1-1-1999

Genetic algorithms using Galib

Bradley John Hendricks
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
https://digitalscholarship.unlv.edu/rtds/1078

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6” x 9” black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeib Road, Ann Arbor, MI 48106-1346 USA

UMI®
800-521-0600
GENETIC ALGORITHMS USING GALIB

by

Bradley Hendricks

Bachelor of Science
Department of Computer Science
University of Nevada, Las Vegas
1997

Bachelor of Science
Department of Physics
University of Nevada, Las Vegas
1997

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree
Department of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
December 1999

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The Thesis prepared by
Bradley Hendricks

Entitled
Genetic Algorithms Using GAlib

is approved in partial fulfillment of the requirements for the degree of
Master of Science, Computer Science

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative
ABSTRACT

Genetic Algorithms using GAlib

by

Bradley Hendricks

Dr. Bein, Examination Committee Chair
Professor of Computer Science
University of Nevada, Las Vegas

GAlib is a C++ library of genetic algorithm objects that was recently developed at the Massachusetts Institute of Technology. This thesis is to demonstrate its functionality and versatility for implementing haploid tripartite genetic algorithms.

We first built a test bed in which GAlib could be used. To achieve this, we used GAlib to solve the Traveling Salesman Problem and implemented two-opt and simulated annealing for comparison. We then examined the use of genetic algorithms for finding loop invariants. We used GAlib successfully to build a model but results remain inconclusive.

In our main thrust we applied genetic algorithms to train and develop neural networks. To develop neural network architectures we used two different methods of representing neural networks: connection matrices and graph-generation grammars. We were able to demonstrate that genetic algorithms are an effective tool for training networks as well as for finding network architectures.
## TABLE OF CONTENTS

ABSTRACT ................................................................. iii

LIST OF FIGURES ................................................... vi

ACKNOWLEDGMENTS .............................................. viii

CHAPTER 1 THE USE OF GALIB .............................. 1
  General Overview .................................................. 2
  Overview of the Genetic Algorithm Object .............. 4
  Overview of the Population Object ....................... 4
  Overview of The Genome Object ......................... 4
  Implementation .................................................... 6
  Writing the Objective Function ............................. 7
  Genetic Algorithm Objects ................................... 9
  Genome Objects .................................................. 10
  Additional Objects .............................................. 13
  Customizations to GAlib .................................... 13
  Customizing the Initialization Method .................. 14
  Customizing the Mutation Method ....................... 15
  Customizing the Crossover Method ....................... 15
  Creating a Custom Genome Class ....................... 17
  Creating a Custom GA Class ............................... 21
  Cellular Automaton Example ............................... 21

CHAPTER 2 GENETIC OPTIMIZATION OF THE TRAVELING
SALESMAN PROBLEM .............................................. 25
  Two-Opt ............................................................ 26
  Simulated Annealing .......................................... 28
  Genetic Algorithm .............................................. 29
  Cycle Crossover .................................................. 30
  Ordered Crossover and One-Point Crossover .......... 30
  Edge Recombination Crossover ............................ 31
  TSP View .......................................................... 32
  Results ............................................................. 38

CHAPTER 3 AUTOMATED SEARCH FOR LOOP INVARIANTS
USING GENETIC ALGORITHMS ............................... 46
LIST OF FIGURES

1  Structure of a Simple Program Using GAlib ............................................ 7
2  Setting parameters for a GA ......................................................................... 7
3  Example of an Objective Function ............................................................... 8
4  Example of an Initializer Function ............................................................... 14
5  Example of a Mutator Function .................................................................. 16
6  Example of a Crossover Function ............................................................... 18
7  Example of a Custom Genome Class Header ............................................ 19
8  Example of a Custom Genome Class Implementation ................................... 20
9  Example of a Custom GA Class .................................................................. 21
10 Near Optimal Tour of a 52 City TSP in 2-D Euclidean Space ............... 26
11 Two-opt Neighbor ............................................................................................. 27
12 The Simulated Annealing Algorithm for Optimizing the TSP ............... 29
13 Example of Ordered Crossover .................................................................. 31
14 The Edge Recombination Algorithm ......................................................... 32
15 Example of an Edge Recombination Crossover ........................................ 33
16 User Interface of TSP View ......................................................................... 35
17 The GA Settings Dialog ................................................................................ 36
18 The Mutation and Crossover Settings Dialog ............................................ 37
19 Circle of 100 Points and the Optimal Tour ............................................... 39
20 Circle of 100 Points and an Intermediate Tour ........................................ 40
21 152 City TSP: Tour Found by Two-Opt ..................................................... 43
22 152 City TSP: Tour Found by Simulated Annealing .................................. 43
23 152 City TSP: Tour Found by the Genetic Algorithm after 2000 Generations With a Population of 1000 .............................................................. 44
24 152 City TSP: Tour Found by the Genetic Algorithm after 7500 Generations With a Population of 2000 .............................................................. 44
25 Simple Program That Multiplies a and b in a Loop .................................... 48
26 Input and Output Assertions, the Loop Invariant, and the Weakest Precondition for the Multiplication Loop ......................................................... 48
27 Proof of First Requirement: \( I \rightarrow P \) ......................................................... 48
28 Proof of Second Requirement: \( P \land C \rightarrow wp(P,S) \) ................................ 49
29 Proof of Third Requirement: \( P \land \neg C \rightarrow O \) ................................................ 49
30 Loop Prover User Interface ............................................................................ 55
31 Initialization Settings Dialog ........................................................................ 56
32 Mutation Settings Dialog ............................................................................... 56
33 Crossover Settings Dialog ............................................................................. 57
34 Objective Settings Dialog ............................................................................. 58
35 Example of a Mutation of a Weighted Neural Network ............................ 62
ACKNOWLEDGMENTS

I would like to thank Dr. Bein for his guidance in the research and preparation for this thesis and for his assistance with the grant funding.

I also would like to thank Dr. Minor for his assistance and suggestions for our work on the loop invariant genetic algorithm.

Thanks to Steve Lumos for developing the \texttt{\LaTeX} document class used to write this paper.
CHAPTER 1

THE USE OF GALIB

GAlib is a C++ library developed by Matthew Wall (see [21]) at the Massachusetts Institute of Technology designed to assist in the development of genetic algorithm applications. The library contains many classes that offer functionality and flexibility in the design of optimization applications with genetic algorithms. The library includes predefined genetic algorithm models, genome types, and genetic operators for the quick creation of simple applications, and the ability to customize the library's functionality for more complicated optimizations. This library was developed so that it may be used with a variety of compilers on many platforms. The library has been used successfully on DOS/Windows, Windows NT/95, MacOS, and UNIX systems. GAlib was designed to work with Microsoft Visual C++, Borland C++ and GNU compilers, as well as others. Our programs were written in the Microsoft Visual C++ environment.

GAlib supports several different models of genetic algorithms. The simple genetic algorithm is the textbook genetic algorithm, where after each generation, the population of possible solutions is completely replaced by the mutation and crossover of the previous generation. The incremental and steady state genetic algorithms both replace only a portion of the population with each generation. Deme GA evolves multiple population and migrates individuals from one population to another. This algorithm model can run on parallel processors, evolving each of the populations on a separate processor. It is also possible to develop a custom genetic algorithm to suit the purposes of an application.
Each of these GA types is simple to implement and gives a great deal of freedom in their operation. A variety of algorithm termination methods, selection methods, random number generators, and statistics are available to choose from. Most of these features may be customized as well. Also, crossover and mutation probabilities, population overlap, and population size are customizable.

Any data type in C++ may be used to create a genome type. GAlib includes several of the most common genome types. These include one-dimensional, two-dimensional and three-dimensional arrays of binary elements, and 1-D arrays of real or character-valued elements. In addition to these, are lists, trees, 1-D, 2-D, and 3-D arrays, all of which are templates and allow the programmer to select any valid C++ data type. All the arrays may be set to any desired length, and the trees and lists have dynamic sizes. Each of these genome types has built-in initialization, crossover, mutation and comparison methods, which can be customized by the programmer. The only routine that must be coded by the programmer is the objective function, which is a function that evaluates an individual from the population with a fitness score.

The versatility and ease of GAlib makes it a useful tool for implementing genetic algorithms. It is versatile enough to apply to complex optimization problems through customization, yet still simplifies the work. For simple genetic algorithm applications, little programming is required. Also, because it includes a large variety of genetic algorithm and genome types, and it is written with a hierarchical structure, it is simple to modify software already using GAlib to perform new tasks.

General Overview

When programming using GAlib, one works primarily with two classes: a genome class and a genetic algorithm class. A genome instance represents a single individual in the population of solutions. The genetic algorithm defines how the solution will
be evolved. In addition to defining these two classes, an objective function is needed. If the classes supplied by GAlib are inadequate to the task at hand, they may be customized, or the programmer may develop his or her own implementations.

The three necessary steps to develop an application using GAlib are:

- define a representation
- define the genetic operators: initialize, mutate, and crossover
- define the objective function

GAlib includes many examples, built-in operators, and genome representations to aid in the first two steps, but the objective function must be implemented by the programmer. Once these three steps have been completed, the genetic algorithm can begin its search for a solution.

A single genome instance is created to represent a possible solution to an optimization problem. The genetic algorithm will create a population of individuals copied from this structure. Then, the genetic algorithm will operate on the population in an attempt to evolve the best solution. The data genome structure used by GAlib is called a GAGenome. The GAlib library contains genomes represented as an array, a list, a tree, and a string of binary bits. These genome types are derived from the class GAGenome and a data structure class. For example, the class GATreeGenome class, which represents a tree structure, inherits from the class GAGenome and the class GATree. The programmer may choose from one of these built-in genomes or if none of GAlib’s available genome types will work as a representation of a solution to the problem at hand, the programmer may develop his or her own genome type. This programming style is the most cumbersome aspect of GAlib and described in detail later.

In addition to the genome types available, GAlib offers a selection of genetic algorithm models to choose from. The basic types of genomes included are the sim-
pie, steady-state, and incremental genetic algorithms. These algorithm types inherit from the class `GAGeneticAlgorithm`. They differ from each other in ways that new population members are created and replace the old population members.

A properly implemented genetic algorithm will be capable of performing local searches as well as global searches for the best solution to an optimization problem. A feature of GAlib is that it is simple to modify the parameters of the genetic algorithm in order to find the best conditions for the search.

**Overview of the Genetic Algorithm Object**

The genetic algorithm object controls the process of evolution. It determines which individuals to mate, which to replace, and which survive. It also keeps track of statistics and determines when to stop the evolution. The genetic algorithm follows a series of steps. First, the population is initialized. Next, for every generation until the termination requirements have been met, individuals are selected for mating. The crossover is performed, the offspring are mutated, and then inserted into the population. The programmer selects the requirements for termination. He or she can choose to terminate after a specified number of generations, once a certain fitness score has been achieved, or by a measurement of the population convergence. The programmer may also write a customized termination function.

**Overview of the Population Object**

The population object contains all the genomes making up the population. It keeps track of statistics about the population as well. It keeps the best solution, the average fitness, the deviation and other metrics. The population object also maintains the selection method used to select the individuals to mate.
Overview of The Genome Object

The genome object has three primary operators used in the evolution of solutions. The initialization operator inserts genetic material into the genome to initialize the evolution. The mutation operator changes a portion of the genetic material in one individual to generate a new one. The crossover operator takes two genomes and combines them to form a new genome. GAlib has defaults for each of these operators, but the programmer can customize them to apply to the problem at hand.

The initialization operator is called at the beginning of the genetic algorithm. It initializes the genome with new genetic material. Instead of creating new genome objects, it inserts the genetic material into the genome data structure. From this genetic material, the GA will evolve the solutions of the optimization problem.

The mutation operator defines how a genome is mutated to produce a new individual. The operator should be able to mutate to obtain new genomes local to the current solution as well as those which are distant from the original. It should be able to introduce new genetic material into the genome and modify existing material. Mutation operators act on different data types differently. For example, mutating an array structure should change a specified value in the array. Mutating a tree should change the structure of the tree as well as the data stored in the tree. It may be necessary to define several different forms of mutation for a single application.

The crossover operator takes two parent genomes and combines them to form a child genome. The crossover, like the mutation operator, should be specific to the data type in the genome. The crossover may also be dependent on the specific problem as well. For example, the traveling salesman problem requires that the genome maintain a permutation of all cities in the genome. The crossover used in the algorithm must sustain this property in the new children generated.

In addition to these three operators, the programmer must create the objective function, which is called to calculate a fitness score for each member of the population.
This function is the only portion of the genetic algorithm that must be programmed. A comparator may also be included, but is not required. This function measures the difference between two members of the population and is used for some statistical measures.

