, by emulating biological selection and reproduction techniques, a GA can effectively search the problem domain in a general, representation-independent manner.

The genetic algorithm maintains a population or pool of candidate solutions for a given objective function. The candidate solutions represent an encoding of the problem into a form that is analogous to the chromo-

*Research partially supported by OCAST Grant AR2-004 and Sun Microsystems, Inc.

ACM-SAC '93/2/93/IN, USA
©1993 ACM 0-89791-568-2/93/0002/0111...$1.50

somes of biological systems. Each chromosome is made up of a string of genes (whose values are called alleles). The chromosome is typically represented in the GA as a string of bits. However, integers and floating point numbers have been used. The chromosome also has associated with it a fitness value, which is found by evaluating the chromosome with the objective function. It is the fitness of a chromosome which determines its ability to survive and produce offspring.

Once an initial pool has been generated and all of its members have been evaluated, the genetic algorithm begins its emulation of the life cycle. At each step in the iteration, chromosomes are probabilistically selected from the population for reproduction. Offspring are generated through a process called crossover, which can be augmented by mutation. The offspring are then placed back in the pool, perhaps replacing other members of the pool. This process can be modeled using either a 'generational' [3, 5] or a 'steady-state' [9] genetic algorithm. The generational GA saves offspring in a temporary location until the end of a generation. At that time the offspring replaces the entire current population. Conversely, the steady-state GA immediately places offspring back into the current population.

The genetic algorithm relys on genetic operators for selection, crossover, mutation, and replacement. The selection operators use the fitness values to select a portion of the population to be parents for the next generation. Parents are combined using the crossover and mutation operators to produce offspring. This process combines the fittest chromosomes and passes superior genes to the next generation, thus providing new points in the solution space. The replacement operators ensure that the 'least fit' or weakest chromosomes of the population are displaced by more fit chromosomes.

While the fundamental concepts of genetic algorithms are fairly simple and straightforward, there are numerous implementation variations and options to incorporate into a genetic algorithm. For example, there are numerous ways to parametrize a model and encode it into a finite length chromosome. There are numerous selection techniques for determining chromosomes for crossover. There are literally dozens of possible crossover operators that have been developed in recent years depending on the problem type and chromosome encoding scheme. There are also several techniques for introducing some random changes to a chromosome (i.e., mutation).

Existing GA Packages

The first widely available genetic algorithm was GENESIS [4], written by John Grefenstette in 1984. Since that time, a variety of genetic algorithm packages have been developed. Most of these use the generational model. However, a few use the steady-state model introduced with Genitor [9] in 1988. The following is an overview of existing GA packages. For a current list of GA packages, the reader is referred to the GA Software Survey [6]. It contains more detailed information concerning what is available and how to obtain GA packages.

GAucsd [7] is a GENESIS-based genetic algorithm. It offers several bug fixes and an improved user interface. It can additionally distribute experiments over a network of machines. Perhaps the most notable feature is Dynamic Parameter Encoding (DPE). DPE is a technique which is used in continuous search spaces. It redefines the encoding used for the chromosome, adapting the 'granularity' to the convergence rate. Like GENESIS, GAucsd was designed primarily for use on bit string type genomes, which does not work well with order-based problems. While GAucsd can encode the integers using grey-code bit strings, it does not ensure that order is preserved. This can produce chromosomes with invalid allele orderings or alleles with meaningless bit patterns. GAucsd uses a variant of roulette for selection, a two-point crossover, and a bit inverting mutation.

GENEsYs [6] is another notable generational genetic algorithm. It is based on GENESIS and includes selection schemes such as linear ranking, Boltzmann, (mu,lambda)-selection, and general extinctive selection variants. Uniform and n-point crossover as well as discrete and intermediate recombination are included. Mutation can be self-adaptive. Unfortunately, it doesn't give the user the ability to test a steady-state model with these options.

Genitor is a steady-state GA which uses a rank-based, biased selection and weakest chromosome replacement. It has the ability to work with bit, integer and floating point types. It includes an example GA for the traveling salesman problem with order, PMX and edge-based crossover operators. It can also perform subpopulation modeling. Genitor has been used successfully by the authors [1] in the past for order-based problems. However, steady-state genetic algorithms can converge prematurely. They require large pool sizes and many trials to ensure the best solution is found.

