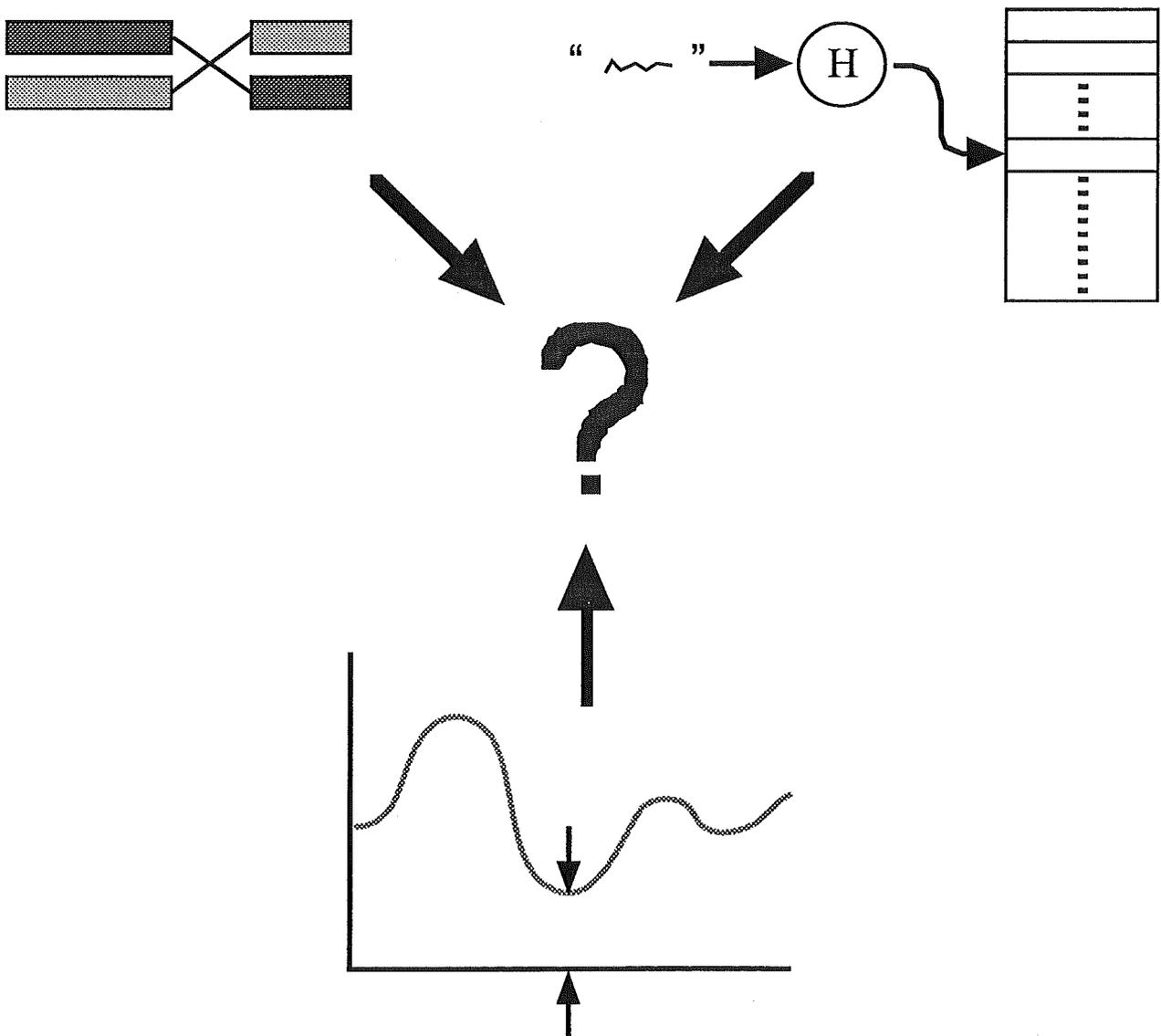


# Optimising Hashing Functions with Genetic Algorithms



## **Abstract**

Genetic algorithms (aka GA's) are a robust global search strategy that ignore local minima and irrelevant parameters, suitable for large search spaces. It is based on an analogy with natural evolution and survival of the fittest. A generation of potential solutions is formed by randomly mating pairs from the previous generation, giving preference to the better performers. By repeating the process many times a (near) optimal solution evolves.

Hashing functions are an implementation for fast table lookup, searching, etc. Given a symbol to store or lookup in a table, a hashing function produces an index into the table, preferably such that all possible symbols will be evenly distributed throughout the table. Their effectiveness is controlled by various parameters such as table size, symbol distribution, and the form of the function itself.

This project aims to couple these two areas together to optimise the parameters of a given hashing function, by searching for a good set of values for the parameters with a genetic algorithm.

## **Aims and Objectives**

The performance of hashing functions are very dependent on their parameters. So the values of the parameters must be chosen carefully. There are no obvious methods for choosing the values. Often the method of choice is a combination of ad hoc with trial and error.

Choice of a hashing function consists of two things. Choice of an algorithm and choice of parameters for the algorithm. My project does not aim to cover the first but rather applying genetic algorithms as an approach to choosing the parameters.

Essential elements consist of software to implement the genetic algorithm, appropriate codings of the parameters and an evaluation function to measure the performance of particular parameters. The genetic algorithm has already been implemented by GENESIS. The evaluation function should be sufficiently general to allow different hashing algorithms and parameter codings to be easily 'plugged in'.

My aim is to produce a system for optimising a hashing function. Before optimisation can be achieved is required a sample of input and the ranges of what parameters that can be varied.

The file containing the sample data should contain each item on a separate line in ASCII format. To improve efficiency, it will be read into memory only once. Obviously larger files will require more processing increasing the run time required for optimisation. The name of the file is passed to the evaluation function as a GENESIS application argument.

To evaluate a genotype, the evaluation function is applied to each of the items from the sample. Statistics are maintained about values produced and the effort required (for estimating speed). After hashing all items the statistics are combined to form penalty score which is returned to GENESIS. By changing the ways the statistics are combined different performance aspects can be emphasised.

## **Genetic Algorithms**

### **Analogy with Evolution**

The concept of genetic algorithms was drawn from the process of natural evolution. So it will be useful to begin by explaining the similarity. In the beginning there was nothing, then there was primordial ooze, followed by dinosaurs, and then

apes, before ending (in a step backwards?) with homo sapiens. How did this happen? The probability of us evolving randomly is mind bogglingly low! ...Or is it?

The essential element is simply "survival of the fittest". When two organisms reproduce they combine their structural information (genes) together to form a new but similar organism. The new combination of genes may gain the child the advantages of both parents. The child will then be more likely to survive, to reproduce itself and continue the species with its superior genes. However if the child gains the disadvantages of both parents, it is less likely to survive, and thus the inferior genes are less likely to continue, effectively removing them from the species.

As time passes the gene pool of the species will become static. The number of new genetic combinations that are possible will become fewer as superior genes begin to dominate, and inferior genes are weeded out. Cosmic rays, dietary chemicals, atomic bombs, etc can cause random mutations within the gene pool, from which novel genetic combinations can be produced. This occasional genetic stirring thus allows further evolution of the species.

Overall the average quality of the species improves with each successive generation. The superior organisms of the population dominating over the inferior. Over the millenia the population evolves to become better and better at living within their environment. The population may split to form separately evolving species, each developing to take advantage of their changing environment in different ways.

### Conceptual Model

With ourselves as evidence, it is obvious that evolution is a very powerful strategy for finding organisms ideally suited to their environments. By analysing the processes that are occurring in evolution, we can construct an algorithm that mimics the powerful search ability of evolution.

At a simplistic conceptual level, a genetic algorithm consists of the sequence shown in figure 1.

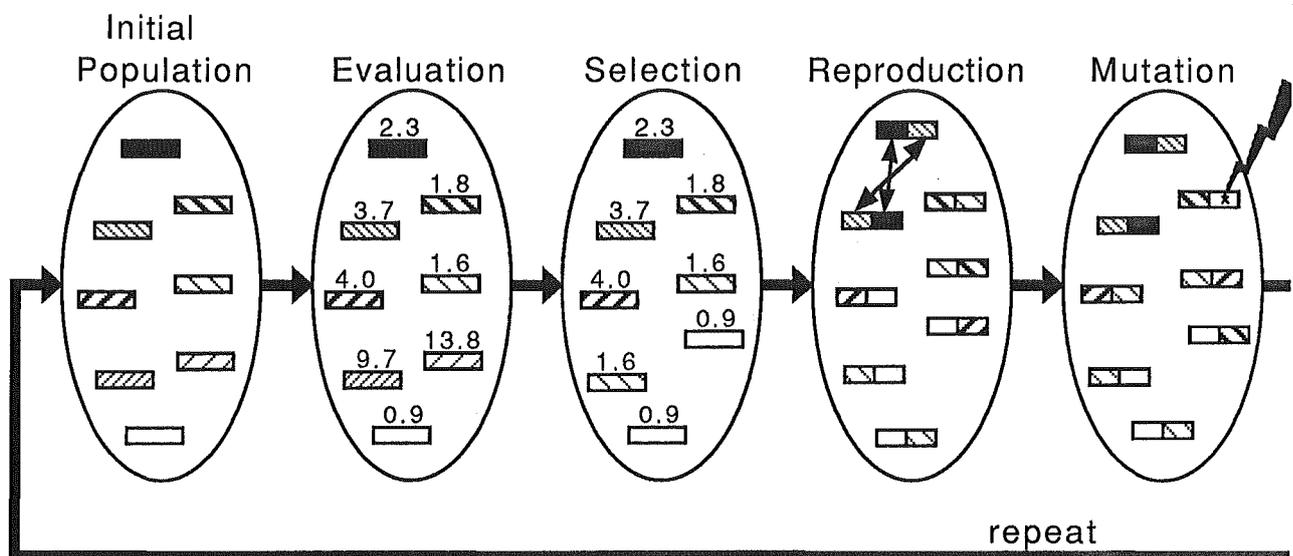


Figure 1. Simple conceptual model of genetic algorithm.