For most applications, the supplied genome types are more than adequate, but it is often times necessary to write one's own genome type. In general, it is not necessary for the library to know the meaning of the contents of the genome. GAlib is written with tremendous generality, so any genome type, custom or otherwise, can be used with any genetic algorithm type.

The genetic algorithm takes care of when to clone the population, perform crossovers, mutations, initializations, etc. All of these operations are performed via the genome member functions.

Implementation

Figure 1 shows an example of a program written with GAlib. To implement a genetic algorithm, first a genome must be declared. This is done on line 3. Notice that the objective function is passed as a parameter to the constructor. The function is passed to the genome so that it can be called when the genome needs to be evaluated. Once a genome has been created, declare an instance of a genetic algorithm object, passing the genome to its constructor. The genome declared here is not used in the algorithm itself, but instead a population of genomes is cloned from it. The GASimpleGA function evolve, line 5, can then be called to initiate and run the algorithm.

In this example, the genome selected is a one-dimensional string of binary values. The length of the string is determined by the value of the variable length. The genetic algorithm object used in this application is a GASimpleGA. This is an algorithm that completely replaces the population each generation with a new one created by the crossover and mutation operators. One may also wish to set various parameters
```c
/*1*/ void main()
/*2*/ {
    // Declare a single genome object, which will be duplicated.
    /*3*/ GA1DBinaryStringGenome genome(length, Objective);
    // This one is a simple GA with a population of 1-D binary
    // strings.
    /*4*/ GASimpleGA ga(genome); // Declare the genetic algorithm.
    // Evolve a solution by calling the evolve member function.
    /*5*/ ga.evolve();
    // Print the results after evolution has completed by calling
    // the statistics function.
    /*6*/ cout << ga.statistics() << endl;
    /*7*/ }
/*8*/ float Objective(GAGenome &)
/*9*/ { // Write the code for the appropriate objective function here.
/*10*/}
```

Figure 1: Structure of a Simple Program Using GAlib

that change the operation of the genetic algorithm, as in Figure 2. These member
functions of GAGeneticAlgorithm types set the size of the population, the number
of generations to evolve and the probabilities for mutation and crossover respectively.
The minimize member function switches the optimization from the default maximiza-
tion to a minimization.

```c
ga.populationSize(popsize);
ga.nGenerations(ngen);
gapMutation(pmut);
gacrossover(pcross);
```

Figure 2: Setting parameters for a GA

Writing the Objective Function

The objective function is the only place where the programmer codes in the meaning
of data stored in the genome. The objective function returns a floating-point value,
which is the objective score for the genome. The function is passed the genome to be evaluated as an instance of the generic class GAGenome. GAlib requires this so that the function header matches with the library. All genomes must inherit from the class GAGenome and the genome must inherit from a data type class as well in order to implement the data type as a genome. This is true for all genome types included with the library. Because the genome passed to the objective function is of the generic GAGenome class, it must first be type cast into the previously defined genome before the objective score can be calculated. Figure 3 is an example of a simple objective function. The function gives fitness scores equal to the number of ones in the genome's array.

```c
/*!*/
float Objective(GAGenome & g)
/*2*/ {
    //Type-casting the GAGenome to a GAlDBinaryStringGenome.
    GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &) g;
    /*5*/
    float score = 0.0;
    /*6*/
    //This for-loop sums all the ones in the genome in score
    for(int i =0; i<genome.length(); i++);
    /*7*/
    score += genome.gene(i); //The member function gene returns the value in the string at location i.
    //Return the sum of ones as the final objective score.
    /*8*/
    return score;
    /*9*/
}
```

Figure 3: Example of an Objective Function

The type cast shown in Figure 3 (line 3) creates a new variable genome of type GA1DBinaryStringGenome, which now is the correct type for it to be evaluated. After this type casting, the data and specific member functions of the genome can be accessed to calculate its score. The gene function of the GA1DBinaryStringGenome called on line 7 returns the value at the given location in the string. The length function returns the length of the string.

The objective function can be defined as a static member of a custom genome
class, as described in detail later, or it can be written independently of the genome and passed to the genome constructor, which is the most common and easiest method. It can also be set with the evaluator member function of the genome class, if it should change during the evolution.

Genetic Algorithm Objects

GAlib is packaged with a selection of genetic algorithm types. The available algorithm types are the GASimpleGA, the GASteadyStateGA, the GAINcrementalGA, and the GADemeGA. Each of these types and any customized algorithm inherit from the class GAGeneticAlgorithm.

The GAGeneticAlgorithm class is an abstract class and can therefore have no instantiations. This class keeps track of statistics (number of crossovers and mutations, best, mean, and worst in each generation, etc.). It also defines the terminator function, which stops the evolution, and parameters, such as crossover and mutation probabilities.

The functions pCrossover and pMutation can be called to set and get the probabilities for crossover and mutation. The function population is called to set and get the population and the function nGenerations can be called to set and get the number of generations to evolve before completing the evolution. The done function returns true if termination requirements for the GA have been met and false if they have not. The function generation returns the current generation the genetic algorithm is evolving.

To control the evolution of the algorithm, the programmer can invoke the functions evolve, initialize, and step. evolve first initializes, then evolves the population generation by generation until the termination requirements have been met. initialize resets the evolution and initializes each individual. step completes a single generation of the evolution.
The GASimpleGA is the simple algorithm as described by Goldberg (see [7]). Every generation the current population is completely replaced by the children generated by the crossover and mutation operators. The elitism flag can be set for this algorithm type with the `elitist` function. This causes the population to always keep the single best individual every generation.

GASTeadyStateGA uses an overlapping population model. The GA creates a population of individuals with the crossover and mutation operators. It then merges this new population with the previous population and removes the worst individuals to return to the original number of individuals. Setting the `pReplacement` parameter determines the percentage of the population to be replaced each generation. The `nReplacement` parameter specifies the exact number of individuals to be replaced. Only one of these parameters can be set at a time. Setting one overrides the other.

The GAINcrementalGA also uses an overlapping population model, but the overlap is very small; one or two new individuals are added to the old population each generation. These new members replace the individuals with the worst score by default, but they can be set to replace individuals based on custom requirements. The number of children generated each generation can be set by the `nOffspring` function to one or two, the default being two. Because this is such a slow evolutionary process, this model was never used in our work.

GADemeGA evolves populations in parallel and migrates individuals between them. Each of the separate populations evolves with a steady-state GA as described above, but each generation, some individuals migrate between populations. The function `nMigration` determines the number of the best population members of each population to migrate. The `nReplacement` or `pReplacement` functions are used to specify the population to be replaced in each generation of the steady-state genetic algorithms as described above.
Genome Objects

Most problems to be optimized by a genetic algorithm can be contained in the genome types provided in the library. These include one, two and three-dimensional binary strings and template arrays, a template list, a template tree, a character array, and a real array. Each of these genomes is implemented as a class within GAlib, inheriting from the abstract class GAGenome.

The genetic operators for mutation, crossover, initialization, comparison, and scoring are all passed one or more objects of type GAGenome. These GAGenome instances must then be type cast into the correct specific genome class so that the data stored in the genome can be accessed. The genomes are passed as the general GAGenome because it facilitates the customization of GAlib's classes.

The functions mutator, crossover, initializer, comparator, and evaluator all specify the function used to perform their named operations during the genetic evolution. The functions mutate, initialize, compare, and evaluate call the functions set by the above. The function sexual returns a pointer to the crossover function, but only the genetic algorithm object is responsible for calling this function. The clone function allocates memory for a new instance of the genome and the copy function replicates the contents of a genome into another genome. Because the genetic algorithm object performs the genetic operations and creates the population, it is usually unnecessary to call these functions directly. They are, however, useful while testing the implementation of newly designed operators and genomes.

The function score returns the fitness score of a genome, and the insertion operator (<<) is defined to output the contents of the genome. With these two class members, the final result of an evolution can be output by displaying the best genome score and contents.

1DBinaryStringGenome, 2DBinaryStringGenome, and 3DBinaryStringGenome contain arrays of binary elements. A single element can be read or modified with the
gene function. The set and unset members are used to modify a range of elements in an array.

\texttt{GA1DArrayGenome<T>, GA2DArrayGenome<T>, and GA3DArrayGenome<T>} are all arrays of the supplied template type. Any class or type may be used as the template class as long as the comparison operators \(==\) and \(!=\) are defined, as well as the assignment operator \(=\) and a copy constructor. This makes these array types very versatile. Many common optimization problems can make use of these genome types. The gene function allows access to the elements of the genome, as it does for the binary genomes, and the swap function is also defined for exchanging two elements in the array.

The array and binary string classes all have user defined dynamic lengths. The functions length, width, and depth set the size of the array in the first, second, and third dimensions respectively.

Also included with GA\text{Lib} is the \texttt{GAListGenome<T>}. This genome type incorporates the flexibility of the template as well. The list is circular and doubly linked. The list can be traversed and modified using the current, head, next, prev, and tail functions. The warp function allows access to a specified location in the list. destroy and remove both remove nodes from the list. However, remove returns a pointer to the item and does not free the memory used by the item. destroy completely removes the item from memory. The insert function can be called to insert an item into the list and the swap function exchanges two items in the list.

The \texttt{GATreeGenome<T>} can represent and manipulate a tree with nodes of any valid type or class. The children of a single node are kept as a circular linked list with the eldest child at the head of the list. All children have a pointer to the parent and the parent node has a pointer to the eldest child. A tree has only one root. A variety of operators have been supplied for traversal of the tree and insertion and deletion of nodes.
All of the above classes are packaged with default crossovers and mutators. Some also have default initializers and comparators. For one-dimensional arrays, the programmer can choose from one-point and two-point crossovers. The one point crossover is also available to the list genome. In multidimensional arrays, the genome matrix is divided into quadrants. These quadrants are merged with quadrants from another parent to create a new child. The tree genome crossover swaps a subtree between two parents.

Mutators for the arrays and list swap two random elements. The tree genome swaps two subtrees within one genome to form a new genome. The initializer is defined for the binary strings. The initializer uniformly selects ones and zeros for each element. A comparator is also defined, and this counts the number of items that differ between the two genomes being compared.

Of course, if any of these genomes or operators do not fit the problem at hand, it is possible to establish operators and genomes of one’s own design. For most problems, however, the supplied genomes are adequate. In many cases, though, it is desirable to write one’s own genetic operators.

Additional Objects

Along with genetic algorithm classes and genome classes, GAlib includes other classes used in the optimization process. A GAStatistics object keeps track of various statistics throughout evolution; the GAPopulation object contains the populations evolved and the GAScalingScheme and GASelectionScheme control how the GA scores are scaled and how genomes are selected to mate respectively. There is also a set of random number generators included with GAlib.

Customizations to GAlib

To use GAlib to its greatest capacity, it is necessary to understand how GAlib can be
customized. Each of the genetic operators and the genome itself can be defined by the programmer. Even a genetic algorithm object can be customized. The following sections explains further how the customization process works.

Customizing the Initialization Method

For most of the genome types, and always when a new genome is created, the programmer must supply a custom initialization function. An initializer function is passed an object of GAGenome class, which must be type cast into the appropriate genome type. The function must be void and therefore returns nothing. Initializers are associated with a genome by using the initializer member function of the genome object. Figure 4 is an initializer function that assigns random floating-point values between min_weight and max_weight to each member of the genome. The genome inherits from class CArray, a dynamic array class packaged with the Microsoft Foundation Class Library under Microsoft Visual C++.

```c++
/*1*/ void CArrayGenome::Init(GAGenome & g)
/*2*/ {
    //Type-casting the GAGenome to a CArrayGenome.
/*3*/ CArrayGenome & genome = (CArrayGenome&) g;
/*4*/ int i;
/*5*/ genome.SetSize(struct_size); //Sets the dynamic CArray's size.
    //Initialize each element of the array to a random value.
/*6*/ for(i=0; i<struct_size; i++)
/*7*/    genome[i] = GARandomFloat(min_weight, max_weight);
/*8*/ }
```

Figure 4: Example of an Initializer Function

The initializer was used to assign random weights to a neural network in the Brain Evolver program. The initialization function only assigns values. The genetic
algorithm object has already allocated the memory for this genome before the initializer is called.

Customizing the Mutation Method

Often it is necessary to implement a custom mutator. Mutations may be dependent on the problem, and it may be desirable to have more than one type of mutation occurring during the evolution. The mutator function is passed two parameters: the genome as a GAGenome, which again, must be type cast, and the mutation probability as a floating point. It is up to the mutator how this probability is interpreted. The mutator should return an integer value as the count of the number of mutations that have occurred. Use the genome member function mutator when assigning the function as the mutation method.

