A User's Guide to GAucsd 1.4 [*]

Nicol N. Schraudolph
nici@cs.ucsd.edu
Computer Science & Engineering Department
University of California, San Diego
La Jolla, CA 92093-0114

John J. Grefenstette
gref@aic.nrl.navy.mil
Navy Center for Applied Research in Artificial Intelligence
Naval Research Laboratory
Washington, DC 20375-5000

July 7, 1992

Abstract

This document describes the GAucsd system for function optimization based on genetic search techniques. Genetic algorithms appear to hold a lot of promise as general purpose adaptive search procedures. However, the authors disclaim any warranties of fitness for a particular problem. The purpose of making this system available is to encourage the experimental use of genetic algorithms on realistic optimization problems, and thereby to identify the strengths and weaknesses of genetic algorithms.

GAucsd was developed by Nicol Schraudolph at the University of California, San Diego; it is based on Genesis 4.5, a genetic algorithm package written by John J. Grefenstette. GAucsd and related materials are available via anonymous ftp from *cs.ucsd.edu* (132.239.51.3) in the *pub/GAucsd* directory or via electronic mail from the first author, who welcomes bug reports, comments and suggestions, and maintains a mailing list of users to announce patches and new releases.

[*]Hardcopies of this document can be obtained by requesting technical report CS92-249 from Technical Reports, CSE Department, UC San Diego, La Jolla, CA 92093-0114. There is a charge of 5 US$ for this service.

1 Introduction

This document describes the GAucsd software package written to promote the study of genetic algorithms for function minimization. Since genetic algorithms are task-independent optimizers, the user must provide only an *evaluation function* which returns a value when given a particular point in the search space. GAucsd was written in C under the Unix[1] operating system, but should be portable to many platforms.

The remainder of this section gives a general overview of genetic algorithms (GAs); for an in-depth introduction see [8]. Details on porting, installing, and running GAucsd are provided in subsequent sections.

GAs are iterative procedures which maintain a *population* P of n candidate solutions to an objective function f:

$$P(t) = \{x_1(t), x_2(t), \ldots x_n(t)\}$$

Each structure x_i in population $P(t)$ at time t is simply a binary string of length l. Generally, each x_i represents a vector of parameters to the function $f(x)$, but the semantics associated with the vector is unknown to the GA. During each iteration step, called a *generation*, the current population is evaluated, and, on the basis of that evaluation, a new population of candidate solutions is formed. A general sketch of this procedure — known as *generational GA* — is shown in Figure 1.

```
t <- 0;
initialize P(t);
evaluate P(t);
while (not finished) do
begin
        t <- t + 1;
        select P(t) from P(t - 1);
        operate on P(t);
        evaluate P(t);
end;
```

Figure 1: A Generational Genetic Algorithm.

The initial population $P(0)$ is usually chosen at random. Alternately, the initial population may contain heuristically chosen initial points. In either case, the initial population should contain a wide variety of structures. Each structure in $P(0)$ is then evaluated. For example, if we are trying to minimize a function f, evaluation might consist of computing and storing $f(x_1), \ldots f(x_n)$.

[1] Unix is a trademark of Bell Laboratories

The structures of the population $P(t+1)$ are chosen from the population $P(t)$ by a randomized *selection procedure* that ensures that the expected number of times a structure is chosen is proportional to that structure's performance, relative to the rest of the population. That is, if x_j has twice the average performance of all the structures in $P(t)$, then it is expected to appear twice in population $P(t+1)$. At the end of the selection procedure, population $P(t+1)$ contains exact duplicates of the selected structures in population $P(t)$.

In order to search other points in the search space, some variation is introduced into the new population by means of idealized *genetic operators*. The most important operator is called *crossover*. Under the crossover operator, two structures in the new population exchange portions of their binary representation. This can be implemented by choosing a point at random, called the crossover point, and exchanging the segments to the right of this point. For example, let

$$x_1 = 100 : 01010 \text{ and } x_2 = 010 : 10100,$$

and suppose that the crossover point has been chosen as indicated. The resulting structures would be

$$y_1 = 100 : 10100 \text{ and } y_2 = 010 : 01010.$$

Crossover serves two complementary search functions. First, it provides new points for further testing of the *schemata* already present in the population. In the above example, both x_1 and y_1 are representatives of the schema 100#####, where the # means "don't care". Thus, by evaluating y_1, the GA gathers further information about this schema. Second, crossover introduces representatives of new schemata into the population. In the above example, $y2$ is a representative of the schema #1001###, which is not represented by either parent. If this schema represents a high-performance area of the search space, the evaluation of y_2 will lead to further exploration in this part of the search space.

Termination may be triggered by finding an acceptable approximate solution to $f(x)$, by fixing the total number of evaluations, or some other application-dependent criterion.