LibGA

LibGA was inspired by the authors frustration at the deficiencies of existing GA packages. Previous research had been limited to the steady-state GA used in Genitor augmented with custom order-based crossover and

```
#include "ga.h"

main()
{
   GA_Info_Ptr ga_info;
   int         obj_fun();

   ga_info = GA_config("ga.cfg", obj_fun);
   GA_run(ga_info);
}

obj_fun(chrom)
   Chrom_Ptr chrom;
{
   chrom->fitness = /* fitness */
}
```

Figure 1: Simple Interface to LibGA

mutation functions. However, the problems that steady-state GAs have with premature convergence led to the desire to use the more commonly used generational genetic algorithm. The authors decided that modifying Genitor to include a generational model was too great a task. Consequently, the authors decided that the only solution was to develop their own genetic algorithm: LibGA.

LibGA is a collection of routines written in the C programming language under a Unix platform. It includes a variety of genetic operators for reproduction, crossover, and mutation. Routines are provided which implement both generational and steady-state genetic algorithms using the genetic operators. Other routines are provided for initialization, reading the configuration file, and generating reports.

Interface

Figure 1 illustrates the interface to LibGA. It is very simple. The file `ga.h` is included to obtain the necessary definitions. In the main function, `GA_config()` is first called to initialize and configure the global parameters as specified in the configuration file. A pointer to these parameters is returned by `GA_config()` and assigned to `ga_info`. The argument `"ga.cfg"` is the pathname of the configuration file and `obj_fun` is a pointer to the user's objective function. `GA_run()` is then invoked to run the genetic algorithm using the parameters in `ga_info`.

The user's objective function is called whenever the genetic algorithm needs to evaluate a chromosome. A pointer to the chromosome in question is passed to the objective function. The objective function must decode the chromosome (if necessary), compute the fitness value, and assign that value to the chromosome.

```
/*--- GA configuration info type ---*/
typedef struct {
   /*--- Basic info ---*/
   char  user_data[80];  /* User data file (unused) */
   int   rand_seed;      /* Random seed */
   int   datatype;       /* Data type */
   int   ip_flag;        /* Initial pool flag */
   char  ip_data[80];    /* Initial pool data */
   int   pool_size;      /* Pool size */
   int   chrom_len;      /* Chr. size */
   int   num_trials;     /* Number of trials */
   in%   minimize;       /* Minimize EV_fun [y/n]? */
   ixt   elitist;        /* Use elitism [y/n]? */
   float bias;           /* Selection bias */
   float mu_rate;        /* Mutation rate */
   float gap;            /* Generation gap */

   /*--- Functions ---*/
   int   (*GA_fun);     /* GA */
   int   (*SE_fun);     /* Selection */
   int   (*X_fun);      /* Crossover */
   int   (*MU_fun);     /* Mutation */
   int   (*EV_fun);     /* Evaluation */
   int   (*RE_fun);     /* Replacement */

   /*--- Reports ---*/
   int   rp_type;      /* Type of output report */
   int   rp_interval;  /* Output report interval */
   char  rp_file[80];  /* Output report file name */

   /*--- Pools ---*/
   Pool_Ptr old_pool, new_pool;

   /*--- Stats ---*/
   Chrom_Ptr best;       /* Best chromosome */
   int num_mut, tot_mut; /* Mutation  Statistics */
} GA_Info_Type;
```

Figure 2: LibGA's GA_Info Structure Type

Figure 2 shows the `GA_Info_Type` structure used by LibGA. A pointer (`GA_Info_Ptr`) to a `GA_Info_Type` structure is declared in the user's program. LibGA uses the information contained in this structure to run the genetic algorithm. Note this makes it possible to run several genetic algorithms in parallel, or to run a hybrid GA which dynamically changes parameters while the program is running. When `GA_config()` is invoked, a `GA_Info_Type` structure is allocated and all of its parameters are set to their default values. Consequently, the default configuration information can be changed by modifying the source code which performs the initialization. However, this is not recommended.