The mutator in Figure 5 was also used in the program Brain Evolver. It mutates the weights of the incoming edges connected to a random node in the network by adding a random float between the min_weight and max_weight values (lines 9-11). The genome, as before, inherits from the CArray class.

Customizing the Crossover Method

As with the mutator, occasionally programmers will need to define their own crossover methods. The crossover function receives four parameters: the two parents and the two children. The parents are passed as GAGenome objects, and the children as GAGenome pointers. All of these must by type cast. The crossover should be defined so that either one or two children can be generated. The function should return an integer, the number of children created, always one or two. If one of the GAGenome pointers is nil, the crossover should not try to generate a child at that pointer. The
/*1*/ int CArrayGenome::Mutate(GAGenome& g, float pmut)  
/*2*/ {  
    //Type-cast GAGenome into CArrayGenome.  
/*3*/    CArrayGenome & genome = (CArrayGenome &) g;  
    //Use a random number to test if the genome should be mutated.  
    //Mutation should only occur with probability pmut.  
/*4*/    if(pmut<=0 || GARandomFloat() >= pmut)  
/*5*/    {  
/*6*/      return 0;  
/*7*/    }  
/*8*/    int node, i;  
/*9*/    //Pick a random node from the neural network to mutate  
/*10*/    node = GARandomInt(0, max_nodes - 1);  
/*11*/    //Mutate all the incoming connections to the random node  
/*12*/    //by adding a random number to their weights.  
/*13*/    for(i=connection_start[node]; i<=connection_finish[node]; i++)  
/*14*/      genome[i] = genome[i] + GARandomFloat(min_weight, max_weight);  
/*15*/    return 1;  
/*16*/}  

Figure 5: Example of a Mutator Function
children have already been allocated, so the crossover function does not need to create
memory for the new children.

The crossover in Figure 6 takes genetic material from one parent and inserts it
into the other to create a new child. This is a one-point crossover, where a single
location in the parent strings is selected at random. Genetic material from parent
one (mom) and parent two (dad) are inserted into the two children. Child one (bro)
gets the material in mom to the left of the crossover location and the material in dad
to the right of that location. Likewise, child two (sis) gets material to the left of the
crossover point in dad and to the right of the location in mom.

Notice that the function tests to see if the variables c1 and c2 are nil before
attempting to crossover and create a new child (lines 11 and 20). Also notice that
the variable nc keeps track of how many children are created and its value is returned
(lines 4, 13, 22 and 29).

Creating a Custom Genome Class

A programmer may derive his or her own genome class from a pre-defined data object.
In Figure 7 and Figure 8, an example showing the definition of the CArrayGenome
class, the class inherits from CArray, the data object, and GAGenome. All custom
genomes must inherit from GAGenome.

The constructors and copy function should be written as in Figure 7 and Figure 8.
The programmer must insert the correct name of the custom genome and copy the
data by the appropriate method for the genome's data type in the copy function.
The clone function and assignment operator (=) also must be defined. The genetic
algorithm object uses the clone function to allocate memory for new genomes in
the population. It is passed a CloneMethod variable, an enumerated type. The
CloneMethod parameter could be used to inform the clone method whether it should
copy the contents of the genome into the newly constructed genome, or if a only a
//1*/ int GAGenome::Cross(const GAGenome & p1, const GAGenome & p2,
//2*/ GAGenome * c1, GAGenome * c2)
//3*/ {
//4*/ int nc = 0; //Number of new children counter.
//5*/ int i, j, cross;
//6*/ //Type-casting GAGenomes into GAlDArrayGenomes for the parents.
//7*/ CArrayGenome & mom = (CArrayGenome &) p1;
//8*/ CArrayGenome & dad = (CArrayGenome &) p2;
//9*/ //Type-casting GAGenome * into GAlDArrayGenome * for children.
//10*/ CArrayGenome * bro = (CArrayGenome *) c1;
//11*/ CArrayGenome * sis = (CArrayGenome *) c2;

//Select the location to crossover with the two parents.
//12*/ cross = GARandomInt(1,mom.GetSize()-2);

//Check if bro is nil before he gets the genetic material.
//13*/ if(bro)
//14*/ {
//15*/ nc++; //Increment number of new children.
//16*/ bro->copy(mom); //Copy mom’s genetic material into bro.
//17*/ //Insert the selected genetic material from dad into bro.
//18*/ for(i=cross;i<mom.GetSize();i++)
//19*/ {
//20*/ (*bro)[i] = dad[i];
//21*/ }
//22*/ }

//Check if sis is nil before she gets the genetic material.
//23*/ if(sis)
//24*/ {
//25*/ nc++; //Increment number of new children.
//26*/ sis->copy(dad); //Copy dad’s genetic material into sis.
//27*/ //Insert the selected genetic material from mom into sis.
//28*/ for(i=cross;i<mom.GetSize();i++)
//29*/ {
//30*/ (*sis)[i] = mom[i];
//31*/ }
//32*/ }
//33*/ //Return the number of children generated.
//34*/ return nc;
//35*/ }

Figure 6: Example of a Crossover Function
class CArrayGenome :
  public CArray<int, int>,
  public GAGenome
{
  public:
  GADefineIdentity("CArrayGenome", 201);
  //Declaration of genome operators and evaluator
  static void Init(GAGenome&);
  static int Mutate(GAGenome& , float);
  static float Compare(const GAGenome& , const GAGenome&);
  static float Objective(GAGenome&);
  static int Cross(const GAGenome& , const GAGenome& ,
                   GAGenome* , GAGenome*);

  //Constructor that assigns the initializer, mutator,
  //crossover, comparator, and objective functions.
  CArrayGenome();

  //The copy constructor.
  CArrayGenome(const CArrayGenome& orig) { copy(orig); }

  //The destructor member function.
  virtual ~CArrayGenome() {}

  //Definition of assignment operator =.
  CArrayGenome& operator=(const GAGenome& orig);

  //The clone function which allocates memory for a new genome
  virtual GAGenome* clone(CloneMethod) const;

  //The copy function duplicates the contents of a genome.
  virtual void copy(const GAGenome& orig);

  //Declare any other member functions and variables here.
};

Figure 7: Example of a Custom Genome Class Header
Figure 8: Example of a Custom Genome Class Implementation
new empty genome is needed. It is not necessary to incorporate this functionality in the implementation by always copying the genome contents, as is done in the example. The code for the initializer, crossover, mutator, and objective functions must be implemented as described in the previous examples. It is not necessary to define the genetic operators and objective functions as members of the class, but it is common practice to do so. A comparator may be implemented as well, but GAlib does not require this. The `GADefineIdentity` function (Figure 7, line 6) takes the genome name and a number greater than 200. This is used to identify the genome in error messages.

Creating a Custom GA Class

It is also possible to create one’s own genetic algorithm class, although it is not likely for this to be necessary. MyGA in Figure 9 inherits from `GASteadyGA`, but the functions in that class can be overridden to do whatever the programmer desires.

```cpp
/* 1 */ class MyGA : public GASteadyStateGA
/* 2 */ {
/* 3 */ public:
/* 4 */   GADefineIdentity(“MyGA”, 280);
/* 5 */   MyGA(const GAGenome& g) : GASteadyStateGA(g) { }
/* 6 */   virtual ~MyGA() {}
        \Override desired functions in GASteadyStateGA here.
/* 7 */   }
```

Figure 9: Example of a Custom GA Class

Cellular Automaton Example

In Appendix A is an example of a complete program using GAlib. The genetic algorithm is used to evolve a one-dimensional cellular automaton (CA) that can determine
if an initial configuration string is filled with 50% or more ones than zeros. A cellular automaton is defined as a set of rules, which, when applied to a binary array, modify the current array into a new one. The CA can be applied to the array successively for a specified number of steps, each time applying the rules over the entire array. The most common CA is known as the game of life. That CA is applied to a two-dimensional array, but in this case, the array is a one-dimensional binary string. Following the previous work done by Mitchell, Crutchfield, Hraber, Das, and Hanson (see [14], [15], and [6]) this GA should find a cellular automaton which solves the majority problem. The CA found by the genetic algorithm should be able to change a string to a complete sequence of ones if the initial string contains more than half ones. It should also create a complete string of zeros if there are less than half ones in the initial configuration.

The rules in a CA are applied over windows in the binary string. From the center of the window, the rule can examine all the bits less than or equal to a set radius in each direction. In the case of the example, the CA can examine two bits in each direction, for a total of five bits in a window. Depending on the state of these five bits, the CA will replace the center bit of the window with a one or zero in the new string. The CA applies the rules in windows at every position in the old string simultaneously, generating a new string. At the ends of the string, the window wraps to the other end, so that every window contains the same number of bits.

Because there are five bits, there are a total of $2^5$ or 32 possible rules. There are rules for the window containing bits 00000, 00001, 00010, up to 11111. Each of these rules returns a 0 or a 1 to replace the current middle value of the window. These ones and zeros are stored in the genome, which has a length of 32-one position for each rule. The binary equivalent of the array position where a gene is located specifies when the rule applies to a window. For example, if position 10 in the genome array is zero, whenever a window contains the bits 00101, the new string will contain a zero at the middle location.
The program generates a set of 100 random initial configurations to score the genome with. To ensure that solutions are evolved which work on all initial cases, new initial configurations are generated each generation and the whole population is re-evaluated. The initial configurations are chosen at varying levels of difficulties; the easiest strings contain almost all ones and zeros to more difficult strings, which contain nearly uniform ones and zeros. An interesting addition to this GA is that every generation the initial configurations used to score the population slowly become more difficult at a specified rate, with more initial configurations closer to uniform.

The program demonstrates how to use GAlib to implement and solve a problem. The genetic algorithm object and the genome object are both defined in the main function. The main function steps through the GA, outputting results and generating new sets of ICs each generation. Notice how the GAGenome member functions initialize and step are used. Before these two functions are called, however, the parameters have been set.

The function GetIC was written to generate sets of initial configurations that increase in difficulty each generation. Within the set of configurations are strings that range in difficulty as well. RunIC takes one CA rule set and one initial configuration and applies the rules to the configuration. If the CA works correctly and solves the majority problem for that IC, the function returns one. Otherwise, zero returns. The objective function, Objective, uses RunIC to add up the number of times a CA works correctly for all the strings in the IC. These three functions act together as the user-programmed part of the GAlib application. They determine the objective score for the genomes in the population.

As seen from this example, GAlib does most of the dirty work in genetic algorithm programming for the programmer. It is not necessary to worry about keeping track of populations, statistics, and generation of new population members. In most cases, the data type, crossover, and mutation are taken care of as well. GAlib saves tremendous
amounts of time in the programming of simple genetic algorithms, and for more complicated algorithms, it frees the programmer to concentrate on the more important aspects of the program.
CHAPTER 2

GENETIC OPTIMIZATION OF THE TRAVELING
SALESMAN PROBLEM

As a test-bed for GAlib, the first optimization application we attempted was the Traveling Salesman Problem (TSP). The Traveling Salesman Problem has no known polynomial-time solution. It is an NP-complete problem, and because of this, heuristics and approximation methods are required to achieve useful solutions to the problem.

With the TSP, a set of n cities is given, along with the distances between each of them. This represents a graph. The graph may be directed or undirected. The solution to the TSP is, simply enough, the tour which visits every city in the set once and has the shortest possible total distance (Figure 10). The simplicity of this problem is, however, deceiving. There exists (n — 1)!/2 unique undirected tours for a TSP with n ≥ 3 cities. To always find the optimal solution, there is no better method than to examine each of the individual tours to determine which tour has the minimum distance. This is, of course, unacceptable because to solve a TSP with a reasonable number of cities would take extraordinary amounts of time. A simple TSP with 100 cities could take centuries to find an optimal solution.

The TSP can have cities in any metric space, with any distance assigned between two cities, but in this study, we limited ourselves to cities in a two-dimensional Euclidean space. Each city has an x and y coordinate and the distance is calculated from the Euclidean distance formula.

In addition to genetic algorithms, we analyzed two other optimization methods
Figure 10: Near Optimal Tour of a 52 City TSP in 2-D Euclidean Space

in solving Traveling Salesman Problems: two-opt and simulated annealing. Two-opt employs a simple method to find a solution, but it will always find a local minimum. It has no way of exiting a local minimum once it has been reached. Simulated annealing can jump out of a local minimum by accepting solutions with longer tour lengths based on probability. The genetic algorithm can also escape local minimums because it keeps a population of several solutions, some of which may contain lower scores than the population's best and because genetic mutations can move a solution out of a local minimum. However, it was known before we attempted genetic optimization that genetic algorithms worked poorly for the TSP. The best results came from, as expected, simulated annealing. The genetic algorithm often could not outscore the two-opt algorithm.
The two-opt TSP optimization method employs a simple search method. Two-opt begins with a random permutation of cities—a random tour. A two-opt neighbor with a shorter, better, tour length will replace this tour. Neighbors are generated until none further reduce the tour length. A two-opt neighbor is generated by selecting two edges from the tour, edge \((a, b)\) and edge \((c, d)\) where \(a, b, c, \) and \(d\) are cities (Figure 11). These edges are then replaced by edges \((a, c)\) and \((b, d)\). If the new tour containing these edges results in a shorter tour, the old best tour is replaced by this newest neighbor. Because there are \(n\) cities, there are \(n(n - 1)/2\) neighbors.