The basic concepts of GAs were developed by John Holland [9] and his students [2, 4, 7, 10]. Theoretical considerations concerning the allocation of trials to schemata [4, 9] show that genetic techniques provide a near-optimal heuristic for information gathering in complex search spaces. A number of experimental studies [2, 3, 4] have shown that GAs exhibit impressive efficiency in practice. While classical gradient search techniques are more efficient for problems which satisfy tight constraints (e.g., continuity, low dimensionality, unimodality, etc.), GAs consistently outperform both gradient techniques and various forms of random search on more difficult (and more common) problems, such as optimizations involving discontinuous, noisy, high-dimensional, and multimodal objective functions. GAs have been applied to various domains, including numerical function optimization [2, 3], adaptive control system design [5], and artificial intelligence task domains [12].

2 Installing the System

This section explains if and how GAucsd can be ported to various computing platforms. It assumes that you have successfully obtained and unpacked the GAucsd source distribution into a designated directory. You will see four subdirectories there: **src** contains source code for the GAucsd library and utilities, **usr** has various templates for files users have to write, **etc** contains a few **sh** and **awk** scripts, and **bin** is where the compiled binaries will be installed.

If you want to run GAucsd on multiple platforms, follow the instructions below for each type of machine in succession. Since this version of GAucsd uses machine-dependent subdirectories for binaries, you can compile for multiple target platforms from a single copy of the source distribution on a shared file system.

2.1 Requirements

GAucsd has been developed in a Unix environment, but should be easy to port to any platform that has a C compiler and the **make** and **awk** utilities. GNU versions of **cc**, **make** and **awk** are available as source distributions and also as binaries free of charge for a variety of machines.

Note that since **make** and **awk** are called as such from deep within the GAucsd package, they must be available as commands by that name. Thus if you are using GNU's **gawk** for instance, you must define **awk** as a link or alias for **gawk** on all target platforms. Also make sure that make and awk are both in your command search path.

GAucsd uses the names **inset**, **report**, and (on Unix systems) **ga** & **gx** for its own commands; if one of these clashes with some other command on your system, you have to take steps to resolve the name conflict. The **gx** command available on Unix platforms uses "mail $USER" to notify users of completed experiments; if this won't work on the target machine, modify the mailing address in file **etc/gx** accordingly, or remove the **mail** command altogether.

2.2 Customizing the Makefile

The top portion of the file **src/Makefile** needs to be customized for each target platform. For many platforms this will be the only modification necessary; on most Unix systems the supplied makefile will work without any changes. On each successful installation, a copy of the makefile will be saved along with the binaries.

Since binaries are installed in machine-dependent subdirectories, the makefile needs the variable **MACH** set to a directory name that is unique to each target platform. The default is set to the output of the **mach** command, which on many UNIX systems returns the machine type. If some of your target platforms do not have a **mach** command, you may consider writing one, although often it will be easier to just manually set MACH to a suitable name for each target platform.

Next, you should set **CFLAGS** to whatever compiler flags you wish to set on this particular platform. In particular, you should add `-DNOGAGX` to **CFLAGS** for platforms where you can't (or don't want to) use the **ga** and **gx** utilities for distributed computing, which work only in certain types of UNIX environment.

Further macros that may need to be customized deal with the command shell's behavior, file naming conventions, file handling and compilation commands on the target platform; all of these are documented in the makefile itself.

2.3 Building GAUCSD

Define the environment variable **GAUCSD** to be the full pathname to the directory you unpacked the GAUCSD source distribution in. From there, change into the **src** subdirectory and type "make all". If all goes well, type "make install", then "make clean". If you get errors due to missing include files, check the top section of file **define.h** for details on how to work around these problems. Finally, communicate to users where you have installed GAUCSD so that they can set the **GAUCSD** variable in their environment.

3 Major Procedures

This section gives a more detailed description of the genetic algorithm implemented in the GAUCSD library.

3.1 Initialization

File **init.c** contains the initialization procedure, whose primary responsibility is to set up the initial population. If you wish to seed the initial population with heuristically chosen structures, put the structures in the **ini** file (see Section 11) and use the **i** option (see Section 10). The rest of the initial population is filled with random structures, or by a reduced-variance technique for super-uniform initialization if the **u** option is used.

3.2 Generation

As previously mentioned, one generation (see **generate.c**) comprises the following steps: selection, mutation, crossover, evaluation, and some data collection procedures.

3.3 Selection

Selection is the process of choosing structures for the next generation from the structures in the current generation. The selection procedure (see file **select.c**) is a stochastic procedure that guarantees that the number of offspring of any structure is bounded by the floor and the ceiling of the (real-valued) expected number of offspring. This procedure is based on an algorithm by James Baker. The idea is to allocate to each structure a portion of a spinning wheel proportional to the structure's relative fitness. A single spin of the wheel assigns the number of offspring to all structures. This algorithm is compared to other selection methods in [1]. The selection pointers are then randomly shuffled, and the selected structures are copied into the new population.