An initial population is chosen at random. Usually a fixed size of population is used to simplify details of the algorithm. In particular, concerns of population explosions or extinction can be ignored. Each member, called a structure, of the population is analogous to an individual organism in evolution.

An evaluation function is applied to each of the structures in the population. The function determines a performance measure of a structures quality, analogous to the ability of an organism to reproduce successfully within the species. To be consistent

with the software I am using, which optimises by minimising the evaluation function, the performance measure is best thought of as a penalty. Smaller values conferring better survival.

A new population is created by choosing structures at random from the old population. Higher probabilities of being chosen are given to the better performers. This produces a population in which the superior structures are strongly represented, while the number of weaker structures is diminished. This is analogous to survival of the fittest for natural evolution.

The structures that have survived to the new population then reproduce. Pairs are chosen at random and mated by swapping portions of information between each other. The crossover of structures closely represents the exchange of information that occurs in biology when fertilisation occurs. Some structures gain the benefits of both parents, some gain the problems of both parents, others find useful new combinations that consist of otherwise irrelevant parameters.

Finally there is an occasional mutation. As said before this prevents the species getting in a deadend alleyway with nowhere else to go. Mutations should not happen too often, which can unnecessarily cause damage to otherwise healthy structures.

Other more complex reproduction operators are available which more closely mimic the processes of reproduction in biology.

### Coding of structures

A structure is best implemented as a binary string, usually of fixed length for simplicity. Variable length structures are sometimes more appropriate, particularly for structures which may contain varying amounts of information, but also for special applications. GENESIS, software which provides most of the functionality of genetic algorithms that I require, only permits fixed length strings. The binary string is analogous to the DNA of cells.

Crossover of pairs of binary strings is easily implemented by simply selecting two random split points and swapping the bits between them across the two structures. This treats the string as being circular, the two ends being regarded as joined. Mutation is also easily implemented as flipping a bit from 1 to 0 or 0 to 1, upon the occurrence of a random event.

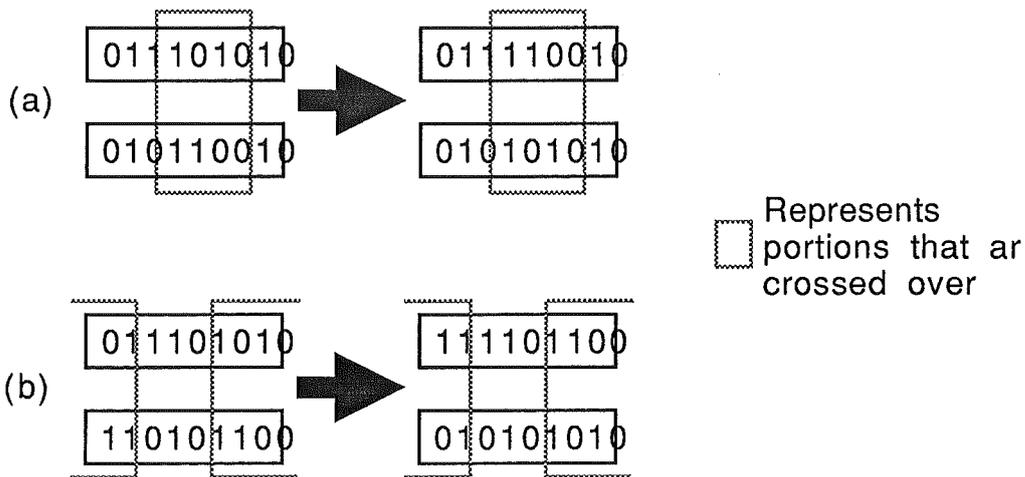


Figure 2. Two examples of crossover.

The internal meaning of the binary string is decoded by the evaluation function, from which it determines the structures performance. No other part of the genetic algorithm requires knowledge of the internal coding of the structure. This means that the design and implementation of the genetic algorithm can be developed independently

from the application. For each application, only the evaluation function needs be written. This feature is exploited by GENESIS to allow the same code to be reused for any application.

The internal coding is up to the programmer and the particular application, but usually it consists of a number of numerical parameters gray coded in binary and concatenated to form a single binary string. Gray coding is very similar to normal binary except that numbers differ in only one bit position between adjacent values. This avoids what are termed hamming cliffs.

It is preferable that the combinations possible from the process of crossover are balanced. For example to go from 128 to 127 requires 8 bit positions to change, but from 128 to 129 requires only one change. With gray coding only one change needs to occur to go from 128 to 127 or 129.

decimal	binary	gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
...	...	...
127	0111 1111	0100 0000
128	1000 0000	1100 0000
129	1000 0001	1100 0001

Correspondence of decimal, binary and gray coding of numbers

### Hashing Functions

In general, a hashing function is used to reduce the range of possible values for its argument to within a smaller range, and to do it quickly. The smaller range is usually an integer to be used as an index into an array called a hash table. A very common application is compilers hashing variable names into symbol tables for fast lookup.

The values produced by the hashing function are not necessarily unique for different argument values. For some applications the function is designed so that unique values are generated for valid values of the argument. Many software utilities already exist to generate what are called perfect hashing functions for applications where the input symbols are known in advance. These software utilities are effective and efficient, thus this project does not aim to cover them.

A collision occurs when a hashing function produces the same value for two different arguments. Many techniques exist for resolving the collision which include:

- rehashing. The hash value is repeatedly hashed again until giving a value not already used.
- separate chaining. The hash table is an array of linked lists.
- linear chaining. If the entry in the hash table is already used, then use the next unused one that follows it.

Hashing functions should be fast. The primary reason for their use is speed. To search an array for an item is an  $O(n)$  algorithm, or  $O(\log n)$  if the array is sorted. Hash tables are nearly  $O(1)$  for hash tables with few collisions. This figure degrades as the number of collisions increases.

The number of collisions can be reduced by ensuring that the distribution of values produced by the hash function are as close to as uniformly distributed as possible. This is the major factor in differentiating between a good hashing function and a bad hashing function. The distribution for a given hashing function may vary for different distributions of input items. For example an input that consists of just numbers is a different input distribution from just uppercase identifiers.

The number of collisions can also be controlled by controlling the utilisation of the hashtable. When the table reaches some threshold ratio of unused locations to total table size, the size of the table might be increased to reduce the frequency of collisions. Likewise if the table empties below some second threshold ratio due to deletions, the table size may be reduced to reclaim the memory for some other use. This technique requires that the hashing function will perform well for many table sizes.

For the purposes of this project optimisation will be defined as finding the set of parameter values for a given hashing algorithm that minimises the time required to hash an item and resolve collisions; ie the time to obtain the entry in the hash table for that item. Finding a flat distribution is implicit in minimising the time for collision resolution.

### **GENESIS**

GENESIS is a complete system for function optimisation using simple genetic search techniques. This was obtained by ftp and has made my project much simpler. Originally I expected to have to implement something similar myself.

All that GENESIS requires is a user defined evaluation function which is to be optimised and the size of the binary strings to be manipulated. Also provided is a utility that generates code to decode the binary strings into parameters for the evaluation function, using a description of the number, type, range of the parameters.

The setup command takes a file containing C source code implementing the evaluation function and compiles it into a program to run the genetic algorithm experiment. The file also contains comments that allows setup to generate interface code that decodes a structure and pushes the values onto the stack as parameters to the evaluation function. Setup also prompts for various arguments and options that control the operation of the genetic algorithm code.

At the end of the experiment a report of statistics for the experiment is produced and a dump of the best performing structures. The report is particularly useful for determining the degree of convergence within the population towards a particular point of the search space.

See the documentation in appendix A for further details.

### **Universal hashing algorithm**

Originally I had hoped to be able to discover an ideal hashing algorithm for a given set of conditions by searching the space of possible machine code programs of some limited length. The binary string of a structure could be interpreted as a sequence of machine instructions and operands for some pseudo machine. For example a disassembled routine and the number of bits estimated to code the instruction might be:

## Optimising Hashing Functions with Genetic Algorithms

```
r0 = 0                                8
:label1
r0 <<= 2                              8+16
r1 = [nextchar]                        8
r0 += r1                               8
r1 <<= 1                              8+4
r0 += r1                               8
if [morechar] branch :label1          8+8
r0 %= [tablesize]                     8+16
```

However the number of bits required to code this I estimate to be about 108 bits. This particular example is a rather compact routine so the length of the structure should be say 200 bits. For a structure length of 200 bits GENESIS suggests a population size of 30000, and total of 6000000 trials. From practical experience, each evaluation I would estimate to require at least 2 seconds. That's a minimum of four and a half months of CPU time for such an experiment (running on a SparcStation doing 28 MIPS). That's a long time to wait before discovering any bugs!

Even executing the genetic algorithm with fewer than suggested number of trials would be beyond my available processing power. So I dropped this approach early in the year. Instead I am going to concentrate on tuning existing hashing algorithms.

With sufficient resources, it would be interesting to try this experiment to discover what develops.