The two-opt algorithm as implemented generates neighbors repeatedly. It takes the first edge in the tour and tests neighbors using all the other edges in the tour. It then generates neighbors with the second edge versus the other edges in the tour and repeats the process until all \(n(n - 1)/2\) neighbors have been generated. It starts the process over again and continues cycling through neighbors until it completes a cycle without finding a two-opt neighbor that can replace the existing best tour. The neighbors analyzed by two-opt are selected in a deterministic order.

Figure 11: Two-opt Neighbor

Two-Opt
Two-opt too easily becomes trapped in local minimums, whose length is far from optimal, yet it is the fastest of the optimization methods tested. Because of its speed, two-opt can be used to assist both the simulated annealing and genetic algorithm methods.

Simulated Annealing

Simulated annealing, as proposed by Kirkpatrick et al. (see [9]) and Černý (see [3]), uses an optimization method that will occasionally accept nearest neighbor tours with longer, worse lengths than their current best tour. Simulated annealing algorithms are based on the processes of annealing metal and glass. In this process, the molten substance is slowly cooled to temper it and make it less brittle. In the simulated annealing algorithm, a cooling temperature is used. When the temperature is high, it is more likely for a tour with a longer distance to be accepted as a possible solution. At cool temperatures, only solutions that decrease in distance are accepted.

A nearest neighbor in the simulated annealing process is defined in the same way as the two-opt nearest neighbor. Also similar to two-opt, simulated annealing begins with a random tour. Unlike two-opt, simulated annealing randomly selects the two edges used to form the nearest neighbor. If the nearest neighbor tour has a shorter distance than the current solution, it is always accepted as the new current best. If the solution is worse, there is still a probability for it to be accepted. This probability is based on the current temperature, $c$. With probability $e^{-\Delta/c}$ the longer nearest neighbor is accepted, where $\Delta$ is equal to the difference in the length of the nearest neighbor and the length of the current solution. The algorithm always keeps track of the best solution encountered throughout the optimization.

The temperature is decreased in small increments as the algorithm works. When "equilibrium" is reached, the temperature is reduced. We declared equilibrium to be reached for $n$ cities after $n(n - 1)$ nearest neighbors had been tested, as suggested by
the text by Aarts and Lenstra (see [1]). In our software, the user may enter the initial
temperature and the amount the temperature is reduced once equilibrium has been
established. The complete simulated annealing algorithm is shown in Figure 12.

1) Generate a random starting solution $S$ and set the initial champion solution $S^\ast = S$.

2) Set initial temperature $c$.

3) Choose a random two-opt neighbor $S'$ of the current solution $S$.

4) Let $\Delta = \text{Length}(S') - \text{Length}(S)$.

5) If $\Delta \leq 0$, set $S = S'$. Else with probability $e^{-\Delta/c}$, set $S = S'$.

6) If $\text{Length}(S) < \text{Length}(S^\ast)$, set $S^\ast = S$.

7) If equilibrium reached, $n(n - 1)$ iterations completed, reduce $c$, else goto 3.

8) If $c > 0$ goto 3.

9) return $S^\ast$ as final solution

Figure 12: The Simulated Annealing Algorithm for Optimizing the TSP

Genetic Algorithm

It is not completely obvious how to apply a genetic algorithm to the TSP. It is not
immediately clear how to store a TSP tour as a genome and how crossovers and
mutations should occur. We know from the results of previous work by Brady (see
[2]) and Mühlenbein et al. (see [17]) that a genetic algorithm is not the best tool for
solving the TSP. For our studies, we chose to represent a tour as a permutation of the
cities in the order they are visited in the tour. The last leg of the tour is from the last
city in the permutation to the first. It is essential that all members of the populations
remain complete permutations of cities. The crossover and mutator functions cannot
add or subtract cities from the tour. Only the order in which cities are visited can
be modified. The initializer simply generates a random permutation of cities.
The mutator we developed has several components and can mutate the tour in a variety of ways. It can swap any two cities in the tour, replace a tour with a random two opt-neighbor, and insert portions of the tour into a random location in the array. These forms of mutation all have their own probability of occurring.

Forming a working and meaningful crossover posed a more difficult problem. We implemented three different crossovers with varying degrees of success. The cycle crossover is a simplistic crossover, but the child generated from this crossover often times has no similarity to the parents. The ordered crossover is a more meaningful crossover. It retains similarity between the child and its parents. The edge recombination crossover is an $O(n^2)$ algorithm for performing crossover and does not perform as well as the ordered crossover. All of these crossovers are explained in detail below.

**Cycle Crossover**

The cycle crossover, as described by Oliver, Smith, and Holland (see [18]), is a simplistic crossover that merges the genetic material from two parents and generates two new children. The new child will always contain a valid tour. The cycle crossover finds a subset of cities in the tour. The positions in the genome that this subset occupies must be the same for both parents. For example, for a TSP with five cities, two parent tours could be 12534 and 31425. The subset \{1,2,3\} occupy the first, second, and fourth locations in both tours. Crossing these positions in both parents generates the new children. The children resulting from this example are 12435 and 31524. Because the same subset of cities is replaced in both parents, the children are always complete permutations.

**Ordered Crossover and One-Point Crossover**

The ordered crossover, as described by Prins (see [19]), is the most effective crossover
we implemented for the TSP. In the cycle crossover, because non-adjacent cities in the parent tours were used in the crossover, the edges in the parent’s tour rarely survived the crossover, and the resulting child consisted of a large percentage of random edges. The ordered crossover preserves more of the parental edges and the general shapes of the parental tours as well.

This is accomplished by taking a random slice of one of the parent’s tour directly. This random slice is taken between two cuts in the parent’s genetic material. The remainder of the permutation is taken from the second parent’s tour. From the second parent, cities are inserted into the child in the order they occur in that parent, starting after the second cut location. Only cities not already in the child’s tour are inserted. See Figure 13 for an example. The vertical line in the genomes is the second cut location.

The one-point crossover from Prins ([19]) is similar to the ordered crossover. The difference in this crossover is that the random slice taken from the first parent always begins at the left of the array used to store the tour. This crossover is more limiting than the ordered crossover, and therefore was not used.

Edge Recombination Crossover

The edge recombination crossover, as implemented by Matthew Wall in an example supplied with GAlib, inserts parental edges into the children whenever possible. When no edge can be inserted, a random edge is selected. This crossover method uses an $O(n^2)$ algorithm and causes the algorithm to run much more slowly than the previously mentioned crossovers. This crossover showed no improvement over either of the above, and was not extensively tested.
The crossover begins by combining the two parents into a single graph $G$ as the union of the two tours. The first city $c_1$ in the tour is selected at random. The next city to be inserted is taken from the union graph. The city attached to $c_1$ by an edge in $G$ with the smallest degree is taken as the next city in the child, $c_2$. If there is more than one city with the smallest degree attached to $c_1$, randomly choose among them. If no city is attached to $c_1$, choose at random among all cities not yet inserted into the child. Repeat this process for $c_3$ and all other cities until every city has been inserted into the child. See Figure 14 for the complete algorithm and Figure 15 for an example. In the example, two possible parent tours are shown for an eight city TSP. The first step is to create the union of the two tours as shown. Then randomly select a city, in this case b. From that city, continue selecting the next adjacent city with the smallest degree.

1. From the two parents, $P_1(V, E_1)$ and $P_2(V, E_2)$, form the graph $G(V, E)$ as the union of $P_1$ and $P_2$.
2. Randomly select a city $c$ from $V$ as the first city in the crossover. Add $c$ to $C$, the new child.
3. In the graph $G$, find the subset $V'$ of all cities attached by an edge to $c$.
4. If $V'$ is empty, select a city $c'$ at random from all cities not in $C$. Else, randomly select a city $c'$ among the cities in $V'$ with the smallest degree.
5. Append $c'$ to the partial tour $C$.
6. Remove all edges in $G$ connected to $c$.
7. Let $c = c'$.
8. Goto 3 until all cities in $V$ have been added to $C$.

Figure 14: The Edge Recombination Algorithm

TSP View

We developed the program TSP View to approximate the Traveling Salesman Problem.
Figure 15: Example of an Edge Recombination Crossover
using GAlib under Microsoft Visual C++. With the software developed, we could test the performance of genetic algorithms versus two-opt and simulated annealing. Also, the software allows the users to modify many parameters in the genetic and simulated annealing algorithms. Figure 16 is a picture of the user interface.

The large display area to the right shows a map of the cities and the tour current connecting them. It also shows the generation for a GA and the current temperature for simulated annealing, along with the current length of the tour. The text area at the lower left displays the permutation representing the tour. The show numbers button displays the city numbers in the map display. The draw map and show path buttons allow the user to disable the map and tour path displays respectively. This may be used to improve the speed of the approximation.

The program was designed to accept files in a TSP format. These files contain a city number and the x and y coordinate for each city in the set. The filename can be entered by hand in the edit area, or by pressing the button, one can browse the directory structure for the desired file.

The edit areas allow the user to set the GA population and number of generations to evolve. Also, the user may set the mutation and crossover probabilities. When the user selects the simulated annealing button, he or she may enter the initial temperature as well as the amount to reduce the temperature each generation.

In addition to these, there are three buttons which change the way optimization occurs. The "Opt. Init. Soln." button causes the software to run the two-opt algorithm on each member of the GA population after initialization, so that the algorithm starts with solutions already in local minimums. It also optimizes the first solution of the simulated annealing algorithm. The "Opt. Each Step" button optimizes each new solution generated by the genetic algorithm, and the "Opt. for Score Only" button optimizes each solution for scoring in the GA, but doesn't change the contents of the GA.
Figure 16: User Interface of TSP View

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The interface includes buttons that allow the selection of the three approximation algorithms and edit windows and dialog boxes for the modification of parameters. The GA button brings up a dialog box to select the type of GA to use (Figure 17). Another dialog box allows the user to specify the mutation probabilities and the crossover to employ in the optimization (Figure 18).

The GA dialog box allows the user to select from the simple genetic algorithm, the steady state genetic algorithm, and the incremental. These are described in detail above, as implemented in GAlib. The steady state GA proved far better than the other two, and therefore was used exclusively in our studies. The user may alter the percentage of the population replaced in each generation by modifying the “% replacement” value. The mutation and crossover dialog allows the user to modify the probabilities of the various forms of crossover occurring, and it allows the user to select which of the four crossovers to apply.

The rotation and inversion mutations do not change the contents of tour; they only change the way they are stored in the genome. Rotation “rotates” the array by shifting the contents a random position to the right. Elements pushed off the array on the right are moved to the front of the array. Inversion flips the contents of the array, so that the last element is now first in the array. The swap mutator swaps two elements in the array. If swap does not occur, the remaining mutation will occur.
Figure 18: The Mutation and Crossover Settings Dialog
This crossover selects a random slice of the tour and inserts it elsewhere in the tour. When inserted, there is a 50% chance that the slice's contents will be inverted.

We chose the CArray as the data type to use in the genome developed for TSP View. This is a dynamic array template class included with Microsoft Visual C++ in the Microsoft Foundation Class Library. The CArray is easy to implement in the Microsoft Visual C++ environment and included operators and functions that aided in programming. The functionality of the GA1DArrayGenome class was adequate to this task as well, but using CArray allowed us to implement a custom genome for an actual problem.

Within the CArrayGenome class is defined the initializer, four crossovers, the four mutators, and the objective function. The objective function merely adds the cost of each leg of the tour and returns that sum as the objective score. The initializer sets the size of each array to the number of cities in the problem and generates a random tour for each member of the population.

Results

To test the three algorithms for solving the TSP, we used a selection of TSP data files compiled by Reinelt (see [20]) with different numbers of cities and different geometries. In addition, we defined a few simple test cases with known optimal tours to determine the basic evaluation capabilities of each search. The cities in these problems formed a circle of points with different numbers of cities (see Figure 19). The optimal tour simply forms the circumference of the circle, visiting each city. All the TSP data used contain cities in a two-dimensional Euclidean plane.

All three algorithms successfully find the optimal tours in the circle of points for up to 400 cities in the tour. It is believed that they would be successful at completing the tour with up to 1000 cities, but time constraints prevented me from performing these tests with anything except the two-opt algorithm, which did find the optimal
Figure 19: Circle of 100 Points and the Optimal Tour

tour. Of course, this is to be completely expected. There are no tours in the circle that are local minimums.