The recommended way to change LibGA's configuration from the defaults is through the use of a configuration file. The user will most likely begin using the file `"ga.cfg"` included with LibGA. The file is parsed by LibGA using a free format. Comments are defined as the text between a '`#`' character and the end of line character. Initially, the entire file contains comments. Each configuration directive has a description indicating its use, default value, side effects, etc. Following the

```
#-----------------------------------------------
# GA Type:
#    generational - generational GA
#    steady-state - steady-state GA
#
#    WARNING: This directive has the following
#             side effects:
#
#       directive    generational  steady-state
#       ---------    ------------  ------------
#       selection    roulette      rank_biased
#       replacement  append        by_rank
#       rp_interval  1             100
#
#    DEFAULT: generational
#-----------------------------------------------
# ga generational
# ga steady-state
```

Figure 3: Sample Configuration File Directive

description are (commented out) examples of how to use the directive. The user need only remove the '#' character to use the new directive to change the configuration for his or her program.

Figure 3 illustrates the use of a directive in the configuration file. The description indicates that this directive, `ga`, sets the GA type. The parameter can be set to either `generational` or `steady-state`. The default value is `generational`. This directive also has the side effect of modifying the values for the `selection`, `replacement`, and `rp_interval` directives for the respective options. To configure the GA for a steady-state method, the user should remove the '#' from the '`ga steady-state`' directive.

Data Structures

There are four data types on which LibGA depends: GA_Info_Type, Gene_Type, Chrom_Type, and Pool_Type. The GA_Info_Type contains all of the configuration information for a particular problem instance, as well as the results after the GA has finished. The Gene_Type is the basic data type (e.g., unsigned int) for each gene or allele in a chromosome. The Chrom_Type defines a chromosome, which includes a collection of genes, fitness value for the chromosome, and other useful information. The Pool_Type, which defines the pool or pools, likewise includes a collection of chromosomes, collective pool fitness, best and worst chromosomes, etc. Pointers to the structures are GA_Info_Ptr, Gene_Ptr, Chrom_Ptr, and Pool_Ptr, respectively. These structures are easily available for a researcher to use and perhaps adapt for special needs.

Genetic Operators

LibGA includes operators for selection, replacement, crossover, and mutation. In all cases, selection and replacement can be augmented with elitism, which ensures that the best member survives into the next generation.

Selection

The selection operators included in LibGA are: *uniform-random*, *roulette*, *min-roulette*, and *rank-biased*. Uniform-random selection simply picks a member of the pool at random, completely ignoring fitness or other factors. Each chromosome in the pool is equally likely to be selected.

Roulette selection is the classical selection operator for generational genetic algorithms as described in Goldberg [3]. Each member of the pool is assigned space on a roulette wheel proportional to its fitness. The members with the greatest fitness have the highest probability of selection. This selection technique works only for a genetic algorithm which maximizes its objective function. The authors are not aware of any literature describing a roulette wheel selection technique for minimizing a function. Consequently, min-roulette selection was developed for this class of problems.

Min-roulette is a very straightforward algorithm. The trick is to make the percentage of total fitness for chromosome i, p_i, inversely proportional to the fitness of that chromosome, f_i. That is, smaller f_i results in larger p_i. This is done by scaling the fitness for each chromosome relative to the total fitness,

$$f_i' = \frac{\sum f_i}{f_i}$$

then scaling the result,

$$p_i = \frac{f_i'}{\sum f_i'}$$

to obtain the percentage of total fitness.

Rank-biased selection is the selection method used in Genitor. The pool is sorted by fitness value and chromosomes are selected using a selection bias parameter. The bias (which ranges from 1.0 to 2.0) specifies the amount of preference to be given to the best members of the pool. For example, a bias value of 2 indicates that the best member is twice as likely to be selected as the average member of the pool.

Replacement

The available replacement operators are: *append*, *by-rank*, *first-weaker*, and *weakest*. The append replacement operator appends new chromosomes to an existing pool. This operator is used in the classical generational GA to place offspring in the new pool. The by-rank operator is the one used in Genitor. The pool is ranked by sorted fitness value. If the chromosome has a high enough fitness, it will be placed in the pool, displacing weaker chromosomes. If its fitness is not better than the weakest member of the pool, it is a 'failure' and is not placed in the pool. The weakest and first-weaker operators are somewhere between the append and by-rank operators. In this case, the pool is not sorted by fitness value. A chromosome may only be placed in the pool if it can find a weaker chromosome to displace. The first-weaker operator performs a linear scan through the pool to find a weaker chromosome and replaces the first one it encounters. The weakest operator replaces the weakest member of the pool with the new chromosome unless the new chromosome itself is weaker.