### Optimisation of an iterative accumulation form

A common application of hashing functions is in compilers to store identifiers into a symbol table. Many of them can be generalised to the following form.

```
h0 = 0;
hi = hi-1 * k + ci    for 1 ≤ i ≤ n;
H(c1c2c3...cn) = hn mod tablesize;
```

Where the c's are the characters of the identifier being hashed, k is some constant, and H is the hashing function. The previous form can be further generalised as the following.

```
h0 = 0;
hi = [(hi-1 a x) b (ci d y)] | z    for 1 ≤ i ≤
      n;
H(c1c2c3...cn) = hn mod tablesize;
```

Where x, y, z are constants and a, b, d, | are operators from the set {-, +, /, \*, &, |, ^, "}. -, +, /, \* are normal integer arithmetic operators. &, |, ^ are the C bitwise operators 'and', 'or', 'exclusive or' respectively. " is the 'left' operator which evaluates to its left hand operand. It is included to bring the number of operators up to eight which can be conveniently represented with three bits.

This generalised form offers seven parameters for optimisation; three integer values and four operators. These can be coded as a structure for the genetic algorithm by concatenating the binary (in gray code) representations of each. The integers are represented with 16 bits each, allowing values in the range 0 to 65535 inclusive. The resulting structure is 60 bits in length.

Generally hash tables for compilers will not be larger than 3000. That means at most modulo 3000 arithmetic. The operators -, +, \*, ^, " operating on numbers greater than (or equal to) the modulus have equivalents amongst the numbers less than the modulus. From experience I've found these to be the most useful operators. /, &, | do

not have equivalences but are not such important operators. Also I feel 65535 should be a large enough range to give the operators a chance to be useful.

32 bits would have been just as useful as (if not better than) 16 bits except that the number of trials needed for convergence increases exponentially with the length of the structure. Trading off against that was the observation that larger values for the parameters seemed to give better performers in general. Many present day CPUs are capable of 16 arithmetic, thus the results are reasonably portable.

### Measurement of hashing time

Empirical measurements are better than theory when it comes to the real world. However empirical measurement of speed is not practical where the operations of the hashing function can be optimised for certain values and operations. The obvious example is multiplication by powers of two, which can be optimised as left shifts. The optimisations cannot be performed at compile time as the parameters and operators are not known at that time. Optimisation must occur at runtime for each evaluation of a structure.

To obtain an estimate of the time involved in hashing, a runtime analysis of the operations and their operands must be made. When an operation is interpreted, a number of time units representative of the speed of the operation are added to the total time so far for the evaluation. Operations that may be optimised for special operands add a modified time factor. The timings and optimisations that are being used are:

Operation	Time value
+0, -0, *0, *1	0
+1, -1	4
+n, -n	6
/n	120
*2, *4, *8, *16, *32, *64, *128	8
*n	60
&n, ln, ^n	6

### Measurement of collision resolution time

In measuring the distribution of values produced I have assumed the use of separate chaining to resolve hashing collisions. In most cases extending it for other collision resolution methods simply requires implementing the collision resolution part of the hashing function.

Separate chaining maintains a linked list for each hash location of all items that hash to it. To find an item in the hash table involves hashing it to find its hash location and then scanning the linked list at that location. So the time for collision resolution is the time used to scan the linked list the particular hash location. The average length of a linked list is dependent upon the distribution of hash values produced by the function.

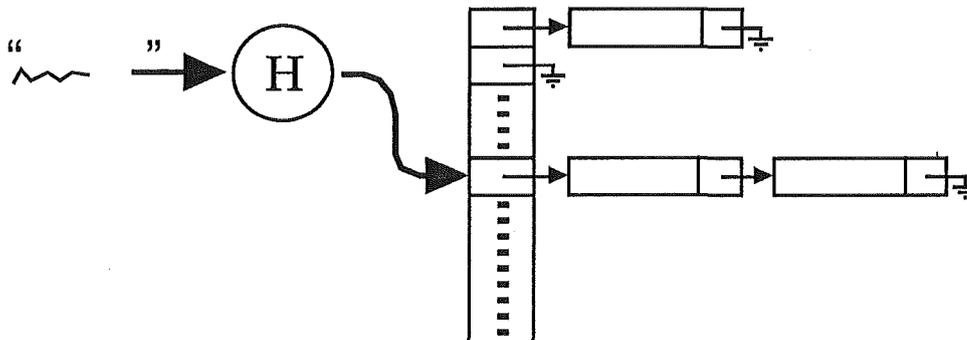


Figure 3. Form of hash table.

The empirical approach to determining the collision resolution time has a number of advantages. It easily allows different distributions of items to hash by specifying a file of sample input. Methods of implementation are obvious. Implementing a theoretical analysis seems to be impractical and non-trivial. It is easily adapted for other hashing algorithms. The only major disadvantage is the long execution time involved.

The method is to hash each of the items read from a representative sample file as if hashing for a specific application. A total of the time used is accumulated as each item is hashed. A table is maintained of the length of the linked list for each hash location. After all the items have been hashed, the performance penalty is the average time to hash each item.

In the early stages of evaluation function development various statistical methods were used to determine the actual distribution of hash locations used. This allowed me to see how effective my evaluation scheme was. Each statistical method produced slightly different results, in particular most methods were dependent on the hash table size. Varying the size of the table generally gave larger tables a worse measure of distribution, even though the same number of identifiers have been hashed to the same shaped distribution. Another (possibly related) effect was the size of the hash table compared to the number of sample items hashed. Variance of the number of hashes to each location seemed to give the best results when the table size was fixed.

The statistical calculation below seemed to give a measure of distribution independent of table size, but did not give as good results as variance when the table size was fixed.

```
sumdiff := 0;
foreach index i
    sumdiff := sumdiff + abs(mean - freq[i]);
flatdistribution := sqrt(sumdiff)/tablesize;
```

Calculating the flat distribution.

### Results

After a run of 240000 evaluations on a population of 2000 structures with a fixed table size of 1403 the best structure decoded as:

$$[(h_{i-1} - 26096) \wedge (c_i \text{ " } k_1)] \text{ " } k_2$$

where  $k_1$  and  $k_2$  are some constants that have no effect due to the " operator. Hence it can be simplified to:

$$(h_{i-1} - 26096) \wedge c_i$$

The fact that the operators /, & and | aren't very useful is not surprising when some consideration is given to their effect on their operands. Integer division evaluates to a disproportionately large number of zeroes. Bitwise and tends to produce values that contain lots of zero bits. Similarly bitwise or tends to produce values with lots of one bits. All three operators share the property of discarding information without using it. The other operators reduce the amount of information but not without using it for some effect on the end result.

Subtraction and addition of a constant value are equivalent in modulo arithmetic. It is merely chance that the genetic algorithm chose one over the other.

It is surprising that multiplication was not amongst the useful operators. Multiplication has the effect of shifting bits left, thus spreading the lower bits throughout the end result. Even though multiplication is an expensive operation, I expected an optimisable multiplication.

## Conclusion

Genetic algorithms are a very powerful search strategy. Although powerful, for particular applications there may be more efficient strategies already available. They are well suited to optimising the parameters of hashing functions, especially as there is no other obvious strategy, except for exhaustive search.

Exhaustive search is not really an option. To try every 60 bit value would require  $2^{60} > 10^{18}$  evaluations. At an optimistic 10 evaluations per second, that's longer than 3 billion years. A simple genetic algorithm might require 500000 trials for a largeish experiment which is about five days at a slow one evaluation per second. Five days is a long time, but for a one off experiment it is acceptable.

With more time I could have made comparisons between various different hashing algorithms.

As an aside. The idea of using genetic algorithms to develop small programs using the approach given under the heading of 'Universal hashing algorithm' is worthy of further research. Variable length structures coupled with appropriate reproduction operators could be used to make the search more efficient.

## References

David E. Goldberg. "Genetic Algorithms in Search, Optimization & Machine Learning". Addison-Wesley, 1989. ISBN 0-201-15767-5.

B.J. McKenzie, R. Harries, T. Bell. "Selecting a Hashing Algorithm", Software-Practice and Experience, Vol. 20(2), pp 209-224, (Feb 1990).

John H. Holland. "Adaption in Natural and Artificial Systems". University of Michigan Press, 1975. ISBN 0-472-08460-7.

Knuth. "The Art of Computer Programming".

John F. Grefenstette. "User's Guide to GENESIS 1.2ucsd". Documentation available with GENESIS.

"GENESIS". Source available via anonymous ftp from the Artificial Life archive server, iuvax.cs.indiana.edu (129.79.254.192) in the pub/alife/software/unix/GAucsd directory.

**Appendix A - GENESIS documentation**

The following pages contain the user manual supplied with GENESIS.

A User's Guide to GENESIS 1.lucsd

written by John J. Grefenstette

Navy Center for Applied Research in Artificial Intelligence  
Naval Research Laboratory  
Washington, D.C. 20375-5000

A User's Guide to GENESIS 1.2ucsd

written by John J. Grefenstette

Navy Center for Applied Research in Artificial Intelligence  
Naval Research Laboratory  
Washington, D.C. 20375-5000

modified by Nicol N. Schraudolph

Computer Science & Engineering Department  
University of California, San Diego  
La Jolla, CA 92093-0114

modified by Nicol N. Schraudolph

Computer Science & Engineering Department  
University of California, San Diego  
La Jolla, CA 92093-0114

ABSTRACT

This document describes the GENESIS 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.

August 14, 1987

Note:

GENESIS 1.2ucsd was derived from GENESIS 4.5 by Nicol Schraudolph at UCSD. It is available via anonymous ftp from the Artificial Life archive server, iuvax.cs.indiana.edu (129.79.254.192) in the pub/alife/software/unix/GAucsd directory. Bug reports and comments on this version should be directed to nici%cs@ucsd.edu. GENESIS 4.5 may be obtained from its author at gref@aic.nrl.navy.mil.

November 14, 1990

1. Introduction

This document describes the GENETic Search Implementation System GENESIS 1.2ucsd. The system was written to promote the study of genetic algorithms for function minimization. GENESIS runs under the UNIX+ operating system, version V7 or higher. 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. The system is written in the language C. Details concerning the interface between the user-written function and GENESIS are explained below. Shell files are provided to ease the construction of genetic algorithms for the user's application.