3.4 Mutation

After the new population has been selected, mutation is applied to each structure in the new population (see **mutate.c**). Each position is given a chance **M_rate** of undergoing mutation. This is implemented by computing an interarrival interval between mutations, assuming a mutation rate of **M_rate**. The **mutation** macro in **define.h** determines what happens if mutation does occur; the default action is to flip the bit value for that position. The mutated structure is then marked for evaluation.

3.5 Crossover

Crossover (see **cross.c**) exchanges alleles among **C_rate * Popsize** adjacent pairs of structures in the new population. Note that a **C_rate** greater than 1.0 will cause some structures to undergo several crossovers. Crossover might be implemented in a variety of ways, but there are theoretical advantages treating the structures as rings, choosing two crossover points, and exchanging the sections between these points [4]. The segments between the crossover points are exchanged, provided that the parents differ somewhere outside of the crossed segment. If, after crossover, the offspring are different from the parents, then the offspring replace the parents, and are marked for evaluation.

4 Fitness Scaling

When minimizing a numerical function with a GA, it is common to define the performance value $u(x)$ — the expected number of offspring — of a structure x as $u(x) = F - f(x)$, where F is a large baseline function value. Negative values of $u(x)$ can either be zeroed or avoided altogether by setting F to f_{max}, the maximum value that $f(x)$ can assume in the given search space. Often f_{max} is not available a priori, in which case we may use $F = f(x_{max})$, the maximum value of any structure evaluated so far.

Either choice of F has the unfortunate effect of making good values of x hard to distinguish. For example, suppose $f_{max} = 100$. After several generations, the current population might contain only structures x for which $5 < f(x) < 10$. At this point no structure in the population has a performance which deviates much from the average. This reduces the selective pressure towards better structures, and the search stagnates. One solution is to update the baseline to, say, $F = 15$, and rate each structure against this new standard.

The problem is then to automate the baseline update in a manner that keeps the amount of selective pressure under control. GAucsd provides two alternative fitness scaling algorithms for this task: *window scaling* and *sigma scaling*.

Window scaling allows the user to control how aggressively the baseline is updated via the *scaling window size* W. If $W > 0$, the system sets F to the greatest value of $f(x)$ which has occurred in the last W generations. A value of $W = 0$ indicates an infinite window, i.e. $F = f(x_{max})$. Note that this method is overly attentive to individuals in that a single "lethal" genotype can all but eliminate selective pressure for W generations.

Sigma scaling (studied by Forrest [6]) achieves more robust performance by setting F to the average population fitness plus a certain multiple, the *sigma scaling factor* s, of the standard deviation of population fitness; structures worse than F are assigned zero performance. Note that for a structure x with $f(x)$ one standard deviation better than the population average, $u(x) = (s + 1)/s$; sigma scaling thus provides very direct control over the selection pressure. Values for s between 1 and 5 are commonly used in practice.

5 Dynamic Parameter Encoding

When encoding real-valued parameters of the evaluation function on a binary genotype there is a conflict between the desire to keep the genes short for good convergence and the need to know the result with a certain precision. An appropriate — but cumbersome — reaction when faced with this dilemma would be to first run an experiment with short genes to quickly obtain a low-precision result, then repeating it with ever-increasing precision while keeping the genotype short by restricting the search to the previously identified solution region.

Dynamic Parameter Encoding (DPE) [11] implements this strategy of iterative refinement by gathering convergence statistics of the top two bits of each parameter. Whenever the population is found to be converged on one of three subregions of the search interval for a parameter, DPE invokes a *zoom operator* that alters the interpretation of the gene in question such that the search proceeds with doubled precision, but restricted to the target subinterval. In order to minimize its disruptiveness the zoom operator preserves most of the phenotype population by modifying the genotypes to match the new interpretation.

The DPE algorithm logs its zoom activity in the **dpe** file (see Section 11).

Since the zoom operation is irreversible it has to be applied conservatively in order to avoid premature convergence; to this end DPE smoothes its convergence statistics through exponential historic averaging. The time constant of this filtering process is an important characteristic of the algorithm: the smaller its value, the bolder DPE becomes, accenting the risks and benefits associated with fast convergence.

Note that although DPE often facilitates a radical reduction of gene length, there is a point beyond which the function to be optimized will no longer be sampled with enough resolution to yield useful results. In particular if the basin of attraction around the optimum is small, a low-resolution search might miss it altogether. Of the five test functions provided in the **$GAUCSD/usr** subdirectory for instance, four can be solved with DPE using as little as three bits per parameter, but the multimodal function **f5** requires twice as much.

The DPE algorithm is activated by selecting a non-zero smoothing time constant in the **inset** program; it may be selectively disabled for certain parameters via a **b** or **g** flag in the **GAeval** comment line (see Section 7). Since DPE is based on strong assumptions about the interpretation of the genome it is meant to be used in conjunction with a high-level evaluation function as facilitated by the **wrapper** described below.

This quick overview was intended to encourage experiments with the DPE algorithm; many aspects have been somewhat glossed over. For a more detailed discussion please refer to [11].