Crossover

LibGA's crossover operators are: *simple*, *order1*, *order2*, *position*, *cycle*, *PMX*, and *asexual*. Simple crossover is used for traditional bit string encodings of the chromosome. A random crossover point is selected which divides each parent into two parts. Alternate parts are contributed by each parent to generate two offspring. This is also known as single point crossover. This does not work for order-based problems since order is not preserved. The other crossover operators will preserve order information. The order1, order2, position, cycle, and PMX operators are described in Starkweather *et al.* [8]. The asexual operator is a simple swap of two randomly selected genes, which also is suitable for order-based problems.

Mutation

LibGA currently offers the mutation operators: *simple-invert*, *simple-random*, and *swap*. The simple-invert and simple-random operators are used in bit string chromosome representations. They both mutate at random, based on the mutation rate. They also pick which bit is to be mutated at random. The difference is that simple-invert inverts the bit and simple-random selects the value randomly. Since the random selection could choose the same bit value, one would expect simple-random to invert the bit only half of the time that mutation occurs. The swap mutation operator is included for integer chromosome representations. Like the other operators, it randomly mutates based on the mutation rate. However, the mutation swaps two randomly selected alleles. Note that asexual crossover and swap mutation are virtually the same.

Dynamic Generation Gap

In generational genetic algorithms the offspring are saved in a separate pool until there are as many as in the original pool. Then the offspring's pool replaces the parent's pool for the next generation. The situation is quite different in the steady-state genetic algorithm where the offspring and parents occupy the same pool. Clearly, these two cases represent the two extremes of overlap between the generations. Consequently, the authors have included 'generation gap' with LibGA. Generation gap is a parameter used in genetic algorithms to specify the amount of overlap desired. In a recent paper, De Jong [2] concluded that generation gap has little importance in a genetic algorithm. He further concluded that the choice of selection and replacement strategies have the most profound effect. However, De Jong based his results on tests performed on genetic algorithms *without* crossover or mutation. Hence, this is still an open question. Consequently, the generation gap parameter has been incorporated into LibGA so that further testing can be done on 'real-world' genetic algorithms. The authors are also currently evaluating the effects of a dynamic generation gap on genetic algorithms. That is, how the GA is effected by varying the generation gap *while the algorithm is running.*

LibGA Applied

The authors are particularly interested in applying genetic algorithms to packing, routing, and scheduling problems [1]. These 'NP-complete' problems have no known polynomial time solution. It is hoped that genetic algorithms can provide an acceptable approximation for these problems.

The following is a description of an example use of LibGA on a two-dimensional packing problem. The goal is to minimize the height of the packing obtained using next fit in a two-dimensional, open-ended bin. The test was limited to a comparison of the different crossover techniques under generational and steady-state models. The generational model used min-roulette selection and append replacement. The steady-state model used uniform-random selection and first-weaker replacement. Elitism was used, but mutation and generation gap were not. The pool size was fixed at 100. The number of generations were bounded at 200 for the generational model

Crossover	Generational Best	Ave	Worst	Steady-State Best	Ave	Worst
Order1	21	23.1	25	**20**	**21.5**	**23**
Order2	21	23.0	25	21	23.7	25
Position	23	24.0	25	21	22.4	25
Cycle	24	25.7	27	23	25.0	26
PMX	23	24.0	25	21	22.8	25
Asexual	**20**	**21.3**	**23**	21	23.2	25
Average	22.0	23.5	25.0	21.2	23.1	24.8

Table 1: Results with 25 Packages (Opt. = 20)

Crossover	Generational Best	Ave	Worst	Steady-State Best	Ave	Worst
Order1	93	97.6	102	92	96.7	100
Order2	95	98.1	101	100	105.5	110
Position	105	107.0	110	101	104.2	108
Cycle	107	110.7	113	107	109.2	112
PMX	101	104.6	108	96	100.7	103
Asexual	**92**	**93.0**	**95**	**90**	**94.7**	**102**
Average	98.8	101.8	104.8	97.7	101.8	105.8

Table 2: Results with 100 Packages (Opt. = 80)

and 100,000 for the steady-state model. Note that these differences reflect the different definition of a generation in either model.

The tests consist of two contrived data sets containing 25 and 100 packages, which can be optimally packed with heights of 20 and 80, respectively. Note that the 25 package data set has approximately 10^{25} possible orderings and the 100 package data set has over 10^{157} possible orderings. Each data set was run under generational and steady-state models, using order1, order2, position, cycle, PMX, and asexual crossover operators. Each test was run using 10 different random seeds.