This section provides a general overview of genetic algorithms (GA's). For more detailed discussions, see [5,8]. GA's are iterative procedures which maintain a "population" of candidate solutions to the objective function  $f(x)$ :

$$P(t) = \langle x_1(t), x_2(t), \dots, x_N(t) \rangle$$

Each structure  $x_i$  in population  $P$  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 the procedure appears in Figure 1.

+ UNIX is a trademark of Bell Laboratories.

```
t <- 0;
initialize P(t); -- P(t) is the population at time t
evaluate P(t);
while (termination condition not satisfied) do
begin
  t <- t+1;
  select P(t) from P(t-1);
```

```

    recombine P(t);
    evaluate P(t);
end;
```

Figure 1. A 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  $f(x_1), \dots, f(x_N)$ .

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  $x_j$  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 recombination operators." The most important recombination 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

$x_1 = 100:01010$ , and

$x_2 = 010:10100$ .

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

$y_1 = 100:10100$  and

$y_2 = 010:01010$ .

- 3 -

Crossover serves two complementary search functions. First, it provides new points for further testing within 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,  $y_2$  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 GA's were developed by Holland 1975 [8] and his students [2,4,7,9]. Theoretical considerations concerning the allocation of trials to schemata [4,8] 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 GA's 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.), GA's 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. GA's have been applied to various domains, including numerical function optimization [2,3], adaptive control system design [5], and artificial intelligence task domains [11]. The next section discusses each of the major modules of this implementation in greater detail.

## 2. Major Procedures

### 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 "init" file (see "Files") and use the 'i' option (see "Options"). 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.

- 4 -

### Generation

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

### 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.

### 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.

### Crossover

Crossover (see "cross.c") exchanges alleles among adjacent pairs of (C\_rate\*Popsize) 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. In the

"setup" program C\_rate is entered in terms of the expected number of crossing points per bit, a measure that directly indicates the disruptiveness of the crossover procedure.

- 5 -

### 3. Evaluation Procedure

To use GENESIS, 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 follows:

```
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) 0's and 1's, and length indicates the length of the array buff. This form of evaluation function was used in the original GENESIS and is assisted by some functions which facilitate the interpretation. One is called Ctoi:

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

which 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 "Options").

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;
```

- 6 -

In the function Gray, "inbuf" contains the fixed point integer, one bit per char. "outbuf" gets the Gray coded version, one bit per char. In Degray, "inbuf" contains the Gray code integer, one bit per char; "outbuf" gets the corresponding fixed point integer.

A procedure Error() is provided which writes an error message to the file "log.error" and to stderr, and then terminates the program. Figure 2 shows an evaluation procedure which uses these GENESIS procedures. Note how a call to the evaluation function with negative "length" parameter is used

```
/****** file fl.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: An Evaluation Function.

- 7 -

by GENESIS to ask for a phenotypic description of the most recently evaluated individual. This description is provided automatically if you use the "wrapper" (see below) and will be printed in the "min" file.

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. GENESIS uses a method similar to that of passing C command line arguments to make such "application-specific" parameters, entered via the "setup" program, accessible through the declarations:

```

extern int  GArgc;
extern char *Gargv[];

```

which the wrapper (see below) provides for you. Note that each of the GArgc string parameters in Gargv[] may contain blank spaces but not '\0' or '\n'.

The Wrapper

This version of GENESIS includes an awk(1) script called "wrapper" which provides a higher level of abstraction: 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 "setup" program and constructs from a file <name>.c the file <name>-ga.c which includes a function "\_eval(gene, length)" which interfaces your fitness function to the GENESIS system. In order to do its job the wrapper needs a comment line (occurring AFTER the first declaration of your fitness function) in

<name>.c which looks as follows:

```

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

```

where <fn> is the name of your fitness 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 fitness 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.

- 8 -

- 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:
  - a) exactly one of 'c', 's', 'i', 'l', 'f' or 'd', specifying the parameter type as char, short, int, long, float or double respectively;
  - b) (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 (see below) which relies on the default Gray coding for its operation.
  - c) (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 format given in 1) - 3). 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. The following figure demonstrates how the evaluation function of Figure 2 is greatly simplified when the wrapper is used:

```

/***** file fl.c *****/
double fl(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 fl 10:5.12d3 */

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

```

Figure 3: Same Evaluation Function using the Wrapper.

#### 4. 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 fast 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 a simulation 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) [10] 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 sub-regions 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 "Files"). 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 included with GENESIS 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 "setup" program; it may be selectively disabled for certain parameters via a 'b' or 'g' flag in the GAeval comment line (see "Wrapper"). Since DPE is based on strong assumptions about the interpretation of the genome it is meant to be used in conjunction with a C-style

evaluation function as facilitated by the wrapper.

This quick overview was intended to encourage and facilitate first experiments with the DPE algorithm; many aspects have been somewhat glossed over. For a more detailed description and discussion of the DPE algorithm and its correct application please refer to [10].

#### 5. Installing the System

Some system tailoring may be necessary when installing GENESIS on a new machine. All of these changes are in the GENESIS source directory.

- 1) If you can't receive mail on the local host, modify the mail address in file "ex" accordingly, or remove the mail command altogether.
- 2) Check the top section of file "define.h" - if you are on a non-standard UNIX system, you may have to modify it.
- 3) If awk(1) is not available on your system, you will not be able to use the wrapper or the "ex" command. To use DPE without the wrapper, you will have to define the global variables GAgenes, GAposn, GAbase and GAfact in your evaluation file, which must end in "-ga.c". Please refer to the sample wrapper output file "fl-ga.c" for further details.
- 4) To compile the system, use the make(1) command:

```
% make all
```

This should compile the programs and create the library "ga.a". This library may then be linked to user-written evaluation procedures as shown below.

#### 6. Setting up Experiments

GENESIS may be set up to run in any directory as follows:

- 1) Copy the Makefile into the current directory:

```
% cp GEN/UserMakefile Makefile
```

where GEN is replaced by the full path name to the GENESIS source directory on your system.

- 2) To get the other essential files into the current directory, use the command:

```
% make ga-install
```

- 3) run "setup", which prompts for the following parameters: (a <cr> response to any prompt gets the default value shown in brackets; a '\*' indicates that the default is derived dynamically from previously entered data)

-- the name of the evaluation file [fl]:

At this point make(1) is called to preprocess, compile and link the appropriate files. If the wrapper aborts with an

error, examine the "-ga.c" file for diagnostics.

- 11 -

-- the suffix for file names [\*]:

The filename extension for this experiment (see "Files"); it defaults to the name of the evaluation file. If an "in" file with the chosen suffix exists already, it will be read at this point to be used as default for subsequent prompts. If the existing "in" file is read-only, you will be asked to provide an alternate suffix for writing - thus "setup" 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, "setup" will create it and try to guess reasonable defaults, with more or less success.

-- the number of experiments [1]:  
(number of independent optimizations of same function)  
-- the length of the structures in bits [\*]:

If there is a comment of the form "/\* GALength <n> \*/" - as produced automatically by the wrapper - in the evaluation file, the length suggested will be <n>.

-- the population size [\*]:  
-- the number of trials per experiment [\*]:  
-- the rate of crossing points per bit [\*]:  
-- the mutation rate [\*]:  
-- the generation gap [1.0]:

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

-- the scaling window [-1]:

When minimizing a numerical function with a GA, it is common to define the performance value  $u(x)$  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 selection 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 scaling window  $W$  allows the user to control how often the baseline performance is updated. 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, ie.  $F = f(x_{max})$ . This window scaling method is

- 12 -

unfortunately overly attentive to individuals in that a single "lethal" genotype can all but eliminate selective pressure for  $W$  generations. A more robust method studied by Forrest [6], which we call "Sigma Scaling", is now available with GENESIS, and can be accessed by setting  $W < 0$ .

-- the sigma scaling factor [2.0]:

In sigma scaling,  $F$  is set to the average population fitness plus a certain multiple, the sigma scaling factor  $s$ , of the standard deviation of population fitness. (Individuals worse than  $F$  are assigned zero performance.) Note that for an individual  $x$  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 have been used in practice.

-- the smoothing time constant for DPE [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.

-- the 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 suggested by "setup" follows an analysis in [10].

-- how many alleles must converge to end the experiment [\*]:  
(0 indicates that no such check will occur)  
-- how large the bias must be to end the experiment [0.99]:  
-- how many consecutive generations without any evaluations occurring will end the experiment [2]:  
(0 indicates that no such check will occur)

If one of the above three termination conditions is met, the remainder of the experiment will be faked by reprinting the current statistics an appropriate number of times.

-- the number of trials between data collections [\*]:  
(0 indicates collect at start and end of experiment only)  
-- how many of the best structures should be saved [\*]:  
-- the number of generations between dumps [\*]:  
(0 indicates no dumps will occur)  
-- the number of dumps that should be saved [1]:  
(0 indicates no dumps will occur)  
-- the options (see chapter 7) [cel]:  
-- the seed for the random number generator [\*]:

At this point setup writes all settings out to the "in" file and prompts for application-specific arguments (see chapter 3). Hitting return will get you the default read previously from the "in" file, or exit the loop when no default exists.

- 13 -

Setup then prompts with "queue [ ]:". A <cr> in reply starts the program in background mode, any other response queues it in the named file for collective - possibly distributed - execution of a set of experiments (see chapter 8).

## Files

For any the file names listed below, you may create a directory in which these files are collected. The report for an experiment with filename extension "foo", for instance, will be in the file "foo" in the directory "report" if it exists, in the file "report.foo" in the current directory otherwise. In either case the "clean" command removes all files with a given extension. There is also a file "log.error" in which GENESIS error messages are collected.

"ckpt" - 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.

"init" - 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 iff the option 'i' is set.

"log" - logs the dates of starts and restarts. This file is produced if the 'l' option is set.

"min" - contains the best structures found by the GA. The number of elements in "min" is indicated by the response to the "save how many" prompt during setup. If the number of experiments is greater than one, the best structures are stored in "min.n" during experiment number n. This file is produced if the number of saved structures is positive.

"out"- contains data describing the performance of the GA. This file is produced if option 'c' is set.

"report" - produced by the report program from the "out" file, this file summarizes the performance of the GA.

"schema" - logs a history of a single schema. This file is required for the 's' option.

- 14 -

## 7. Options

GENESIS 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 with a string containing the appropriate characters. If no options are desired, respond

'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' always implies 'a'.

'b': at the end of the experiments, write the average best value (over all experiments) to the standard output.

'c': collect statistics concerning the convergence of the algorithm. These statistics are written to the "out" file, after 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, after 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 "ckpt" file AFTER EACH EVALUATION. WARNING: This may considerably slow down the program. This may be useful when each evaluation represents a significant 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, thanks to crossover or mutation.

'i': read structures into the initial population. The initial population will be read from the "init" file. 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.

- 15 -

'l': log activity (starts and restarts) in the "log" file. Some error messages also end up in the "log" file.

'L': dump the last generation to the "ckpt" file. This allows the user to restart the experiment at a later time, using option 'r'.

'o': at the end of the experiments, write the average online performance measure to the standard output. Online performance is the average of all evaluations during the experiment.

'O': at the end of the experiments, write the average Offline performance measure to the standard output. Offline performance is the average of the best current values.

to the prompt by typing '.'. Options may be indicated in any order. All options may be invoked independently.

'r': restart a previously interrupted execution. In

Oct 4 15:15 1991 root:/tmp/3607.lwf\_tmp Page 8

this case, the "ckpt" 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 file named "schema" exist 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 - FOR DEBUGGING. 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 (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, where crossover rates are too low.

- 16 -

## 8. Running the Programs

A GENESIS program with, say, evaluation file name "fl" and file name extension "foo" may be started directly by typing "ga.fl foo". In most cases, however, it is preferable to use the "go" or "ex" shell scripts instead (see below). You can terminate a GENESIS program prematurely by sending it "TERM" signals using the kill(1) command. The first such signal causes the program to exit after the current experiment is completed; the second forces a "ckpt" dump and immediate termination.

The command "go ga.fl foo &" will run the same program at low priority in the background and then call the "report" program if appropriate (see below). "go" can also be used to execute a GENESIS program remotely provided you have the necessary permissions on the remote machine: the command