6 Evaluation Procedure

To use GAucsd, the user must write an evaluation procedure. There are three levels of abstraction at which such a procedure may be written. At the lowest level a function called **_eval()** receives a pointer to the genome and its length in bit as input and returns a double precision value. It must be declared as

```
double _eval(genome, length)
char genome[];
int length;
```

The interpretation of the genome is entirely up to the user, thus allowing great flexibility and efficiency. However, the packed form of the genotype can be awkward to deal with if the parameters are not aligned with byte boundaries. Therefore an evaluation function **eval()** may be declared instead which receives an unpacked genotype:

```
double eval(buff, length)
char buff[];
int length;
```

where **buff** is a character array containing (integer) zeroes and ones, and **length** indicates the length of the array **buff**. This form of evaluation function was used in the original GENESIS software and is assisted by some functions which facilitate its interpretation. One is called **Ctoi()**:

```
double Ctoi(buf, length)
char buf[];
int length;
```

and takes two arguments, a pointer to a char array and a length indicator. **Ctoi()** returns the value computed by interpreting the buffer as an unsigned binary string with the indicated length. If the **A** option is used, **Ctoi()** will add a random fractional part to the value in order to avoid aliasing effects that might otherwise compromise the search of continuous spaces (see Section 10).

Gray codes are often used to represent integers in genetic algorithms. They have the property that adjacent integer values differ at exactly one bit position; their use avoids the unfortunate effects of *Hamming cliffs* in which adjacent values, say 31 and 32, differ in every position of their fixed point binary representations (01111 and 10000, respectively). Functions which translate between Gray code and fixed point representations are provided:

```
Gray(inbuf, outbuf, length)
char *inbuf, *outbuf;
register int length;

Degray(inbuf, outbuf, length)
char *inbuf, *outbuf;
register int length;
```

In the function **Gray()**, **inbuf** contains the fixed point integer, one bit per char, while **outbuf** gets the Gray coded version, one bit per char. In **Degray()**, **inbuf** contains the Gray code integer and **outbuf** gets the corresponding fixed point integer.

A procedure **Error()** is provided which writes an error message to both the file **errors** and the standard error output (**stderr**), and then terminates the program. Functions for re-encoding, packing and unpacking genomes are also provided — see the files **encode.c** and **decode.c** in **$GAUCSD/src** for details.

Figure 2 shows a low-level evaluation function which makes use of these GAUCSD procedures. Note how a call to the evaluation function with negative **length** parameter is used by GAUCSD to ask for a phenotypic description of the most recently evaluated structure. This description is provided automatically if you use the **wrapper** (see Section 7) and will be printed in the **min** file (see Section 11).

```
/****************************************  file f1.c  ****/

double _eval(genome, length)
char genome[];
int length;
{
    register int i;
    char buff[30];
    char outbuf[10];
    double sum = 0.0;

    /* phenotype description, must be static */
    static double x[3];

    /* return previous phenotype on request  */
    if (length < 0)
        sprintf(genome, "\n%lf %lf %lf", x[0], x[1], x[2]);
    else
    {
        /* GAlength 30 */
        if (length != 30) Error("length error in eval");

        /* unpack the genotype */
        Unpack(genome, buff);

        for (i = 0; i < 3; i++)
        {
            /*  convert next 10 bits to an integer  */
            Degray(&buff[i*10], outbuf, 10);
            x[i] = Ctoi(outbuf, 10);

            /*  scale x to the range [-5.12, 5.11]  */
            x[i] = (x[i] - 512.0) / 100.0;

            /*  accumulate sum of squares of x's  */
            sum += x[i]*x[i];
        }
    }
    return(sum);
}

***************************************** end of file ****/
```

Figure 2: A Low-level Evaluation Function.

A comment of the form `/* GAlength` *l* `*/` (as shown in Figure 2) is recognized by the **inset** program and used to set the default value of the "Genome Length" parameter. It may safely be omitted but is provided automatically by the **wrapper**.

It is often desirable to pass parameters to the evaluation function that might vary from experiment to experiment but should not be subjected to the GA search. GAucsd uses a method similar to that of passing C command line arguments to make such *application-specific* parameters, entered via the **inset** program, accessible through the declarations:

```
extern int   GArgc;
extern char *GArgv[];
```

which, again, the **wrapper** provides for you. Note that each of the **GArgc** string parameters in **GArgv[ ]** may contain blank spaces but not '\0' or '\n'.

If you are writing evaluation functions as shown in this section, GAucsd leaves the interpretation of the genome entirely up to you. While this affords great flexibility, it also means that the DPE technique — which assumes a known layout of gray-coded parameters on the genome — cannot be used unless additional information about this layout is provided. This can be done by declaring and initializing the global variables **GAgenes, GAposn, GAbase** and **GAfact** in the evaluation file. For further details, consult the sample file **f1_ga.c** in **$GAUCSD/usr**.