For any non-optimal tour, there exists a two-opt neighbor that reduces the total tour length. Because all the points lie on the edge of a circle and the optimal tour connects the circle, any complete tour that is not optimal must contain two edges that intersect (see Figure 20). If this were otherwise, the tour would have to be separated into two or more sub-tours. By its definition, a two-opt neighbor will uncross any pair of intersecting edges, producing two new edges with a smaller total weight than the original pair. Since there exists a two-opt neighbor with smaller weight that will uncross any pair of edges, there exists a two-opt neighbor for any non-optimal tour. By this and the definition of a local minimum with two-opt moves, the optimal tour is the only minimum for the set of points on the circumference of a circle.

Because of this property, the circle of points is not a rigorous TSP for the testing of approximation methods, but it does allow for performing benchmarks and to find
the best parameters to apply to the algorithms. Two-opt naturally was the speediest of the algorithms, and the only method capable of conquering the 1000 point circle in a reasonable amount of time. Within five minutes the two-opt algorithm returned an optimal tour. It returns an optimal tour for 400 cities in less than one minute. Because it is a deterministic algorithm and there are no local minimums, it would be impossible for it to return a non-optimal solution.

Simulated annealing is also rapid, but takes considerably longer to solve the 1000 point TSP. Because the algorithm is randomized, it must wait until it can find a random two-opt that uncrosses two edges. Once most of the edges that cross the center of the circle have been optimized out, it becomes more difficult and time consuming to find two-opt neighbors that decrease the tour length. To establish an optimal tour for a 400-point circle takes about ten minutes.

The genetic algorithm is a much slower approximation method and its speed is heavily dependent upon the population size and the crossover method. For every
case of the circle of cities, we used the steady-state model for the genetic algorithm, replacing the entire population each generation, and a maximum of 2000 generations, although for the smaller number of cities, optimal was usually reached before this limit. Most testing was done for the 100-city circle. We selected a population of 500 individuals for these tests.

The edge recombination crossover is by far the slowest of the crossovers implemented. Each generation of evolution for the 100-city problem takes approximately ten seconds. To complete 2000 generations would take about five and a half hours. This crossover method shows no improvement over any of the others and its consumption of time makes it an undesirable choice as a crossover method.

The cycle crossover is the fastest crossover, but is only minutely faster than the ordered crossover. It can establish an optimal tour for the 100-city problem in approximately 1100 generations and in about seven minutes, considerably longer for the 400-city circle.

The ordered crossover is slightly slower than the cycle, but it can complete the 100 city circle in about eight minutes and 1100 generations. It is slightly slower in time, but equal in number of generations. For more complicated problems, this crossover generally outperforms the cycle crossover because of its ability to retain most of the parental tours in the children of a crossover.

Through experimentation, we have found that it is best to keep the probability of inversion and rotation very low. When too much of these mutations take place, the crossover operators cannot function as well because similar parent tours are offset to each other and do not match in the crossing of genetic material. The swap and insertion crossovers are set to occur with equal likelihood. Their probability of occurring is not critical to the algorithm.

The circle TSP is useful for the measuring and the honing of parameters, but is in no way a true test of an approximation’s abilities to optimize a TSP. For the next
round of testing, we used a TSP in the TSPLIB95 (see [20]) designed by Grötschel of fifty-two locations in the city of Berlin. Figure 10 shows the optimal tour for this set of cities. This tour was found by the genetic algorithm, but has been found from all three algorithms. All of the algorithms have reached an optimal tour for this set, but it is much more likely for simulated annealing to find the best tour. The genetic algorithm also finds the best tour and almost always will find a tour within five percent of optimal. Two-opt rarely comes up with optimal, and often times find solutions more than ten percent of the optimal length.

Many other sets of cities, some with complicated geometries, were tested as well, with simulated annealing always outperforming the other two. In the TSP of Figure 21, Figure 22, and Figure 23 it is easy to see how this makes the problem of finding the optimal tour more difficult. Obviously, the 152-city tour created by Padberg and Rinaldi (see [20]) will contain a minimal number of edges between the vertical columns of cities, but also it must find the best edges between the columns. These three figures illustrate the relative performances of the three algorithms. The genetic algorithm stopped after 2000 generations, but could have decreased its length with larger population, more generations and time. Figure 24 shows the performance of the GA after 7500 generations with a population of 2000. The shortest known tour length for this TSP is 73682. This run took about three hours to complete. It is easy to see from these examples that the genetic algorithm with additional time can be more effective than two-opt because it can often break free of local minimums where two-opt would be frozen. Yet, because of its slow speed and poorer performance, it is easy to see that genetic algorithms are not well suited for solving the TSP. Simulated annealing is a much faster and overall better approximation for this type of problem.

From the study of the TSP, though, we have shown that a genetic algorithm can be used to approximate simple TSP sets. We have also shown the versatility of GAlib for solving non-trivial problems, as was shown as well in our work where we applied
Figure 21: 152 City TSP: Tour Found by Two-Opt

Figure 22: 152 City TSP: Tour Found by Simulated Annealing
Figure 23: 152 City TSP: Tour Found by the Genetic Algorithm after 2000 Generations With a Population of 1000

Figure 24: 152 City TSP: Tour Found by the Genetic Algorithm after 7500 Generations With a Population of 2000
genetic algorithms in the search for loop invariants, even though our work here did not bear fruit. The full success of genetic algorithms is shown in its application to neural nets, where our research has shown that genetic algorithms can be used to train networks and evolve network structures.
CHAPTER 3

AUTOMATED SEARCH FOR LOOP INVARIANTS USING GENETIC ALGORITHMS

Currently, there is no automated way of proving a computer program is correct. There are algorithmic methods to show that a computer program fragment containing no loops is correct, but there exists no method to prove a program with a loop and given input restrictions produces the correct output. This is because to prove such a program containing a loop is correct, one must first find the loop invariant. This is a first-order logic expression that describes the changes that occur in the loop and is true for every iteration of the loop. To find the loop invariant directly, one would have to solve a second-order logic expression. There is currently no automated way of solving second or higher order logic expressions. This therefore forces us to apply heuristics and search methods to find a loop invariant.

To prove a program correct, one must be supplied with the program source, the input restrictions called the input assertion, and the output restrictions called the output assertion. These assertions are first-order logic expressions. Because we are only concerned with finding the loop invariants, we will assume that the input and output assertions are taken directly before and after the loop in the program code respectively. This means that the program we analyze will consist only of a loop. The input assertion will describe the state of any of the program variables needed in the loop before its execution. The output assertion will describe the state of any variables immediately after the loop exits. The code will be the body of the loop and we will also need the loop condition as a first-order logic expression.
The code in the loop, in our research, is simplified to assignment statements, with a program variable on the left and an expression on the right. For full functionality, it is necessary to include if-statements as well. The variables in the program hold integer values or are integer arrays. With this simple functionality, we were able to test simple programs that multiply with iterative addition and calculate the factorial of an input value. Also implemented was a loop that zeros an array, sums the values in an array, and searches an array for a specific value.

A loop invariant must satisfy these three logic conditions:

1. The assertion before the loop must imply the loop invariant.

2. The loop invariant and the loop condition must imply the weakest precondition of the loop invariant after execution of the loop body.

3. The loop invariant and the negated loop condition must imply the output assertion.

The weakest precondition is found by applying the known rules for assignment and if-statements to the loop invariant. Starting with the last statement in the loop, modify the loop invariant using the known rules. The resulting formula is now true just before that statement of the program is executed. Repeat this process for every line in the loop, until the statement is true just before the loop execution.

The simple program shown in Figure 25 adds the number $b$ to $a$ times, effectively multiplying $a$ and $b$. The input assertion, output assertion, loop invariant being proven, and the weakest precondition of the loop are listed in Figure 26. The first-order logic proofs of the three logic conditions above are shown in Figure 27, Figure 28, and Figure 29. Resolution and paramodulation are the inference rules applied to solve the proofs.

If and only if all of these proofs hold true for a given first-order logic expression, is that expression a valid loop invariant. The first proof shows that the loop invariant is
\begin{verbatim}
  i := 0;
c := 0;
while (i < a)
  c := c + b;
i := i + 1;
\end{verbatim}

Figure 25: Simple Program That Multiplies a and b in a Loop

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a \geq 0$</td>
</tr>
<tr>
<td>2</td>
<td>$i = 0$</td>
</tr>
<tr>
<td>3</td>
<td>$c = 0$</td>
</tr>
<tr>
<td>4</td>
<td>$c \neq i \land a \geq i$</td>
</tr>
<tr>
<td>5</td>
<td>$0 \times x = 0$</td>
</tr>
<tr>
<td>6</td>
<td>$c \neq 0 \land a \geq i$</td>
</tr>
<tr>
<td>7</td>
<td>$c \neq 0 \land a \geq i$</td>
</tr>
<tr>
<td>8</td>
<td>$a \geq i$</td>
</tr>
<tr>
<td>9</td>
<td>$a \geq 0$</td>
</tr>
<tr>
<td>10</td>
<td>□</td>
</tr>
</tbody>
</table>

Figure 27: Proof of First Requirement: I $\rightarrow$ P

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
1 | $c = i \cdot b$ | P
2 | $a \geq i$ |
3 | $i < a$ | C
4 | $c + b \neq (i + 1) \cdot b \lor a \geq i + 1$ | negated wp(P,S)
5 | $x \cdot y + y = (x + 1) \cdot y$ | Axioms
6 | $x \nleq y \lor y > x$ |
7 | $x \nleq y \lor x \geq y + 1$ |
8 | $x = x$ |

9 | $i \cdot b + b \neq (i + 1) \cdot b \lor a \geq i + 1$ | (1p, 4)
10 | $(i + 1) \cdot b \neq (i + 1) \cdot b \lor a \geq i + 1$ | (5p, 9)
11 | $a \nleq i + 1$ | (8, 10)
12 | $a > i$ | (3, 6)
13 | $a \geq i + 1$ | (12, 7)
14 | $\square$ | (11, 13)

Figure 28: Proof of Second Requirement: $P \land C \rightarrow wp(P,S)$

1 | $c = i \cdot b$ | P
2 | $a \geq i$ |
3 | $i < a$ | negated C
4 | $c \neq a \cdot b$ | negated O
5 | $x < y \lor y < x \lor x = y$ | Axioms
6 | $x \nleq y \lor x \nleq y$ |

7 | $a < i \lor i = a$ | (3, 5)
8 | $a \nleq i$ | (2, 6)
9 | $i = a$ | (7, 8)
10 | $c = a \cdot b$ | (9p, 1)
11 | $\square$ | (10, 4)

Figure 29: Proof of Third Requirement: $P \land \neg C \rightarrow O$
true just before the loop is entered, and the last proof shows that the loop invariant is true once the loop has exited. The middle proof shows that the loop invariant is true for every iteration of the loop. Of course, these proofs can only test if an expression is the invariant or not. The task of finding the invariant is a much more difficult task.

Heuristic Methods

To find a loop invariant by hand, one must implement a selection of known heuristics. In The Science of Programming by Gries and Conway (see [8]), several heuristics for generating a loop invariant from the output assertion are suggested.

1. Delete a conjunct from the output assertion.

2. Replace a constant in the output assertion with a variable.

3. Enlarge the range of a variable in the output assertion.

4. Add a disjunct to the output assertion.

By applying these methods, it can be possible to find a loop invariant, but the method is not perfect. It fails to take into account the input assertion, the condition of the loop and any code in the loop to find a loop invariant. This method also relies on the experience of the person searching for the invariant. For difficult programs, this heuristic method could be completely fruitless. With an automated method, the computer could apply a search method and find the loop invariant with minimal human effort.

Genetic Method

In order to automate the search for loop invariants by applying a genetic algorithm, one must define the genome representation of the data, the genetic operators, and
the objective function. It is not immediately clear how any of these things should be
done. Because a logic formula cannot be easily described by any of the genome data
types supplied with GAlib, we had to develop our own.

Logic as a Genome

Every logic expression used by the genetic algorithm is entered in clause normal form
(CNF). Each clause of a CNF logic expression is a disjunction of one or more literals.
A literal is a predicate that may or may not be negated. Each clause in the CNF
is conjuncted together. In first-order logic, a predicate can have any number of pa­
rameters. These parameters can be constants, variables, or functions. The predicates
defined for this problem are the equality predicate and the greater than predicate.
The less than, less than or equal, greater than, and not equal predicates can all be
translated into equality and greater than predicates. Functions, as well, can have
any number of parameters. Because variables in first-order logic can be existential or
universal, all the existential variables are replaced with Skolem functions.