```
go ga.fl foo neuromancer gref /data/genesis &
```

for instance will recompile "fl.c" on host "neuromancer" in the directory "/data/genesis", which must contain a correctly installed UserMakefile. It will then copy "in.foo" (also "init.foo" and "schema.foo" if applicable) into the remote directory, run the program there (using login name "gref"), 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 GENESIS to be installed on the remote host.

to it as directory argument: "go" exploits this special case to avoid the overhead of copying files between the hosts.

If you have GENESIS experiments queued in files you can execute selected queues by typing "ex <file name(s)>". "ex" notifies you via write(1) or mail(1) when all experiments are completed. "ex" distributes experiments to remote hosts specified in a file "GAhosts" in either the local directory, your home directory or the GENESIS source directory, then runs the remaining experiments (if any) locally. 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 "go" as described above - see the sample GAhosts file in the GENESIS source directory for details.

## The Report

If the 'c' or 'C' option is selected, a report describing the performance of the GA can be produced by the "report" program, which summarizes the mean and variance of several measurements, including online performance, offline performance, the average performance of the current popula-

- 17 -

tion, 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 selected, three additional measures are 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% (ie. 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% 0's or 75% 1's.

## 9. Example

Figure 3 shows an example of a user-defined evaluation function for the following problem:

Min  $f(x) = \sum [(x_i)^2]$ , where  $-5.12 \leq x_i \leq 5.11$ ,  $i=1,2,3$ .

Each  $x_i$  is represented by 10 bits, so that the structure length is 30, and the precision for each  $x_i$  is 0.01. The minimum occurs at the origin. (Of course, this problem does not require the full power of genetic algorithms and can be more appropriately solved using classical optimization techniques.) The following illustrates a typical dialog with the "setup" program, with the user's responses underlined:

```
% setup
```

```
Evaluation File Name [fl]:  
awk -f GEN/wrapper fl.c > fl-ga.c  
cc -c fl-ga.c  
cc -o ga.fl fl-ga.o GEN/ga.a -lm  
awk '/\/\* +GAlength +[0-9]+ +\*\// {print $3}' fl-ga.c  
Object file: ga.fl
```

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

File Suffix [fl]:  
Using 20 defaults from in.fl

Oct 4 15:15 1991 root:/tmp/3607.lwf\_tmp Page 9

Can't open in.fl for writing  
Please provide alternate suffix: expl

Number of Experiments [1]:  
Genome Length [30]:  
Population Size [100]: 50  
Trials per Experiment [3000]: 1000  
Per-Bit Crossover Rate [0.040000]:  
Mutation Rate [0.002000]: 0.0005

Generation Gap [1.000000]:  
Window size (negative: sigma scaling) [-1]: 5  
DPE Time Constant [0]:  
Convergence Threshold [0.800000]:

- 18 -

Max Alleles to Converge [0]:  
Maximum Bias [0.990000]: 0.8  
Max Gens w/o Eval [2]:  
Report Interval [200]:  
Structures Saved [20]: 5  
Dump Interval [2]: 0  
Options [cel]: aceL  
Random Seed: [532125729]:

Writing new settings to in.expl

Application-specific Arguments:  
0 []:

queue []:  
go command executed  
Setup Done

%

The program ga.fl executes. The raw output data is sent to file "out.expl", and the values of the global variables, including the final population, are sent to "ckpt.expl." The report generator produces file "report.expl":

report.expl for ga.fl  
Tue Aug 14 10:38:42 PDT 1990  
Experiments = 1  
Total Trials = 1000  
Population Size = 50  
Structure Length = 30  
Crossover Rate = 0.600000  
Mutation Rate = 0.000500  
Generation Gap = 1.000000  
Scaling Window = 5  
Report Interval = 200  
Structures Saved = 5  
Max Gens w/o Eval = 2  
Dump Interval = 0  
Dumps Saved = 1  
Options = aceL  
Random Seed = 532125729  
Maximum Bias = 0.800000  
Max Convergence = 0

MEAN Gens	Trials	Lost	Conv	Bias	Online	Offline	Best	Average
0	50	0	0	0.568	2.741e+01	7.275e+00	2.017e+00	2.741e+01
3	200	0	0	0.603	2.072e+01	2.768e+00	7.034e-01	1.477e+01
7	400	0	0	0.645	1.583e+01	1.736e+00	7.034e-01	9.079e+00
11	600	0	3	0.708	1.274e+01	1.364e+00	4.916e-01	4.741e+00
15	800	3	3	0.718	1.047e+01	1.068e+00	1.306e-01	3.164e+00
19	1000	3	4	0.727	8.819e+00	8.726e-01	4.020e-02	2.013e+00

- 19 -

The 5 best structures are saved in file "min.expl":

% cat min.expl

01000000	00110000	00101100	110110	1.3060e-01	12	630
01000000	00110000	01101100	110110	1.3130e-01	17	871
01000000	00110000	00011100	011110	4.0200e-02	18	909
01000000	00110001	01101100	110110	2.0260e-01	19	960
01000000	10110000	00101100	110110	1.3210e-01	19	968

Each line of the minfile displays a binary structure, its evaluation, and the generation and trials counters at the time of the first occurrence of this structure.

If it is desired to continue this experiment, edit the input file "in.expl" to increase the total number of trials and add 'x' to the options, then restart the program either directly (by typing "ga.fl expl") or via "go" or "ex".

#### 10. Making Modifications

GENESIS was designed to encourage experiments with genetic algorithms. It is relatively easy for the user to create variations of GENESIS. Suppose for example that you wish to test a new crossover operator. Simply create a file called, say, "mycross.c" which contains a function called "crossover()". This file should "#include extern.h", in order to access global variables (see cross.c). Now, modify the Makefile file so that the loader gets your function instead of the crossover provided, i.e.,

```
cc -o ga.fl fl.o mycross.o GEN/ga.a -lm
```

No recompilation of GENESIS is necessary.

In order to facilitate such experimentation, most of the important variables in GENESIS are global. All global identifiers in GENESIS begin with a capital letter, to minimize conflicts with user-defined global identifiers.

#### Acknowledgments

The author wishes to thank the early users of GENESIS for their suggestions and comments, especially Ray Ford, Jeremy Norton and Mike Fitzpatrick. Further suggestions and

Oct 4 15:15 1991 root:/tmp/3607.lwf\_tmp Page 10

- 20 -

#### References

1. James E. Baker, "Reducing bias and inefficiency in the selection algorithm," in Genetic Algorithms and Their Applications: Proc. 2nd Intl. Conf., ed. J. J. Grefenstette, pp. 14-21, LEA, Cambridge, MA, July 1987.
2. A. D. Bethke, Genetic algorithms as function optimizers, Ph. D. Thesis, Dept. Computer and Communication Sciences, Univ. of Michigan, 1981.
3. A. Brindle, Genetic algorithms for function optimization, Ph. D. Thesis, Computer Science Dept., Univ. of Alberta, 1981.
4. K. A. DeJong, Analysis of the behavior of a class of genetic adaptive systems, Ph. D. Thesis, Dept. Computer and Communication Sciences, Univ. of Michigan, 1975.
5. K. A. DeJong, "Adaptive system design: a genetic approach," IEEE Trans. Syst., Man, and Cyber., vol. SMC-10, no. 9, pp. 566-574, Sept. 1980.
6. S. Forrest, Documentation for Prisoner's Dilemma and Norms Programs that use the genetic algorithm, Technical Report, Univ. of Michigan, 1985.
7. D. R. Frantz, Non-linearities in genetic adaptive search, Ph. D. Thesis, Dept. Computer and Communication Sciences, Univ. of Michigan, 1972.
8. J. H. Holland, Adaptation in Natural and Artificial Systems, Univ. Michigan Press, Ann Arbor, 1975.
9. R. B. Hollstien, Artificial genetic adaptation in computer control systems, Ph. D. Thesis, Dept. Computer and Communication Sciences, Univ. of Michigan, 1971.
10. N. N. Schraudolph and R. K. Belew, Dynamic Parameter Encoding for Genetic Algorithms, Technical Report No. LAUR 90-2795, Los Alamos National Laboratory, 1990.
11. S. F. Smith, "Flexible learning of problem solving heuristics through adaptive search," Proc. 8th Intl. J. Conf. Artif. Intel. (IJCAI), Aug. 1983.

**Appendix B - Source listing of evaluation code *std.c***

The following pages contain the C source listing of evaluation code for the 'iterative accumulation algorithm'.