7 The Wrapper

GAucsd includes an **awk** script called **wrapper** which provides a higher level of abstraction: by supplying the code for decoding and printing the evaluation function parameters automatically, it allows the direct use of most C functions as evaluation functions. The only restrictions are:

- the function must not be called **_eval()**;
- it must return a scalar type or a pointer to such a type;
- all its parameters must be simple C types as described below, or pointers to such types (this allows for passing arrays by reference).

The wrapper gets invoked from the **inset** program and constructs from <*name*>**.c** the file <*name*>**_ga.c** which includes a function **_eval(gene, length)** interfacing your evaluation function to the GAucsd system. In order to do its job the wrapper needs a comment line (occurring *after* the first declaration of your evaluation function) in <*name*>**.c** which looks as follows:

```
/* GAeval <fn> <field1> <field2> ...  */
```

where <*fn*> is the name of your evaluation function, possibly prefixed with an asterisk for indirect return values. It is followed by one or more fields, where each field specifies a parameter to the evaluation function. White space delimits fields and hence may not occur within fields. The format of an individual field is (in this order):

1. an integer indicating the number of bits to be used for representing this parameter on the genotype. This must be between 1 and the number of bits of an "int" on your machine to make sense.

2. (optional) a colon followed by a number r specifying the range of the parameter. This means that the parameter will range from $-r$ (or zero if unsigned — see below) inclusive to $+r$ exclusive. If omitted, the range is determined directly from the number of bits used to represent the parameter. A second number s, separated by a colon, may be specified, forcing a range from r inclusive to s exclusive. Both r and s may contain a decimal point and a sign, but no exponent, and s must be strictly greater than r.

3. a character string containing in any order, in upper or lower case:

   - exactly one of **c, s, i, l, f** or **d**, specifying the parameter type as char, short, int, long, float or double respectively;
   - (optional) a **b** or **g** indicating that the parameter is to be encoded in binary or Gray code, respectively. Either character causes the parameter to be left alone by the **DPE** algorithm which relies on the default Gray coding for its operation.
   - (optional) a **u** indicating that the parameter is unsigned. For float or double parameters the type will not change, but the default range will be from zero to r instead of $-r$ to r (see above).

4. (optional) an integer n indicating replication: the parameter is a pointer to an array of n values of the same format. Values of 1 (simple indirection) or 0 (same as no n at all) for n are allowed.

Space for parameters on the genotype is allocated from the left in order of the fields. Figure 3 demonstrates how the evaluation function of Figure 2 is greatly simplified when the wrapper is used. The function shown is the first of a suite of five test functions that have been used extensively in the GA community since their introduction by De Jong [4]; all five can be found in the **$GAUCSD/usr** subdirectory.

If **awk** is not available on your system, you will not be able to use the wrapper. You can emulate its operation manually by writing your low-level evaluation function in a file ending with **_ga.c** and including the additional data the wrapper normally provides. Please refer to the sample wrapper output file **f1_ga.c** in **$GAUCSD/usr** for further details.

```
/****************************************  file f1.c  ****/

double f1(x)
register double *x;
{
    register int i;
    register double sum;

    /*  accumulate sum of squares of x's  */
    for (sum = 0.0, i = 0; i < 3; i++)
        sum += x[i]*x[i];
    return (sum);
}

/* GAeval f1 10:5.12d3 */

/**************************************** end of file ****/
```

Figure 3: Same Evaluation Function using the Wrapper.

8 Setting up Experiments

In order to use GAucsd, you must set the variable **GAUCSD** in your environment to the full pathname of the directory where GAucsd was installed on your system. You also have to add the directory where the GAucsd binaries for your system are located — a subdirectory of **$GAUCSD/bin** whose name reflects the machine type — to your command search path. On Unix systems, the directory **$GAUCSD/etc** should be added to the command search path as well. You may want to put the commands that achieve this into a file that gets executed automatically whenever you use the system, such as **.cshrc** under Unix or **AUTOEXEC.BAT** under DOS.

The easiest way to write an evaluation function for GAucsd to optimize is to copy one of the examples provided in **$GAUCSD/usr** and use it as a template. Once you have the desired evaluation function in your current directory, run the **inset** program to construct an **in** file. As **inset** prompts for various GA parameters, hitting return in response results in the default value shown in brackets being used. Many defaults (indicated with a * below) are computed on the fly by **inset** from previously entered data.

It is important to keep in mind that while these "dynamic defaults" are a nice feature, the heuristics employed by no means guarantee a good, nor indeed even adequate setting of GA parameters. The research on how to find good values for them is still in its infancy, and as of today there are no strong results that could replace the intuitive exploration of this parameter space by the experimenter.

The parameters set via **inset** are:

- Evaluation File Name [f1]:

  At this point **make** is called to preprocess, compile and link the appropriate files. If the wrapper aborts with an error, examine the **_ga.c** file for diagnostics.

- Name of Experiment [*]:

  The basename for all files associated with this experiment (see Section 11); it defaults to the name of the evaluation file. If an **in** file with the chosen basename exists already, it will be read at this point and used as default for subsequent prompts. If the existing **in** file is read-only, you will be asked to provide an alternative basename for writing — thus **inset** may be used to re-edit existing **in** files, or to make modified copies from a read-only master file. If there is no appropriate **in** file, **inset** will create it and try to guess reasonable defaults as described above.