These formula elements are stored in a series of linked-list objects. There is a
list of clauses, and each clause has a list of literals. Each literal holds the name of its
predicate and whether the predicate is negated or not. Also each literal has a list of
parameters which may be functions, constants, or variables. Every function, literal,
and constant keeps track of its name. Any function has a further list of parameters,
which may contain other functions with parameters.

Every function, predicate, constant, and variable has a unique name. In the
implementation, every predicate and function should have the same number of pa­
rameters wherever it appears in a logic expression.
Genetic Operators

Because of the complex nature of the genome data, the genetic operators are complicated as well. The mutator, for example, has seventeen unique ways of mutating a genome. Each of these seventeen mutations has its own probability of occurring. The crossover and initializer have many different possibilities as well.

The seventeen available mutations are designed to modify the genome on large scales as well as small scales. There are mutations that add a clause from the input and output assertions or loop condition, and remove a clause from the genome. One mutation will merge two clauses in the genome into one, effectively changing a conjunct into a disjunct. Likewise, a single clause may be split into two clauses, changing a disjunct to a conjunct. There are mutators that modify literals as well. Literals can be added or removed from a clause, they can be renamed to another literal with the same number of parameters, they can have their signs changed, and they can have their parameters swapped.

There are nine different mutations on the level of functions, constants, and variables – collectively called terms. These mutations can occur at any level of nesting in the logic expression. A random function from the assertion data or program code can be added somewhere in the expression. The number one can be added or subtracted from any term. Any variable in the expression can be instantiated to a constant from the program data or assertions. Likewise, a constant can be exchanged with an existing variable or replaced with a new one. Any term can be renamed. A random function can be replaced with one of its parameters, and a binary function can have its parameters swapped.

Each of these mutators has its own probability of occurring, and can even have zero probability of occurring if desired. For a given individual, it is possible that some of the above mutations cannot be applied. For example, if the genome being mutated has only one clause, the merge clause mutation will not affect it.
The initialize operator takes advantage of the complex functionality of the mutator. It randomly selects a clause from the assertions or loop condition and mutates it a given number of times. Occasionally this will produce an expression that can be reduced to a true or false. The objective score for these expressions is always zero. Since the mutators are loosely based on the heuristics mentioned above, we hoped the genomes produced in this way would match the genomes produced using the heuristic method.

The crossover operator has three methods of crossover, but it will examine the two parents before crossover to find similarities between the two. In this way, we hoped the crossover of two expressions would result in more meaningful children. On a macroscopic scale, the crossover operator will exchange a clause between the two parents and generate two children. Two literals may also be swapped between a clause in each parent, and on the most microscopic level, two terms may be crossed.

Each of these crossovers has its own probability of occurring. In addition to these probabilities, there is a probability that the crossover operator will check to find a similar location in the two parents where a crossover might have the most meaning. It will look to crossover two clauses of the same length, crossover two literals with the same predicate, or in a predicate in each parent of the same type, it will crossover two parameters. If a genome contains only one clause, the clause crossover is meaningless. If all the clauses in a genome only have a single literal, literal crossover is also meaningless. Unlike these two cases, term crossover can always occur.

Calculating the Objective Score

The most difficult aspect of the representation of the search for loop invariants as a genetic algorithm is to determine how each genome is to be scored. There are three proofs the GA can use to test each genome, but these only allow for scores of zero, one, two, or three. This is not nearly enough of a spread in scores for the GA to evolve
a solution. In addition to these three proofs, if a given proof has any conjuncts in its conclusion, the conjuncts can be split into additional proofs. This only increases the number of unique scores by one or two, still not enough for the GA to work correctly.

The only other way to increase the spread of objective scores is to somehow score a failed proof. We attempted to score failed proofs by counting resolvants and paramodulants that might lead to a proof. In other words, we counted resolvants and paramodulants with the conclusion expression and resolvants and paramodulants with clauses other than axioms. We also counted the number of literals produced in such resolvants and paramodulants. Each of these counts is given a weight, and the weights are summed for each proof. A correct complete proof always has a score of one, so that 3.0 is the maximum objective score. Any failed proof always is scored below 1.0.

Theorem Prover

It is known that proving first-order logic is an undecidable problem. This means that when the proof is satisfiable, the theorem prover could run forever without returning a solution. For this reason, the GA used a theorem prover that employed lock-resolution and lock-paramodulation. In this type of theorem proving, every literal is given a number and a resolution or paramodulation can only occur between literals with the smallest number in two clauses. This prevents the theorem prover from generating too many unnecessary clauses in the proof. We also put a limit on the number of clauses the proof can generate, the maximum number of literals a clause can contain and a maximum depth of nested functions.

Loop Prover

Figure 30 shows the user interface for the Loop Prover program, the software we
developed in our attempts to search for loop invariants. Because of the large number of parameters and settings in this software, we used a menu-driven interface for the entry of data. The buttons on the toolbar can be used as well as the menus. The user can modify all of the crossover, mutation and initialization parameters (see Figure 31, Figure 32, and Figure 33). In the initialization settings dialog, the "Heuristic Initialization" was not used. The settings for the objective scoring can be modified as well, as shown in Figure 34.

Other functionality of this software is that it can read the input and output assertions, loop code and loop condition from a file. We also programmed a feature that allowed us to test the theorem prover, genetic operators, and weakest precondition operator on the files. In the main area of the window, the genome and its resulting score after each proof is displayed as the GA evolves its solution.

Once the GA is activated, the grayed buttons in the toolbar also become active. With these buttons, the user can set the GA to evolve until it finds a solution, evolve a single generation, pause the evolution, or reset the evolution.
Initialization Settings

Initial Mutations: 10

Initializer:
- Random Initialization
- Heuristic Initialization

Figure 31: Initialization Settings Dialog

Mutation Probabilities

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Add a random function:</td>
<td>0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add one to a term:</td>
<td>0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtract one from a term:</td>
<td>0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiate a variable:</td>
<td>0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rename a term:</td>
<td>0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove a function:</td>
<td>0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swap parameters of a binary function:</td>
<td>0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Change a constant to a variable:</td>
<td>0.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 32: Mutation Settings Dialog

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Results

We tested the Loop Prover program with two simple examples. We used the iterative multiplication example, shown in this chapter, and an iterative method for calculating factorials. In both cases, after running the GA for extended periods of time, the GA was unable to find even partial solutions to either problem. Since the other examples we had planned to use to test the GA were considerably more complex, we halted our testing in favor of working with neural networks.

We believe that our inconclusive results come in large part because we have no accurate method for measuring the correctness of proofs. Also, because an approximate solution of a loop invariant has no value, the GA had to evolve a perfect solution in order to succeed. It may be possible with alterations of our scoring method and genetic operators to achieve a genetic algorithm capable of finding a loop invariant, but we believe that the GA is not the correct tool to use for this problem.
### Objective Settings

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight of proof 1</td>
<td>0.25</td>
</tr>
<tr>
<td>Weight of proof 2</td>
<td>0.25</td>
</tr>
<tr>
<td>Weight of proof 3</td>
<td>0.5</td>
</tr>
<tr>
<td>Weight of non-axiom resolvents</td>
<td>0</td>
</tr>
<tr>
<td>Weight of non-axiom predicates in resolvents</td>
<td>0</td>
</tr>
<tr>
<td>Weight of non-axiom paramodulents</td>
<td>0</td>
</tr>
<tr>
<td>Weight of non-axiom predicates in paramods</td>
<td>0</td>
</tr>
<tr>
<td>Weight of resolvents with conc.</td>
<td>0.2</td>
</tr>
<tr>
<td>Weight of preds in conc resolvents</td>
<td>0.6</td>
</tr>
<tr>
<td>Weight of conc paramod</td>
<td>0.025</td>
</tr>
<tr>
<td>Weight of predicates in conc paramods</td>
<td>0.175</td>
</tr>
</tbody>
</table>

**Figure 34: Objective Settings Dialog**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHAPTER 4

TRAINING AN ARTIFICIAL NEURAL NETWORK USING GENETIC ALGORITHMS

An artificial neural network (ANN) is a computational mechanism whose design is inspired by the function of the brain. A neural network consists of a graph of nodes, or axioms, and weighted connections between them. A subset of these nodes is the input, which is "stimulated" with floating-point values. Another subset is the output, from which the results of the network's calculation are read.

Networks may be feedforward or feedback, and they may have a layered structure or an amorphous structure. Feedforward networks contain only directional connections that propagate through the network from the input nodes to the output nodes. In a feedback network, the directed edges have no limitations on where they connect. In a layered network, the neurons are separated into a number of distinct layers. The input forms the first layer, and the output forms the last layer. The other remaining layers are called hidden layers. The connections can only connect adjacent layers. In our studies, we only concerned ourselves with feedforward layered networks, although our software was capable of analyzing non-layered networks as well. In the following chapter, the structure of networks is of primary concern, as we use a genetic algorithm to find adequate architectures for solving problems.

In feedforward networks, once the input neurons have been stimulated, the data is propagated to the next layer via the connecting edges. Each of these edges has a floating-point weight. At every node in the next layer, the incoming values are multiplied by the weights on the connecting edges and added together. If the total...
sum is larger than that neuron's threshold value, the node fires and further propagates the signal. The threshold value is a floating-point value, and in our studies, the threshold was set to 0.0. If an alternate threshold value is needed, an additional node is supplied in the input layer. This node is 1.0 for every data point sent to the network. The weights of the connections to this node act as threshold values.

Before a neural network can be applied, it must be trained. Training consists of applying a set of data whose correct output is known in advance to the network. The edges of the network should be then modified so that it can correctly reproduce the correct output of the training set. This is usually a time consuming process and requires several passes of the data set in order for the network to work correctly. The most common for training networks is called back propagation. In this paper, however, we will show that genetic algorithms are quite successful at training networks as well.

Back Propagation

To train a neural network with back propagation, one begins with a network containing random weights. The training data is entered into the network, one point at a time. Each time the network is stimulated, the output is examined for correctness. If the network gives an incorrect result, the weights in the network are modified slightly to correct the problem. The entire training data is passed through the network and this is repeated with the same data many times, or epochs, until the network has been satisfactorily trained. The number of epochs can sometimes be exponential in the number of input nodes.

In back propagation, the weights in the network are updated based on the error at the node and the learning rate, $\alpha$. At an output node $i$, the weight $W_{i,j}$ on the edge between nodes $i$ and $j$ is calculated by the formula:
\[ W_{i,j} \leftarrow W_{i,j} + \alpha \Delta_i \]

Where \( \Delta_i \) is equal to the correct value at the output node minus the value generated by the network. These errors are then propagated to the next layer by summing the error at each output node times the weight of each edge that connects the output to the hidden node \( j \). Then, with the same formula for the output nodes, the weights incoming to \( j \) are modified. This process is repeated for every hidden node in the network.

In effect, this method of training a neural network performs a gradient descent on the error space. The error space is a function in a space where each axis is a weight in the network. The value of the function is the error at that point. The back propagation method effectively examines the partial derivative in each axis and moves the current network slightly downhill on each axis. Eventually the network weights should reach a minimal point in the space. This is simply a local search on the error space.

This method of training networks because of this is not always perfect. It is possible that the initial weights are arranged in such a way that the network is not trainable. The network may enter a point that is a local minimum and it will be unable to find a better solution. This method is also very time consuming, as the number of epochs to train the data can be an exponential function on the number of inputs. It is for these reasons that we use genetic algorithms to train neural networks.

**Genetic Method**

The genetic method for evolving weights of an artificial neural network uses the crossover and mutation operators to modify an initial randomly generated popula-
tion of weighted neural networks. The initial population is generated by randomly selecting weights between a specifiable maximum and minimum value. The genetic operators are as implemented by Montsma and Davis (see [16]). The mutation operator randomly selects a node in the network and modifies the weights of the node's incoming edges. In Figure 35, the node number 5 is chosen. The operator adds a random value between the minimum and maximum values, in this case, -1.0 and 1.0, to each of the incoming edge weights.

Similar to the mutation operator, the crossover operator picks a random node and copies the weights of the incoming edges in one parent and inserts them into the child. The remaining weights come from the other parent. In the example of Figure 36 the weights from parent one are copied into the child. The only weights that come from parent two are the weights of the edges incoming to node four.