```

/*
 * What I call the standard algorithm for hashing strings...
 * H hashes the string clc2...cn to an integer.
 *   h[0] = 0
 *   h[i] = h[i-1] * x + c[i] * y + z,   0 < i <= n
 *   H(c[1]c[2]...c[n]) = h[n] modulo tablesize,   0 <= H < tablesize
 *
 * The file of sample items is specified in GENESIS
 * application-specific arguments by "sample=filename".
 *
 * The tablesize is determined by get_tablesize() for each evaluation.
 * If this is used to choose a random tablesize then GENESIS should be
 * given the option -a to ensure reevaluation at each generation.
 * For a fixed table size the function should return the same number each time.
 */

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
double drand48 ();

```

```

/*
 * Estimates of units of time required to execute the operations
 * increment, add, multiply, divide, bitwise and. Assuming and, or, and
 * exclusive or are equivalent. Assuming Addition and subtraction are
 * equivalent. PROBETIME is an estimate of the time required per item to
 * search the linked lists of the separate chaining collision resolution.
 */

```

```

#define INCTIME 4
#define ADDTIME 6
#define MULTTIME 60
#define SHIFTTIME 8
#define DIVTIME 120
#define MODTIME 120
#define ANDTIME 6
#define PROBETIME 450

```

```

/* Maximum char length for each item in the sample file. */
#define MAXITEMLENGTH 256

```

```

/* Range of values to allowed for the size of the hash table. */
/* Used by the function to determine a random table size. */
#define MINTABLESIZE 200
#define MAXTABLESIZE 2000

```

```

/* Application-specific arguments from GENESIS. */
extern char *Gargv[];
extern int Gargc;

```

```

/* Flag to let evalhash know whether it has initialised itself yet. */
int initialised = 0;

```

```

/*
 * The sample items for hashing are read into memory for improved
 * efficiency. They are stored in a linked list (in reverse order to
 * that of file). They are read in when evalhash initialises itself.
 * num_items is obviously the number of items in the list to avoid
 * recounting them many times.
 */

```

```

typedef struct itemnode {
    char *item;
    struct itemnode *next;
} itemnode, *itemnodeptr;
itemnodeptr item_list;
int num_items;

```

```

/*
 * A pseudo hash table. Counts the number items that have been hashed
 * to each of the locations - ie the length of the 'separate chains'.
 * tablesize is size of the hashtable for the current evaluation.
 */

```

```

int hashtable[MAXTABLESIZE];
int tablesize;

```

```

/*
 * The total amount of time used by the hashing function including collision
 * resolution for the current evaluation, based on the estimated unit times.
 */

```

```

unsigned long totalhashtime;

```

```

/*
 * Return the result of applying 'op' to 'arg1' and 'arg2'.
 * 'op' is a numbered operator from the set {- + / * & | ^}, any other
 * numbers return 'arg1'.
 * Division by zero returns 0.
 * An estimate of the time required to excute the operation is added
 * to the global 'totalhashtime'.
 */

```

```

int timedcalc (arg1, op, arg2)
    int arg1, op, arg2;

```

```

{
    switch (op) {
        case 0 :
            totalhashtime += ADDTIME;
            return (arg1 - arg2);
        case 1 :
            totalhashtime += ADDTIME;
            return (arg1 + arg2);
        case 2 :
            totalhashtime += DIVTIME;
            if (arg2 != 0) {
                return (arg1 / arg2);
            }
            else {
                return 0;
            }
        case 3 :
            totalhashtime += MULTTIME;
            return (arg1 * arg2);
        case 4 :
            totalhashtime += ANDTIME;
            return (arg1 & arg2);
        case 5 :
            totalhashtime += ANDTIME;
            return (arg1 | arg2);
        case 6 :
            totalhashtime += ANDTIME;
            return (arg1 ^ arg2);
        default : return arg1;
    }
}

```

```

/*
 * Return the result of applying 'op' to 'arg1' and 'arg2'.
 * 'op' is a numbered operator from the set {- + / * & | ^}, any other
 * numbers return 'arg1'.
 * Division by zero returns 0.
 * An estimate of the time required to excute the operation is added
 * to the global 'totalhashtime'.
 * This is same as 'timedcalc' except that optimisation on 'arg2' is possible.
 * eg:  arg1 * 4 :=> arg1 << 2,   arg1 + 0 :=> arg1

```

```

* Written as a separate function (instead of adding if statement to
* timedcalc) for performance reasons - this is probably most commonly
* called routine - and things are already too slow!
*/

```

```

int opttimedcalc (arg1, op, arg2)
    int arg1, op, arg2;

```

```

{
    switch (op) {
    case 0 :
        if (arg2 != 0)
            if (arg2 == 1)
                totalhashtime += INCTIME;
            else
                totalhashtime += ADDTIME;
        return (arg1 - arg2);
    case 1 :
        if (arg2 != 0)
            if (arg2 == 1)
                totalhashtime += INCTIME;
            else
                totalhashtime += ADDTIME;
        return (arg1 + arg2);
    case 2 :
        if (arg2 != 0) {
            totalhashtime += DIVTIME;
            return (arg1 / arg2);
        }
        else {
            return 0;
        }
    case 3 :
        if (arg2 == 0)
            ;
        else if (arg2 == 1)
            ;
        else if (arg2 == 2)
            totalhashtime += SHIFETIME;
        else if (arg2 == 4)
            totalhashtime += SHIFETIME;
        else if (arg2 == 8)
            totalhashtime += SHIFETIME;
        else if (arg2 == 16)
            totalhashtime += SHIFETIME;
        else if (arg2 == 32)
            totalhashtime += SHIFETIME;
        else if (arg2 == 64)
            totalhashtime += SHIFETIME;
        else if (arg2 == 128)
            totalhashtime += SHIFETIME;
        else
            totalhashtime += MULTTIME;
        return (arg1 * arg2);
    case 4 :
        totalhashtime += ANDTIME;
        return (arg1 & arg2);
    case 5 :
        totalhashtime += ANDTIME;
        return (arg1 | arg2);
    case 6 :
        totalhashtime += ANDTIME;
        return (arg1 ^ arg2);
    default : return arg1;
    }
}

```

```

/*

```

```

* Hash the 'string' using 'coeffs' and 'ops'...
* h[i] = ((h[i-1] o[0] k[0]) o[1] (c[i] o[2] k[1])) o[3] k[2]
* ...where k[] are the coefficients and o[] are the (numbered) operators.
* The hash value returned is guaranteed to be 0 <= H < tablesize.
* Time required for hashing and collision resolution should be added to
* totalhashtime here or in subsidiary functions.
* Updating the hashtable should also happen here.
*/

```

```

int hash (string, coeffs, ops)
    char *string;
    int coeffs[], ops[];

```

```

{
    char c;
    int h=0;
    while ((c=*string++)!='\0') && (c!='\n') {
        h = opttimedcalc(timedcalc(opttimedcalc(h, ops[0], coeffs[0]),
                                ops[1],
                                opttimedcalc(c, ops[2], coeffs[1])),
                        ops[3],
                        coeffs[2]) % tablesize;
    }
    if (h<0)
        h = tablesize + h;
    totalhashtime += PROBETIME * hashtable[h];
    hashtable[h]++;
}

```

```

/*
* Search the application arguments in GARGV for a specification of the
* file of sample items. The specification should be "sample =
* filename" for which '~' as the first character of the filename is
* expanded to the environment variable 'HOME'.
* eg "sample=/tmp/names" -> "/tmp/names",
* "sample = ~/names" -> "/users/cosc/undergrad/honours/mark/names"
* Filenames should be no longer than 256 chars.
*/

```

```

char *getfilename()
{
    char buf[256], tbuf[256], *home;
    int i;
    buf[0] = '\0';
    for (i=0; i<GARGC; i++)
        if (sscanf(GARGV[i], "sample = %s", buf) == 1)
            break;
    if (buf[0] == '\0')
        Error("No sample file in application-specific arguments");
    if (buf[0] == '~') {
        home = getenv("HOME");
        if (home == NULL)
            Error("HOME not in environment\n");
        (void) strcpy(tbuf, buf);
        (void) strcpy(buf, home);
        (void) strcat(buf, tbuf+1);
    }
    return strdup(buf);
}

```