- Genome Length [*]:

  If there is a comment of the form `/* GAlength` *l* `*/` — as produced automatically by the wrapper — in the evaluation file, the default length suggested will be *l*.

- Population Size [*]:

  The number of structures in the population.

- Trials per Run [*]:

  The maximum number of function evaluations in each GA run. When they are computationally expensive, trials are a better indicator of processing time than the number of generations.

- Number of Runs [1]:

  The number of independent optimizations of the same function in this experiment; multiple runs can increase the chance of finding a good solution.

- Crossing Rate (per individual) [*]:

  The expected number of two-point crossovers for each structure.

- Mutation Rate [*]:

  The expected number of mutations for each bit in the population.

- Generation Gap [1.0]:

  The generation gap is the percentage of the population which is replaced in each generation. Note that GAucsd operates very inefficiently for small generation gaps.

- Windowsize [-1]:

  The size of the scaling window (see Section 4). Zero indicates infinite size, and any negative value indicates sigma scaling.

- Sigma Scaling Factor [2.0]:

  Used only when sigma scaling is chosen.

- DPE Time Constant [0]:

  This is the time constant (in generations) with which the DPE algorithm smoothes its convergence statistics through exponential historic averaging in order to avoid premature convergence. A value of zero switches DPE off altogether.

- Convergence Threshold [*]:

  The percentage of the population that needs to have the same value in a given allele for it to be considered converged. Since it is used as the trigger threshold for the zoom operator, this is an important parameter for DPE. The default value follows an analysis in [11].

- Max Alleles to Converge [*]:

- Maximum Bias [0.99]:

- Max Gens w/o Evaluation [2]:

The three parameters above allow termination of a run when a certain bias is reached, a certain number of alleles have converged, or a certain number of generations has passed without creating a new genome. If any of these conditions is met, the remainder of the run will be "faked" by reprinting the current statistics an appropriate number of times; this simplifies post-processing of GAucsd data e.g. into graphs. The bias check can be disabled with a value of 1.0 or greater, the other two with a value of zero.

- Report Interval [*]:

  The number of evaluations between data collections; zero indicates collections at the start and end of each run only.

- Structures Saved [*]:

  How many of the best structures should be saved to the **min** file.

- Dump Interval [*]:

  The number of generations between data dumps to the **cpt** file; zero indicates that no dumps will be made.

- Dumps Saved [1]:

  The number of dump files that should be kept. One means that only the current dump file is kept; zero indicates that no dumps will be made.

- Options [Aclu]:

  Used to set a variety of options; see Section 10 for details.

- Random Seed [*]:

  The seed value for the random number generator used by GAucsd. Its control allows exact replication of experiments; the default is derived from the system clock.

At this point **inset** writes all settings out to the **in** file and prompts for any application-specific arguments (cf. Section 6). Hitting return will obtain the default read previously from the **in** file, or exit the loop when no default exists.

If GAucsd was compiled with the **-DNOGAGX** flag on your system, **inset** then prints the command that will start the experiment and exit. On some Unix systems, however, it will prompt with `queue []:`. Hitting return in reply will start the program via **ga** in background mode; any other response will queue it in the named file for collective — possibly distributed — execution of a set of experiments via **gx** (see Section 9).

9 Running Experiments

9.1 Direct Execution

A GAucsd program compiled from, say, evaluation file "f1.c" may be run directly on an experiment named "foo" by typing "f1 foo". On systems supporting Unix-style signal handling, you can terminate the experiment prematurely by sending it **INT** or **TERM** signals, which can be generated by pressing key combinations such as `CTRL-C`. The first such signal causes the experiment to conclude after the current run while the second forces a **cpt** dump and exit after the current generation. The third signal causes immediate termination; in this case all progress since the last **cpt** dump will be lost.

9.2 The Report Utility

If the **c** or **C** option is active, the experiment will append a line of data to the **out** file every "Report Interval" trials. To summarize the mean and variance of these measurements over all runs, use the **report** utility: typing `report` *foo* for instance will summarize data file *foo*.**out**.

The summary contains a copy of the **in** file, followed by the means and variances of measurements such as online performance, offline performance, the average performance of the current population, and the current best value. Online performance is the mean of all evaluations; offline performance is the mean of all current best evaluations — see [5].

If option **c** is active, three measures of convergence are also printed: "Conv" is the number of positions which have converged at least to the chosen threshold, "Lost" is the number of those which have converged 100% (i.e. the entire population has the same value), and "Bias" indicates the average percentage of the most prominent value in each position. For instance, a bias of 0.75 means that on average each position has converged to either 75% zeroes or 75% ones.