The neural net weights are stored in a one-dimensional array. The incoming edge weights for each node are stored in order of the node in the array. A second array is used to store where the edges originate. The training data used to train the network is also stored within a data structure. The objective score is calculated by stimulating the network with each item of the training data. For each incorrect output node after the input has propagated, one is added to the error. The square of the error for each member of the training data is summed and returned as the objective score.
**Brain Evolver: Evolving ANN Weights**

We wrote the Brain Evolver program to evolve weights on artificial neural nets and to evolve neural net structures. The second feature is described in detail in chapter 5. Figure 37 shows the user interface for this software. The text region to the lower left of the dialog box is where the training data is displayed. Next to that is shown the resulting weights after the evolution. The large area shows the network structure with the weights of each edge. Positive weights are drawn with black lines and the negative edges are drawn in the light gray, appearing red on a color display. The line thickness reflects the size of each weight. Selecting a member of the training data on the left shows which nodes fire in the network display (see Figure 38). The nodes that have fired are shown in gray or green on a color display.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 37: The User Interface for Brain Evolver

Figure 38: User Interface Displaying a Network With Fired Nodes
By clicking on the GA settings, the dialog box shown in Figure 39 is activated. From this window, the population size and number of generations to evolve can be modified, along with the probabilities of crossover and mutation. This software uses only the steady-state model for the genetic algorithm. The "p replacement" edit area sets the percentage of the population to be replaced each generation. The "Min init/mut value" and "Max init/mut value" specify the minimum and maximum values used in initializing the population and for mutating an individual as described above.

Before weights in a network are evolved, the network structure and training data must be entered. This is accomplished by selecting the "Load NN Struct" and "Load Training Data" buttons. Once loaded, each member of the training data is
Figure 40: The Contents of the Training Data File for XOR displayed. The “Evolve NN” button begins the evolution process; each generation the network weights with the lowest score are shown in the network display. Once paused, by clicking the “Pause” button, the user can scan through each member of the population with the spinner button above the display.

To save the weights evolved in the network, click on the “Save NN” button. These can be loaded at a later time with the “Load NN” button to test it on other data sets.

This software was developed to work on neural network structures with any number of nodes. There may be any number of output or input nodes in the network. In addition, the training data may be of any length. We developed simple file formats for both the neural net structures and training data. The training data files contain integers representing the number of entries in the training data and the number of input and output nodes needed. Following these quantities is the training data. On a single line, appearing first is the input nodes as floating points, and then the desired output node results for that input data, zeros or ones. Figure 40 shows an example of what the training data file for the XOR operator looks like. The neural net structure is saved as a matrix and is discussed in detail in the following chapter.

Results

We tested the genetic algorithm’s ability to train neural networks on a selection of
simple x-or operations, and a more complicated geometric problem. In all the tests we performed, the genetic algorithm proved that it was capable of training networks to solve these problems.

The first test we performed was for generating the weights in a simple neural network to calculate the x-or function. The x-or problem stimulates the network's two input nodes with ones or zeros. If the x-or of two input bits is one, the network should fire. The resulting network evolved by the genetic algorithm is shown in Figure 41. This weighted network was evolved in seconds, using a small population of fifty and only twenty generations. The GA always found perfect solutions to the x-or problem.

The slightly more complicated x-or3 problem evolved its solution in fifty generations with a population of 200. The x-or3 problem simply counts the ones on the input; if the number of ones is even, the output should fire. Otherwise, it should
not fire. The GA found networks with perfect scores in about one out of two runs. Figure 42 shows the weighted network the GA evolved. The network architectures used for both of these x-or problems are the smallest possible layered networks for solving these problems.

The third application we used to train neural nets uses floating-point valued inputs to the network. In this application, the network received three floating valued inputs to the network, representing $x$, $y$, and $z$ coordinates. A fourth input to the network is always one and allows the network to evolve threshold values. The network should calculate if the point lies within a three-dimensional diamond in space, satisfying the equation $|x| + |y| + |z| \leq 2.0$. If the point satisfies the equation, the output should fire.

We generated a set of 200 random points with values between -2.0 and 2.0. To ensure that approximately the same number of points fell inside the region and outside the region, we modified the probabilities of the random number generator to favor smaller numbers. We did this by squaring the randomly generated number.
between one and zero then multiplying it by two. With equal probability, we made the resulting value positive or negative. We generated one set of such numbers to train the data, and we made a second set of different values to test the network once it had been evolved.

By hand, we found a network with eight hidden nodes which would solve this problem (see Figure 43). Each of the eight hidden nodes determines, via the weighted edges connected to the input nodes, on which side the point lies of one of the eight points that define the facets of the diamond. The output node then uses this information to ensure that it lies on the inside of the diamond. We believe, although we have not proved this, that this is the smallest possible network capable of solving this problem.

We added one node to this structure to assist the genetic algorithm in its evolution of a solution. This ninth hidden node has a connection from the threshold input-node and is connected to the output node. This additional node can be used by the network and the GA to find a threshold value for the output node. In our
Figure 44: Weighted Neural Network for the 3-D Diamond Evolved by the GA

network, we did not need this extra node because we designed the edge weights in such a way that the threshold of the output node was zero. With this additional node, that restriction is lifted and the GA is free to evolve threshold values at the output node.

The weighted neural network in Figure 44 is the network found by the GA using the 200 training points. We used a population of 1000 and evolved for 1000 generations; this took about four hours to complete the evolution. The network evolved correctly determined the location of 193 of the 200 points and incorrectly placed seven points. When we scored the same network with the second set of 200 points, it correctly placed 163 of the points and was incorrect for 37 of the points.

This accuracy shows the potential for genetic algorithms in the development of weighted artificial neural networks. We believe that genetic algorithms can be used to find neural network weights in a variety of applications. Even though the examples shown here are simple, their dramatic success illustrates how effective genetic algorithms are at generating weighted neural networks. This, along with the genetic algorithm's ability to find network architectures, as will be shown in the following chapter, makes them useful tools in neural net research.
CHAPTER 5

DEVELOPING NEURAL NETWORK ARCHITECTURES
WITH GENETIC ALGORITHMS

In previous work with neural networks, it has been necessary to guess the network structure to use when solving a problem. Through past experience, one could select the number of nodes, connections, and layers to include in a feedforward network. In our work, we have designed software using GAlib that can find neural network structures which are best adapted to solving problems. In order to do this, the network architecture must be represented as a genome, and there must be some way of calculating a fitness score for a given architecture.

We explored two ways of representing ANN structures in a genome type. The simplest is to represent the ANN as a graph in a two-dimensional connection matrix. The second is to represent the connection matrix in a graph-generation grammar. Both methods are explained in detail below.

To calculate the fitness scores for each genome, we employed a second genetic algorithm to evolve weights as described in the previous chapter. This GA will evolve its population for a pre-specified number of generations for each individual in the population. The best objective score after the last generation is the objective score for the network structure tested. In addition to this score, additional weights can be added for number of nodes, connections and layers. In this way, the genetic algorithm will find the network architecture with not only the best structure, but also the least complicated structure to solve the problem at hand.
Network Construction from a Two-dimensional Matrix

The neural network structure is stored in a two-dimensional array representing a directed graph. Each row and column represent nodes in the network, and the edges between them are marked by ones in the matrix. If a one appears on the diagonal, the node of that column and row will be a part of the network. Otherwise if there is a zero, that node will not appear. A one in the matrix defines an edge originating from the column node and incoming to the row node. This is based on the work of Miller, Todd and Hedge (see [12]). Naturally, because this research deals only with layered feedforward networks, not all matrices taken as directed graphs represent valid neural networks. Therefore, the matrix was parsed in such a way that some of the edges that appear in the matrix are not a part of the final network.

For a training data set requiring \(i\) input nodes, the first \(i\) nodes in the matrix with ones in their diagonals are chosen as the input nodes. In the same way, for \(j\) output nodes, the last \(j\) nodes are selected. Any edges incoming to the input nodes are ignored, as well as any outgoing edges from the output. In a breadth first manner, nodes are added one layer at a time. First, any nodes with incoming edges from the input nodes are inserted. All the edges originating from the inputs to these nodes become part of the network. Any other edges in the matrix incoming to these nodes are ignored. This prevents any edges between nodes in the same layer and any edges incoming from nodes not yet in the network. The next layer contains the nodes with incoming edges from the previous layer. They are added in the same way. The algorithm continues to add nodes and layers until no more can be added. Any connections between the last layer added and the previously chosen output nodes are then inserted, completing the network architecture. Figure 45 shows the network resulting from a the two-dimensional matrix shown.

The genomes for evolving networks from such a matrix are two-dimensional binary strings. The crossover operator divides the parent matrices into four quadrants.
Matrix

1 0 0 0 0
0 1 0 0 0
1 1 1 0 0
1 1 0 1 0
0 0 1 1 1

Figure 45: Translation of a 2-D Matrix into a Neural Network With 2 Inputs and 1 Output

by randomly selecting locations to slice the array vertically and horizontally (See Figure 46). The new children result from taking the upper left and lower right quadrants from one parent and merging them with the lower left and upper right quadrants from the other. The second child is formed with the unused quadrants of the two parents. The mutator simply flips bits with the probability of mutation. The initializer generates a matrix of uniformly generated random ones and zeros.

Using a two-dimensional array as a graph representation of network architectures is an effective representation for a genome. Its greatest drawback is that the genome of a matrix that can represent an n node network takes \( n^2 \) space. Graph-generation grammars can be used to reduce this length, as well as introducing the possibility of beneficial repeated structures in the network.

Neural Network Structures as Graph-generation Grammars

Work using graph-generation grammars to evolve artificial neural network structures.
was initially done by Kitano (see [10]). A graph-generation grammar is a set of rules that define a two-dimensional matrix. In the graph-generation grammar model we used, each rule in the grammar translates on non-terminal symbol into four non-terminals or four terminals (see Figure 47 and Figure 48). Each non-terminal only has one rule in the grammar, and therefore only one graph can be produced from a single graph grammar. The terminal symbols are the letters a–p, each representing a $4 \times 4$ matrix. For example, $a \rightarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $b \rightarrow \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$. A rule that does not generate a matrix of four terminals must generate a rule of four non-terminals. This rule can arrange its four non-terminals, labeled $A-D$, in any combination. As an example, in the rule $A \rightarrow \begin{bmatrix} D_A \\ C_A \end{bmatrix} \begin{bmatrix} A_A \\ C_A \end{bmatrix}$ where $C$ occurs twice and $B$ never occurs is a valid rule. The subscript letter used in this representation is to clarify that the rules $A$, $A_A$, and $A_B$ all define different sub-matrices. This is to avoid confusion, but the subscripts are not encoded by the computer. Rules like these are applied from the start rule and eventually define a two-dimensional matrix of ones and zeros.

The Graph Grammar as a Genome

This representation of graph-generation grammars allows for the grammar to be rep-
Figure 47: Example of a Graph-generation Grammar

\[
S \rightarrow A \begin{array}{c} B \end{array} A \quad A \rightarrow D_A \begin{array}{c} C_A \end{array} \quad B \rightarrow A_B \begin{array}{c} A_B \end{array}
\]

\[
A_A \rightarrow b \begin{array}{c} c \end{array} \quad C_A \rightarrow j \begin{array}{c} h \end{array} \quad D_A \rightarrow d \begin{array}{c} c \end{array} \quad A_B \rightarrow a \begin{array}{c} a \end{array} \quad B_B \rightarrow d \begin{array}{c} a \end{array}
\]

\[
a \rightarrow 0 \begin{array}{c} 0 \end{array} \quad b \rightarrow 0 \begin{array}{c} 0 \end{array} \quad c \rightarrow 0 \begin{array}{c} 0 \end{array} \quad d \rightarrow 0 \begin{array}{c} 0 \end{array} \quad h \rightarrow 0 \begin{array}{c} 1 \end{array}
\]

\[
j \rightarrow 1 \begin{array}{c} 0 \end{array} \quad p \rightarrow 1 \begin{array}{c} 1 \end{array}
\]
Figure 48: Network Produced by the Graph-generation Grammar
resented in the genome in a recursive fashion. The genome is an array of integers: the integers zero through fifteen represent the letters a through p. To keep initialization and mutation methods simple, the letters A through D are represented by the integers zero through fifteen modulus four. Each rule is contained in four positions in the array, the first two are the top left and right values and the last two are the bottom left and right. The rules are ordered in such a way that they can be easily accessed by a recursive function.

The first four locations in the array hold the starting rule. The remaining array is divided into four equal sections, each section representing the letters A, B, C, and D in that order. In our example, the array contains 84 elements and the four sub-arrays contain 20 elements each. The rules describing each of these letters are located in the first four positions in the sub-arrays. The first sub-array for the letter A, from element 4 to element 23, is again divided into four sections and represent the letters A_A, B_A, C_A and D_A. This division continues until the sub-arrays contain only four terminals. This is done similarly for every letter, A through D, whether they appear in the final structure or not. Once the sub-array contains only terminals, the graph-grammar generates an array of ones and zeros. The resulting matrix can then be analyzed as described above.