```

/*
* Append the items stored in the file specified by filename
* to list of items in item_list. Each line of the file is treated
* as a separate item. The number of items read is added to
* num_items. The items are added in reverse order at the
* beginning of the list.
*/

```

```

void read_items ()
{
    char *filename;
    FILE *inputfile;
    char buffer[MAXITEMLENGTH];
    itemnodeptr newnode;

    filename = getfilename();
    inputfile = fopen(filename, "r");
    if (inputfile == NULL)
        Error("Couldn't open sample file");

    item_list = NULL;
    num_items = 0;
    while (fgets(buffer, MAXITEMLENGTH, inputfile) != NULL) {
        newnode = (itemnodeptr) malloc(sizeof(itemnode));
        if (newnode == NULL)
            Error("Not enough memory");
        newnode->item = strdup(buffer);
        newnode->next = item_list;
        item_list = newnode;
        num_items++;
    }
    (void) fclose(inputfile);
}

/*
 * Determine a tablesize for the evaluation run.
 * To optimise for a fixed tablesize, simply return the constant fixed size.
 * Similarly tablesizes can be restricted to powers of 2 by rounding to
 * a power of 2 before return.
 * To return a random integer in the range MINTABLESIZE to MAXTABLESIZE
 * inclusive, uncomment the relevant section.
 */
int get_tablesize ()
{
    int t;
    t = (drand48() * (MAXTABLESIZE - MINTABLESIZE + 1)) + MINTABLESIZE;
    return t;
}

/*
 * Evaluation function.
 * The items are read from the sample file if not already done.
 * The hash frequencies 'hashtable' are set to zero.
 * The time used hashing is init to zero.
 * Each item is hashed and the count of it hash value incremented.
 * The performance is the flat (size independent) distribution of the
 * hash frequencies.
 */
double evalhash (coeffs, ops)
    int coeffs[], ops[];
{
    itemnodeptr current_node;
    int i;
    double avgtime;

    if (!initialised) {
        read_items();
        initialised = 1;
    }
}

```

```

tablesize = get_tablesize();
for (i=0; i<tablesize; i++)
    hashtable[i]=0;

totalhashtime = 0;

current_node = item_list;
while (current_node != NULL) {
    hash(current_node->item, coeffs, ops);
    current_node = current_node->next;
}

avgtime = ((double) totalhashtime) / num_items;
return (avgtime);
}

```

```

/*
 * Below is the comment that specifies to the GENESIS wrapper
 * what parameters, their range and format, should be passed to
 * evalhash().
 *
 * a = # bits for          }
 * b = lower limit of range }
 * c = upper limit of range } each parameter
 * d = declare as INT      }
 * e = number of repetitions }
 *
 *          a b   c de operators
 *          V V   V  VV \_____/ */
/* GAeval evalhash 16:0:6553613 3:0:814 */

```

**Appendix C - Output of GENESIS experiment for *std.c***

The following pages contain the report produced by GENESIS for the final experiment using the evaluation code of *std.c*.