9.3 Remote Execution via ga

On UNIX systems, the command `ga f1 foo &` will run the same experiment as `f1 foo`, but at low priority and in the background. **ga** also calls the **report** program if appropriate, and can be used to execute GAUCSD experiments remotely provided you have the necessary permissions in your **.rhosts** file on the remote machine: the command

```
ga f1 foo neuromancer smith /usr/ga &
```

for instance will recompile `f1` on host `neuromancer` in the directory `/usr/ga`. It will then copy **foo.in** (also **foo.ini, foo.out** and **foo.sma** if applicable) into the remote directory, run the program there (using login name `smith`), then copy any resulting data files back into your local directory and produce a report if appropriate.

For binary compatible hosts the directory may be omitted, causing the executable program to be run directly in **/tmp** on the remote host. This eliminates the compilation time and does not require GAUCSD to be installed on the remote host. In this mode the login name defaults to **$USER** if omitted. If the remote machine has direct access to the local directory through a shared file system, specify the remote host's path to it as directory argument: **ga** can exploit this special case to avoid the overhead of copying files between the hosts.

9.4 Distributed Execution via gx

On UNIX systems, you can execute entire sets of experiments sequentially or in parallel on a network of machines. This is done by accumulating experiments in one or several queue files (see Section 8) whose names are then given as arguments to the **gx** script. When all experiments have been completed, **gx** will notify you via **write** or — if that fails — **mail**.

gx distributes experiments to remote hosts specified in the **GAhosts** in the local directory, your home directory or **$GAucsd/usr**. Each entry in the **GAhosts** file consists of a load factor (how many programs will be sent to that host) followed by the remote execution arguments to **ga** as described above

— see the sample **GAhosts** file in **$GAucsd/usr** for details. Any remaining experiments (after the **GAhosts** file has been exhausted) are executed locally.

10 Options

GAucsd allows a number of options which control the kinds of output produced, as well as certain strategies employed during the search. Each option is associated with a single character. The options are indicated by responding to the "Options" prompt in **inset** with a string containing the appropriate characters. If no options are desired, respond to the prompt by typing '**.**'. Options may be indicated in any order. All options may be invoked independently.

a — evaluate all structures in each generation. This may be useful when evaluating a noisy function, since it allows the GA to sample a given structure several times. If this option is not selected then structures which are identical to parents are not evaluated.

A — causes **Ctoi()** to add a random fractional part to its conversion results in order to avoid aliasing problems that might otherwise occur when searching continuous spaces, due to the quantized nature of the genetic encoding. Since this option makes **Ctoi()** stochastic, **A** automatically implies **a**.

b — at the end of the experiment, write the average best value (over all runs) to the standard output.

c — collect statistics concerning the convergence of the algorithm. These statistics are written to the **out** file every "Report Interval" trials. The intervals are approximate, since statistics are collected only at the end of a generation. Option **c** implies **C** but is computationally more expensive.

C — collect statistics concerning the performance of the algorithm. These statistics are written to the **out** file every "Report Interval" trials. The intervals are approximate, since statistics are collected only at the end of a generation.

d — dump the current population to **cpt** file *after each evaluation*. This may considerably slow down the program, and is only useful when each evaluation represents a large amount of computation.

e — use the *elitist* selection strategy. The elitist strategy stipulates that the best performing structure always survives intact from one generation to the next. In the absence of this strategy, it is possible that the best structure disappears due to crossover or mutation.

i — read structures from the **ini** file into the initial population. If the file contains fewer structures than the population needs, the remaining structures will be initialized randomly, or super-uniformly if the **u** option is used.

l — log activity (such as starts and restarts) in the **log** file. Some error messages also end up in the **log** file.

L — dump the last generation to the **cpt** file. This allows the user to extend the experiment at a later time, using the **r** option.

o — at the end of the experiment, write the average *online performance* to the standard output. Online performance is the average of all evaluations.

O — at the end of the experiment, write the average *offline performance* to the standard output. Offline performance is the average of the "best value so far" over the course of the run.

r — restart a previously interrupted execution. In this case, the **cpt** file is read back in, and the GA takes up where it left off.

s — trace the history of one schema. This options requires that a **sma** file exists in which the first line contains a string which has the length of one structure and which contains only the characters '0', '1', and '#' (and no blanks). The system will append one line to the schema file after each generation describing the performance characteristics of the indicated schema (number of representatives, relative fitness, etc.).

t — trace each major function call; only useful for debugging purposes. Tracing statements are written to the standard output.

u — create a super-uniform initial population in which all schemata up to a certain defining length (limited by the population size) are equally represented. In crossover-dominated GAs (i.e. those with low mutation rate) this eliminates the risk of pathological initial populations in which an important low-order schema just happens to be missing, and has to be created by an unlikely mutation event. The **u** option uses a reduced-variance stochastic algorithm which produces a population with no local, but large global correlations. Crossover is very effective in destroying such long-range correlations, but this option should not be used in mutation-dominated GAs with very low crossover rates.