Figure 49 shows a portion of the array representing our example. The start rule is encoded in the array elements 0 through 4. The rule for A is encoded in elements 4 through 7. In elements 8 through 11 is the rule for A_A. The rule for B starts at location 24. Next to that rule is the rules for A_B and B_B. There are rules for A through D at every level, although they are not shown here and they do not appear in the final matrix. These unused portions of the genome are carried along and may appear after the genome is mutated or in the child of a crossover.

The initializer simply sets each value in the array to a random number between zero and fifteen and the mutator changes a position in the array to a random integer
between zero and fifteen as well. Because the genome is divided into rules four integers wide, the crossover selects one to ten of these rules from one parent and replaces those rules in the second parent to generate a new child. There are other choices of crossovers, but this one has been effective in our tests.

Brain Evolver: Evolving ANN Architectures

The Brain Evolver program was designed to not only genetically evolve weights for a neural network, it also can evolve network structures using graph-generation grammars and two-dimensional matrix genomes. Both methods employ a second genetic algorithm to calculate their fitness scores for each genome. The settings for this genetic algorithm, along with specifications for the matrix size and which genome type to use are found by clicking on the “Arch Settings” button. This brings up the architecture settings dialog as shown in Figure 50.

The weights shown here are multiplied by the number of nodes, edges and layers respectively, and then added into the objective score. This results in a higher objective score for more complex network architectures, driving the GA to find the smallest architecture that solves the problem put forth. For graph grammars, the number of nodes can only be odd powers of four larger than or equal to eight. Any value entered that is not a valid size will be changed to the next-largest odd power of two before evolution begins. The values entered near the bottom of the dialog box change the

Figure 49: Array Representing a Section of the Graph-generation Grammar
parameters for the objective GA. For the architecture genetic algorithm, the settings are changed in the GA settings dialog as shown in Figure 39.

Before any evolution can begin, the user must load the training data by clicking “Load Training Data.” Once this is done, to start the evolution, click the “Evolve Arch” button shown in Figure 37. The “Pause” button temporarily stops evolution and the “Reset” allows the user to reset the population and reinitialize and evolve a new solution. The “Individual No.” spinner allows the user to examine each architecture in the population. The score and current generation along with the best network structure is displayed in the large display area to the lower right. Once a structure has been evolved, it can be saved with the “Save NN Struct” button.

Results

Our work to evolve neural network architectures with genetic algorithms proved successful. Our first work was done using the two-dimensional matrix genome. To test the ability of the genetic algorithm to find neural network architectures using this genome, we used the x-or and x-or3 data sets as described in the previous chapter.
Tests with the x-or data set always produced the known best architecture for the problem (see Figure 51). The GA used a population of forty individuals and found the solution after only ten generations. The GA used to calculate objective scores used a population of twenty and evolved for twenty generations. The additional weights on number of nodes, edges, and layers were all set to 0.01.

Tests with the x-or3 training data produced the best architecture of that problem also. Notice that in Figure 52 the resulting architecture has three hidden nodes, but one of the nodes, number ten, is connected to only one input. Because of the additional weight for edges, the GA was able to eliminate the extra edges to this node. This architecture was evolved after running for 100 generations with a population of fifty networks. The maximum number of nodes in the network was set to twenty. The objective GA had a population of fifty and ran for thirty generations. The additional weights were the same as the x-or run.

The graph-grammar genome found the same architectures for the x-or and x-or3 problems to those found by the matrix genome. The graph-grammar found both of these solutions in about ten generations. It found the x-or3 architecture in con-
Figure 52: Architecture for X-or3 Found by the GA With the Matrix Genome

considerably less time than the matrix genome, presumably because of its symmetric structure, which is more easily represented by the graph grammar (see Figure 53). Because the graph grammar produces a matrix with dimensions of powers of two, the maximum number of nodes was set to 16 for both examples. Other than this, we used the same parameters as above to find these networks.

As shown by these examples, a genetic algorithm can find good architectures for artificial neural networks. Both the matrix genome and the graph-grammar genome are acceptable representations of network structures. Our testing, however, has not yet shown the benefits of one genome type over the other. It is true that the grammar is a smaller representation, but it is limited by the fact that the maximum number of nodes must be set to a power of two. It was also able to quickly find the architecture for the x-or3 problem, and may be more suited to finding architectures of networks that contain a degree of symmetry. We do believe that this application shows the power of genetic algorithms with neural networks.
Figure 53: Architecture for X-or3 Found by the GA With the Graph Grammar Genome
APPENDIX A

CELLULAR AUTOMATON PROGRAM

#include <stdio.h>
#include <iostream.h>

#include <ga/GASSStateGA.h>  // We are using a steady state GA
#include <ga/GA1DBinStrGenome.h>  // With 1-D binary string genomes

const int ICwidth = 101;  // Size of strings analyzed by the CA
const int ICiterations = 150;  // Number of times the CA is run on a
                               // string
const int ICquantity = 100;  // Number of strings used to score a CA
const float DistribRate = 0.75;  // Rate the initial conditions become
                                 // more difficult

const int Generations = 100;  // Number of generations to evolve
const int Population = 130;  // GA population

const int R = 2;  // Radius of the window a CA can use
const int GenSize = 32;  // Number of possible CA rules, defining
                         // the size of the genome

const float Rho = 0.5;  // The CA should detect if a string has
                       // more or less than Rho ones. Usually
                       // this is 0.5.

int IC[ICquantity][ICwidth+1];  // Array of strings used to test the CAs

float Objective( GAGenome &);  // This is the declaration of the
                                 // objective function. The definition
                                 // comes later in the file.

void getIC(int);  // Function to generate initial
                  // conditions randomly.

int runIC(int[], int, int);  // Applies a CA to an IC string
int one_count(int[]);  //Counts 1s in a string to see if the
   //CA worked.

int main(int argc, char **argv)
{
   GA1DBinaryStringGenome genome(GenSize, Objective);

   GASteadyStateGA ga(genome);   //Declaration of the GA object
   ga.populationSize(Population);  //Setting the GA parameters
   ga.nGenerations(Generations);
   ga.pMutation(0.005);
   ga.pCrossover(0.85);
   ga.pReplacement(0.80);

   getIC(0);   //get the first ICs
   cout << "Generation #" " 1 " endl;
   ga.initialize();   //Initialize the population

   //This for-loop steps through each generation of the GA. Each
   //generation, the ICs become a little more difficult.
   for(int i=1;i<Generations;i++)
   {
      getIC(i);   //Gets a new set of slightly more difficult ICs.
      cout << "Generation #" " i+1 " endl;

      //Forces the GA to reevaluate each CA in the population.
      ga.objectiveFunction(Objective);
      ga.step();  //Evolve one generation of the GA
   }

   //Print out the best genome that the GA found.
   cout<<"The GA found:
"<<ga.statistics().bestIndividual()<<"\n";
   return 0;
}

//This function generates a random set of ICs each generation. It
//slowly becomes closer to uniform and therefore more difficult as the
//GA progresses.
void getIC(int cur_gen)
{
   float prob;
   for(int i=0; i<ICquantity; i++)
   {
      //Find the probability with which the IC string is to use when
      //randomly placing ones and zeros. The fist string in the set of
//ICs should contain mostly zeros. In the middle of the set, the
//distribution of ones and zeros should be close to uniform, and
//near the end of the set, the strings should contain mostly
//ones.
if (float(Generations) > cur_gen*DistribRate)

    //This equation determines the probability based on the
    //constants Rho and DistribRate, the current generation, and
    //which IC in the set is being generated.
    prob = (Rho*ICquantity +
            ((float(Generations) - cur_gen*DistribRate)
             /Generations)*(i - Rho*ICquantity)
             + 2*Rho) / float(ICquantity + 2);
else
    //Use Rho when the equation above gives negative values.
    prob = Rho;

    //Generate the string with the calculated probability.
    for (int j=0; j<ICwidth; j++)
    {
        IC[i][j] = GAFlipCoin(prob);
        cout « IC[i][j];
    }
    //Count the Is and save the value along with the IC string.
    IC[i][ICwidth] = one_count(IC[i]); cout « endl;
}
cout « endl;

//Count the ones in an array.
int one_count(int X[])
{
    int i, one_ct = 0;
    for (i=0; i<ICwidth; i++)
    {
        if (X[i] == 1)
            one_ct++;
    }
    return one_ct;
}

//Calculate the score for a CA
float Objective( GAGenome & g)
{
float score = 0.0;
int i;
int gen_array[GenSize];

//Type-cast the GAGenome to a GA1DBinaryStringGenome
GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;

for (i=0; i<GenSize; i++)
{
    cout << genome.gene(i); //Output the genome contents
    gen_array[i] = genome.gene(i); //Copy it into an array
    if(i==100)
        cout << endl;
}
cout << endl;

//Run the CA stored in the genome on each IC in the set. Each time
//the CA works on a string, one is added to the objective score.
//There is no "partial credit" for a CA that almost works.
for(i=0; i<ICquantity; i++)
    score += runIC(gen_array, i, IC[i][ICwidth]);

cout << endl << score << endl;
return score;

//This function takes a sing CA and a single IC. It applies the CA to
//the IC for the specified number of times and checks to see if it
//correctly generated a string of all ones or zeros. If the CA worked,
//1 is returned. Otherwise, 0 is returned.
int runIC(int gen_array[], int ic, int ones)
{
    int k;
    int gene_index;
    int final_bit, one_ct;
    int newIC1[ICwidth], newIC2[ICwidth];

    //Determine whether all ones or all zeros should be generated. If
    //all ones, final_bit is set to one. If all zeros final_bit equals
    //zero.
    final_bit = (float(ones)/float(ICwidth) > Rho);

    //Applies the CA to the IC string. The resulting string is stored
    //in newIC1.
    for(int i=0; i<ICwidth;i++)
    {
        k=1;
gene_index = 0;

// This for loop converts the bits in the window over the IC to
// their binary equivalent and stores the value in gene_index.
for(int j=R; j>=-R; j--)
{
    gene_index += IC[ic][(i+j+ICwidth)%ICwidth]*k;
    k = k*2;
}

// Place the bit in that position in the genome in the new string.
newIC1[i] = gen_array[gene_index];
}

// Applies the CA to the new string twice. It is applied two times
// because two strings are used to store the new string.
for(int iter=1; iter<ICiterations; iter+=2)
{
    one_ct = 0;
    for(i=0; i<ICwidth; i++)
    {
        k = 1;
        gene_index = 0;

        // Again, this for loop converts the bits in the window to
        // their binary equivalent and stores the value in gene_index.
        for(int j=R; j>=-R; j--)
        {
            gene_index += newIC1[(i+j+ICwidth)%ICwidth]*k;
            k = k*2;
        }

        // newIC2 is used to store the newly created string.
        newIC2[i] = gen_array[gene_index];
    }

    for(i=0; i<ICwidth; i++)
    {
        k = 1;
        gene_index = 0;

        // Again, this for loop converts the bits in the window to
        // their binary equivalent and stores the value in gene_index.
        for(int j=R; j>=-R; j--)
        {
            gene_index += newIC2[(i+j+ICwidth)%ICwidth]*k;
            k = k*2;
        }

        // newIC3 is used to store the newly created string.
        newIC3[i] = gen_array[gene_index];
    }
}
{ 
  gene_index += newIC2[(i+j+ICwidth)%ICwidth]*k;
  k=k*2;
}

//newIC2 is used to store the newly created string.
newIC1[i] = gen_array[gene_index];

//Count the ones in the new string.
one_ct += gen_array[gene_index];

//Check to see if the CA succeeded in generating all zeros or ones
if (one_ct == ICwidth) {
  cout << final_bit;
  //If a line of all ones was created, return one
  //if final_bit = 1, 0 if it equals 0.
  return final_bit;
}
else if (one_ct == 0) {
  cout << !final_bit;
  //If a line of all zeros was created, return
  //one if final_bit = 0, 0 if it equals 1.
  return !final_bit;
}

} 

cout << 0;

//Did not converge to 0's or 1's, so return 0.
return 0;
BIBLIOGRAPHY


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


VITA

Graduate College
University of Nevada, Las Vegas

Bradley Hendricks

Home Address:
4506 Palm Grove Dr.
Las Vegas, Nevada 89120

Degrees:
Bachelor of Science, Computer Science, 1997
University of Nevada, Las Vegas

Bachelor of Science, Physics, 1997
University of Nevada, Las Vegas

Thesis Title:
Genetic Algorithms Using GAlib

Thesis Examination Committee:
Chairperson, Dr. Wolfgang Bein, Ph. D.
Committee Member, Dr. Lawrence Larmore, Ph. D.
Committee Member, Dr. Laxmi Gewali, Ph. D.
Graduate Faculty Representative, Dr. Lori Bruce, Ph. D.