```

report.std2 for ga.std2
Mon Oct 7 01:08:16 NZST 1991
  Experiments = 1
  Total Trials = 240000
  Population Size = 2000
  Structure Length = 60
  Crossover Rate = 0.600000
  Mutation Rate = 0.000050
  Generation Gap = 1.000000
  Scaling Window = -1
  Report Interval = 2000
  Structures Saved = 40
  Max Gens w/o Eval = 2
  Dump Interval = 2000
  Dumps Saved = 1
  Options = cellur
  Random Seed = 2401887004
  Maximum Bias = 0.990000
  Max Convergence = 60
  Conv Threshold = 1.000000
  DPE Time Constant = 0
  Sigma Scaling = 2.000000
  
```

-- sample=~/ga/identifiers/1010ids

MEAN

Gens	Trials	Lost	Conv	Bias	Online	Offline	Best	Average
0	2000	0	0	0.505	9.11664e+04	1.25407e+03	7.55000e+02	9.11664e+04
2	4341	0	0	0.525	6.11412e+04	9.72839e+02	7.18000e+02	1.92149e+04
4	6677	0	0	0.532	4.54873e+04	8.83486e+02	6.99000e+02	1.13372e+04
6	9005	0	0	0.537	3.68578e+04	8.35792e+02	6.99000e+02	7.47307e+03
7	10174	0	0	0.539	3.37711e+04	8.20074e+02	6.99000e+02	7.30284e+03
9	12497	0	0	0.542	2.92303e+04	7.97568e+02	6.99000e+02	7.16597e+03
11	14823	0	0	0.543	2.58820e+04	7.82101e+02	6.99000e+02	5.36579e+03
13	17170	0	0	0.546	2.33933e+04	7.70742e+02	6.99000e+02	5.82448e+03
14	18351	0	0	0.547	2.23709e+04	7.66125e+02	6.99000e+02	5.36281e+03
16	20675	0	0	0.548	2.06056e+04	7.58580e+02	6.99000e+02	4.53334e+03
18	23022	0	0	0.549	1.90516e+04	7.52506e+02	6.99000e+02	3.90617e+03
19	24169	0	0	0.550	1.84462e+04	7.49967e+02	6.99000e+02	4.39702e+03
21	26522	0	0	0.552	1.72573e+04	7.45430e+02	6.98000e+02	3.50533e+03
23	28859	0	0	0.553	1.62699e+04	7.41589e+02	6.98000e+02	3.66372e+03
24	30021	0	0	0.554	1.58341e+04	7.39902e+02	6.98000e+02	3.61455e+03
26	32345	0	0	0.556	1.50218e+04	7.36891e+02	6.98000e+02	2.96865e+03
28	34694	0	0	0.557	1.43151e+04	7.34258e+02	6.98000e+02	2.45526e+03
30	37009	0	0	0.557	1.36882e+04	7.31989e+02	6.92000e+02	2.88153e+03
31	38181	0	0	0.558	1.33706e+04	7.30762e+02	6.92000e+02	2.53878e+03
33	40518	0	0	0.559	1.28077e+04	7.28526e+02	6.92000e+02	2.97637e+03
35	42865	0	0	0.559	1.23244e+04	7.26526e+02	6.92000e+02	3.05974e+03
36	44015	0	0	0.560	1.20852e+04	7.25624e+02	6.92000e+02	2.42214e+03
38	46352	0	0	0.561	1.16433e+04	7.23929e+02	6.92000e+02	2.19449e+03
40	48682	0	0	0.562	1.12114e+04	7.22398e+02	6.89000e+02	2.21654e+03
42	50978	0	0	0.564	1.08312e+04	7.20894e+02	6.89000e+02	2.08764e+03
43	52136	0	0	0.563	1.06440e+04	7.20185e+02	6.89000e+02	1.94811e+03
45	54446	0	0	0.564	1.03279e+04	7.18862e+02	6.89000e+02	2.55877e+03
47	56788	0	0	0.565	1.00088e+04	7.17631e+02	6.89000e+02	1.84645e+03
49	59119	0	0	0.566	9.72895e+03	7.16502e+02	6.89000e+02	2.16194e+03
50	60299	0	0	0.567	9.57750e+03	7.15964e+02	6.89000e+02	1.69687e+03
52	62621	0	0	0.568	9.32719e+03	7.14964e+02	6.89000e+02	2.14024e+03
54	64949	0	0	0.568	9.07331e+03	7.14033e+02	6.89000e+02	1.62515e+03
55	66113	0	0	0.569	8.94702e+03	7.13592e+02	6.89000e+02	1.59908e+03
57	68439	0	0	0.569	8.71960e+03	7.12757e+02	6.89000e+02	1.69043e+03
59	70751	0	0	0.570	8.50114e+03	7.11980e+02	6.89000e+02	1.75424e+03
61	73084	0	0	0.571	8.28720e+03	7.11247e+02	6.89000e+02	1.57528e+03
62	74229	0	0	0.571	8.18534e+03	7.10904e+02	6.89000e+02	1.44246e+03
64	76537	0	0	0.572	7.99689e+03	7.10243e+02	6.89000e+02	1.66964e+03
66	78856	0	0	0.572	7.82370e+03	7.09618e+02	6.89000e+02	1.56325e+03
67	80021	0	0	0.573	7.73286e+03	7.09318e+02	6.89000e+02	1.38561e+03

69	82323	0	0	0.574	7.57244e+03	7.08750e+02	6.89000e+02	1.65718e+03
71	84633	0	0	0.575	7.40760e+03	7.08211e+02	6.89000e+02	1.44085e+03
73	86950	0	0	0.575	7.25388e+03	7.07699e+02	6.89000e+02	1.58986e+03
74	88101	0	0	0.576	7.18035e+03	7.07455e+02	6.89000e+02	1.37672e+03
76	90416	0	0	0.577	7.02850e+03	7.06982e+02	6.89000e+02	1.10389e+03
78	92724	0	0	0.578	6.88920e+03	7.06535e+02	6.89000e+02	1.20357e+03
80	95023	0	0	0.579	6.75109e+03	7.06110e+02	6.89000e+02	1.06643e+03
81	96165	0	0	0.580	6.68586e+03	7.05907e+02	6.89000e+02	1.13632e+03
83	98475	0	0	0.580	6.55908e+03	7.05511e+02	6.89000e+02	1.16996e+03
85	100785	0	0	0.581	6.43741e+03	7.05132e+02	6.89000e+02	1.12420e+03
87	103110	0	0	0.581	6.32499e+03	7.04768e+02	6.89000e+02	1.23663e+03
88	104253	0	0	0.581	6.26942e+03	7.04596e+02	6.89000e+02	1.14474e+03
90	106577	0	0	0.581	6.16091e+03	7.04255e+02	6.89000e+02	1.16939e+03
92	108894	0	0	0.582	6.05169e+03	7.03931e+02	6.89000e+02	9.92093e+02
93	110038	0	0	0.583	5.99899e+03	7.03776e+02	6.89000e+02	9.66688e+02
95	112313	0	0	0.585	5.89815e+03	7.03476e+02	6.89000e+02	9.61244e+02
97	114619	0	0	0.585	5.80201e+03	7.03185e+02	6.89000e+02	9.67419e+02
99	116922	0	0	0.586	5.71051e+03	7.02906e+02	6.89000e+02	1.05085e+03
100	118096	0	0	0.586	5.66360e+03	7.02767e+02	6.89000e+02	9.70470e+02
102	120404	0	0	0.587	5.57375e+03	7.02504e+02	6.89000e+02	9.35041e+02
104	122730	0	0	0.589	5.48692e+03	7.02248e+02	6.89000e+02	9.73442e+02
106	125043	2	2	0.589	5.40280e+03	7.02003e+02	6.89000e+02	9.14187e+02
107	126203	2	2	0.590	5.36345e+03	7.01883e+02	6.89000e+02	1.01831e+03
109	128509	2	2	0.591	5.28441e+03	7.01652e+02	6.89000e+02	9.38982e+02
111	130823	3	3	0.591	5.20953e+03	7.01428e+02	6.89000e+02	9.29042e+02
113	133133	2	2	0.592	5.13553e+03	7.01212e+02	6.89000e+02	9.23071e+02
114	134286	3	3	0.592	5.09948e+03	7.01108e+02	6.89000e+02	9.12365e+02
116	136581	2	2	0.593	5.03104e+03	7.00904e+02	6.89000e+02	9.12742e+02
118	138867	3	3	0.593	4.96660e+03	7.00708e+02	6.89000e+02	1.00090e+03
119	140023	3	3	0.593	4.93330e+03	7.00611e+02	6.89000e+02	9.13869e+02
121	142316	3	3	0.593	4.86907e+03	7.00424e+02	6.89000e+02	9.03288e+02
123	144620	3	3	0.595	4.80673e+03	7.00242e+02	6.89000e+02	9.18702e+02
125	146909	4	4	0.597	4.74629e+03	7.00067e+02	6.89000e+02	9.03712e+02
126	148055	4	4	0.597	4.71869e+03	6.99982e+02	6.89000e+02	1.04500e+03
128	150370	4	4	0.598	4.66010e+03	6.99812e+02	6.89000e+02	8.79708e+02
130	152652	4	4	0.600	4.60461e+03	6.99651e+02	6.89000e+02	9.28019e+02
132	154922	3	3	0.600	4.55165e+03	6.99495e+02	6.89000e+02	8.78840e+02
133	156070	4	4	0.601	4.52474e+03	6.99418e+02	6.89000e+02	8.64081e+02
135	158378	4	4	0.603	4.47215e+03	6.99266e+02	6.89000e+02	8.55333e+02
137	160678	2	2	0.603	4.42288e+03	6.99119e+02	6.89000e+02	9.08091e+02
139	162978	3	3	0.604	4.37277e+03	6.98976e+02	6.89000e+02	8.47621e+02
140	164122	4	4	0.606	4.34840e+03	6.98906e+02	6.89000e+02	8.49465e+02
142	166413	3	3	0.606	4.30249e+03	6.98770e+02	6.89000e+02	9.88955e+02
144	168716	1	1	0.607	4.25740e+03	6.98637e+02	6.89000e+02	9.72409e+02
146	171018	4	4	0.607	4.21244e+03	6.98507e+02	6.89000e+02	8.81553e+02
147	172136	4	4	0.607	4.19199e+03	6.98445e+02	6.89000e+02	9.65212e+02
149	174444	4	4	0.608	4.14857e+03	6.98320e+02	6.89000e+02	8.82335e+02
151	176715	3	3	0.609	4.10919e+03	6.98201e+02	6.89000e+02	9.79428e+02
153	179009	4	4	0.609	4.06970e+03	6.98083e+02	6.89000e+02	1.01025e+03
154	180147	4	4	0.609	4.04946e+03	6.98025e+02	6.89000e+02	8.58869e+02
156	182418	4	4	0.610	4.01163e+03	6.97913e+02	6.89000e+02	8.83564e+02
158	184681	4	4	0.611	3.97366e+03	6.97804e+02	6.89000e+02	8.73582e+02
160	186958	4	4	0.612	3.93712e+03	6.97696e+02	6.89000e+02	8.52758e+02
161	188105	4	4	0.612	3.91848e+03	6.97643e+02	6.89000e+02	8.64712e+02
163	190369	3	3	0.613	3.88351e+03	6.97541e+02	6.89000e+02	8.98791e+02
165	192634	4	4	0.614	3.84847e+03	6.97440e+02	6.89000e+02	8.65078e+02
167	194922	4	4	0.614	3.81472e+03	6.97341e+02	6.89000e+02	9.69102e+02
168	196069	4	4	0.614	3.79745e+03	6.97292e+02	6.89000e+02	8.52773e+02
170	198355	4	4	0.615	3.76400e+03	6.97197e+02	6.89000e+02	8.45010e+02
172	200627	2	2	0.617	3.73156e+03	6.97104e+02	6.89000e+02	8.87611e+02
174	202893	3	3	0.617	3.69951e+03	6.97013e+02	6.89000e+02	8.42862e+02
175	204036	3	3	0.618	3.68474e+03	6.96962e+02	6.87000e+02	9.55562e+02
177	206328	2	2	0.618	3.65393e+03	6.96852e+02	6.87000e+02	8.15000e+02
179	208594	3	3	0.619	3.62371e+03	6.96745e+02	6.87000e+02	8.50639e+02
181	210861	4	4	0.619	3.59519e+03	6.96640e+02	6.87000e+02	8.57697e+02
183	213138	4	4	0.618	3.56723e+03	6.96537e+02	6.87000e+02	8.43848e+02
184	214267	4	4	0.619	3.55294e+03	6.96487e+02	6.87000e+02	8.33345e+02

186	216529	4	4	0.619	3.52608e+03	6.96387e+02	6.87000e+02	8.45636e+02
188	218770	3	3	0.619	3.49987e+03	6.96291e+02	6.87000e+02	9.52039e+02
190	221009	4	4	0.619	3.47540e+03	6.96197e+02	6.87000e+02	9.82490e+02
191	222146	3	3	0.619	3.46223e+03	6.96150e+02	6.87000e+02	8.75169e+02
193	224360	4	4	0.619	3.43632e+03	6.96060e+02	6.87000e+02	8.18408e+02
195	226633	4	4	0.621	3.41049e+03	6.95969e+02	6.87000e+02	8.28915e+02
197	228914	4	4	0.621	3.38600e+03	6.95880e+02	6.87000e+02	9.48314e+02
198	230062	3	3	0.621	3.37328e+03	6.95835e+02	6.87000e+02	8.24140e+02
200	232281	3	3	0.622	3.35039e+03	6.95751e+02	6.87000e+02	9.58611e+02
202	234569	4	4	0.622	3.32594e+03	6.95665e+02	6.87000e+02	8.15922e+02
204	236840	2	2	0.622	3.30219e+03	6.95582e+02	6.87000e+02	8.30273e+02
206	239098	3	3	0.622	3.27905e+03	6.95501e+02	6.87000e+02	8.26000e+02
207	240236	3	3	0.622	3.26763e+03	6.95461e+02	6.87000e+02	8.38590e+02

**Appendix D - Best structures from GENESIS  
experiment for *std.c***

The following pages contain the best structures produced by GENESIS for the final experiment using the evaluation code of *std.c*. The columns from left to right are performance, coefficients x, y, z, and the four operators a, b, d, l. These form the hashing function H:

```

h0 = 0;
hi = [(hi-1 a x) b (ci d y)] | z   for 1 ≤ i ≤
      n;
H(c1c2c3...cn) = hn mod tablesize;
    
```

687	26096	17088	59805	0	6	7	7
687	26096	24886	6557	0	6	7	7
687	26096	24889	6557	0	6	7	7
687	26096	30280	35323	0	6	7	7
687	26096	30704	25803	0	6	7	7
687	26096	30705	52347	0	6	7	7
687	26096	30711	36411	0	6	7	7
687	26096	30711	5636	0	6	7	7
687	26096	31532	61026	0	6	7	7
687	26096	31532	63652	0	6	7	7
687	26096	32021	6557	0	6	7	7
687	26096	32057	6242	0	6	7	7
687	26096	32057	6557	0	6	7	7
687	26096	32058	42594	0	6	7	7
687	26096	32063	59805	0	6	7	7
687	26096	32075	39453	0	6	7	7
687	26096	32454	6557	0	6	7	7
689	11531	27129	30453	0	6	7	7
689	11531	3290	12877	0	6	7	7
689	11531	47950	16045	0	6	7	7
689	11531	54768	27114	0	6	7	7
689	11531	55573	42317	0	6	7	7
689	11531	55618	39589	0	6	7	7
689	11531	56395	51594	0	6	7	7
689	11531	56969	3349	0	6	7	7
689	11531	57318	27114	0	6	7	7
689	11531	59874	51621	0	6	7	7
689	11531	59893	61591	0	6	7	7
689	11531	59898	18210	0	6	7	7
689	11531	59898	29426	0	6	7	7
689	11531	59898	8690	0	6	7	7
689	11531	63310	6605	0	6	7	7
689	11531	63756	12877	0	6	7	7
689	11531	63794	37082	0	6	7	7
689	11531	63796	10458	0	6	7	7
689	11531	63820	20140	0	6	7	7
689	11531	63820	20141	0	6	7	7
689	11531	63822	4388	0	6	7	7
689	11531	64857	50354	0	6	7	7
689	11531	64857	6605	0	6	7	7