Table 1 shows the results of running LibGA on the 25 package data set. The values under *best*, *ave*, and *worst* indicate the distribution of the fitness obtained under the 10 random seeds. That is, they show the range of best values obtained by each test. As indicated in bold face, the best performing crossover operator on average for the generational model was asexual crossover. The best for the steady-state was order1. Considering the average performer, the best generational crossover was marginally better than the best steady-state crossover (21.3 versus 21.5). However, the steady-state was marginally better when averaging all crossover operators (23.1 versus 23.5). Asexual and order2 crossover gave better results on average for the generational model while the other crossover operators performed better using the steady-state model. Note the generational GA had not yet converged in all cases using asexual crossover and in most cases using the order2 crossover. The steady-state GA converged in all cases. The lack of convergence actually made no difference since the generational GA already had better fitness. The optimal packing was found in 4 of the runs for generational and 3 of the runs for steady-state.

Table 2 shows the results of running LibGA on the 100 package data set. As indicated in bold face, the asexual crossover operator was the best performing crossover for both models. This was a surprising result. It had significantly better performance under the generational model. When averaging the performance of all crossover operators, the result was a tie (101.8). As in Table 1, the asexual and order2 were winners for generational. Order1, PMX, position, and cycle were winners for steady-state. The optimal packing was never found. The best value was 90 (1.12 times optimal), which is much better than expected using next fit.

It is important to remember that these results are based on the particular problem and GA parameters used. However, it shows how useful LibGA can be by giving the researcher the ability to compare different techniques and options, and construct a particular GA specific for the given application.

Conclusions

LibGA provides a user-friendly workbench for order-based genetic algorithm research. It can also be used in other problem domains. It provides a rich set of operators for selection, crossover, mutation, and replacement. Unlike any other GA package, LibGA allows the researcher to compare both generational and steady-state models using all of these operators. LibGA makes it easy for the researcher to add new operators and even new or hybrid GA models specific to an application. Other features such as elitism, generation gap, and the unique new dynamic generation gap feature enhance LibGA's problem solving skill. In the future, LibGA will be extended to include more operators and features. It will be used not only for studying specific problems, but also for studying genetic algorithms in general (theory and applications).

Acknowledgements

This research has been partially supported by OCAST Grant AR2-004. The authors also wish to acknowledge the support of Sun Microsystems, Inc.

LibGA Availability

LibGA is available at no cost by sending an email request to the authors. The authors respective

email addresses are *corcoran@penguin.mcs.utulsa.edu*, and *rogerw@penguin.mcs.utulsa.edu*. Conventional mail should be addressed to the authors at:

Department of Mathematical and Computer Science
University of Tulsa
600 South College Avenue
Tulsa, OK 74104-3189
USA

References

[1] A. L. Corcoran and R. L. Wainwright. A genetic algorithm for packing in three dimensions. In H. Berghel, E. Deaton, G. Hedrick, D. Roach, and R. Wainwright, editors, *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing*, pages 1021–1030, New York, 1992. ACM Press.

[2] K. A. De Jong and J. Sarma. Generation gaps revisited. In D. Whitley, editor, *Foundations of Genetic Algorithms 2*. Morgan Kaufman, San Mateo, California, 1993.

[3] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, Massachusetts, 1989.

[4] J. J. Grefenstette. GENESIS: A system for using genetic search procedures. In *Proceedings of the Conference on Intelligent Systems and Machines*, pages 161–165, 1984.

[5] J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, Michigan, 1975.

[6] N. N. Schraudolph. Genetic algorithm software survey. Available by anonymous ftp from cs.ucsd.edu as /pub/GAucsd/GAsoft.txt, Aug. 1992.

[7] N. N. Schraudolph and J. J. Grefenstette. A user's guide to GAucsd 1.4. Technical Report CS 92-249, University of California, San Diego, July 1992.

[8] T. Starkweather, S. McDaniel, K. Mathias, D. Whitley, and C. Whitley. A comparison of genetic sequencing operators. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 69–76, San Mateo, California, 1991. Morgan Kaufmann.

[9] D. Whitley and J. Kauth. GENITOR: A different genetic algorithm. In *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, pages 118–130, Denver, Colorado, 1988.