11 Files

For any the file names listed below, you may create a subdirectory in which these files are collected. The output of an experiment with name "foo", for instance, will be in the file **foo** in the subdirectory **out** if that exists, in the file **foo.out** in the current directory otherwise. There is also a file "errors" in which GAucsd error messages are collected.

cpt — a checkpoint file containing a snapshot of important variables, and the current population. This file is produced if the **d** option is set, the second termination signal is received, or both the number of saved dumps and the dump interval are positive. This file is necessary for the restart option **r** to work, but can also be interesting in its own right.

dpe — this file, produced when the DPE algorithm is used, logs the activity of the zoom operator. For each zoom one line is appended, containing generation and trial number, the index of the zoomed parameter (starting at zero), the endpoints of its new search interval, and its new precision.

in — contains all input parameters. This file is required.

ini — contains structures which will be included in the initial population. This is useful if you have heuristics for selecting plausible starting structures. This file is read if the **i** option is set.

log — logs the dates of starts and restarts; produced if the **l** option is set.

min — contains the best structures found by the GA. Each paragraph of the **min** file displays a binary structure, its evaluation, and the generation and trial counters at the time of the first occurrence of this structure, followed by a phenotypic description of the structure as provided by the evaluation function. The number of elements in **min** is indicated by the response to **inset**'s "Structures Saved" prompt. If the number of runs is greater than one, the best structures are stored in **min.**n during run number n.

out — if the **c** or **C** option is set, a line of data describing the performance of the GA is appended to this file every "Report Interval" trials. From left to right, the data columns are: generations, trials, lost and converged alleles, bias, and online, offline, best and average performance.

rep — produced by **ga** (via the **report** program) from the **out** file, this file summarizes the performance of the GA. It consists of a copy of the **in** file followed by the mean and variance of the **out** file data averaged over all runs.

sma — logs a history of a single schema. This file is required for the **s** option; its first line must contain the schema in question and is read at the start of the experiment. A line of data describing the schema's performance is then appended each generation (cf. option **s**).

12 Making Modifications

GAucsd was designed to encourage experiments with genetic algorithms. It is relatively easy for the user to create variations of GAucsd. Suppose for example that you wish to test a new crossover operator. Simply copy the file **$GAUCSD/src/cross.c** into your own directory and modify it as desired.

Now copy the GAucsd **Makefile** from the appropriate subdirectory of **$GAUCSD/bin** into your directory and add "cross.o" (or "CROSS.OBJ" under DOS) to the **LOBJS** macro at its top. Now when you run **inset**, or compile your experiment manually, the loader will include your crossover function instead of the one provided in the GAucsd library. Recompilation of the library is thus not necessary.

In order to facilitate such experimentation, most of the important variables in GAucsd are global. All global identifiers in **$GAUCSD/src/global.h** begin with a capital letter to minimize conflict with user-defined identifiers.

Acknowledgments

John Grefenstette wishes to thank the early users of Genesis for their suggestions and comments, especially Ray Ford, Jeremy Norton and Mike Fitzpatrick. Nicol Schraudolph is indebted to John Grefenstette for providing with the Genesis package such a useful basis for modifications, and a User's Guide that forms the backbone of this manual. Special thanks go to all the users of GAucsd, who with countless bug reports and suggestions have been vital in shaping this package. Further suggestions and comments are welcome.

References

[1] James E. Baker. Reducing bias and inefficiency in the selection algorithm. In John J. Grefenstette, editor, *Proc. 2nd Int. Conf. Genetic Algorithms and their Applications*, pages 14–21, Hillsdale, NJ, 1987. Lawrence Erlbaum Associates.

[2] A. D. Bethke. *Genetic Algorithms as Function Optimizers.* PhD thesis, Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, MI, 1981.

[3] A. Brindle. *Genetic Algorithms for Function Optimization.* PhD thesis, Computer Science Dept., Univ. of Alberta, 1981.

[4] Kenneth A. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, MI, 1975. Univ. Microfilms No. 76-9381.

[5] Kenneth A. De Jong. Adaptive system design: a genetic approach. *IEEE Trans. Systems, Man, and Cybernetics*, SMC-10(9):566–574, 1980.

[6] Stephanie Forrest. Documentation for prisoner's dilemma and norms programs that use the genetic algorithm. Technical report, Univ. of Michigan, Ann Arbor, MI, 1985.

[7] D. R. Frantz. *Non-linearities in Genetic Adaptive Search.* PhD thesis, Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, MI, 1972.

[8] David E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[9] John H. Holland. *Adaptation in Natural and Artificial Systems.* The Univ. of Michigan Press, Ann Arbor, MI, 1975.

[10] R. B. Hollstien. *Artificial Genetic Adaptation in Computer Control Systems.* PhD thesis, Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, MI, 1971.

[11] Nicol N. Schraudolph and Richard K. Belew. Dynamic parameter encoding for genetic algorithms. *Machine Learning*, 9:9–21, 1992.

[12] S. F. Smith. Flexible learning of problem solving heuristics through adaptive search. In *Proc. 8th Int. Joint Conf. Artif. Intelligence (IJCAI)*, August 1983.