CHAPTER 1

THE USE OF GALIB

GAlib is a C++ library developed by Matthew Wall (see [21])at the Massachusetts Institution of Technology designed to assist in the development of genetic algorithm applications. The library contains numerous classes that offer functionality and flexibility in the design of optimization applications with genetic algorithms. The library includes default genetic algorithm models, genome types, and genetic operators for the quick creation of simple applications, and the ability to customize GAlib for more complicated optimizations. This library was programmed so that it may be used on 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 applications were written under the Microsoft Visual C++ environment and all examples given in this paper were written in that environment.

GAlib supports several different models of genetic algorithms. The simple GA is the standard 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. The deme GA evolves multiple population and migrates individuals from one population to another. With this GA model, GAlib can run on parallel processors, evolving each population on a separate processor. It is also possible to develop a custom GA 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 quantities.

Any datatype 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. This is the function that evaluates an individual from the population and calculates 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 GAlib includes a large variety of genetic algorithm and genome types and is written with a hierarchical structure, it is simple to modify software already written with GAlib to perform new tasks.

General Overview

When programming using GAlib, one will work primarily with two classes: a genome class and a genetic algorithm class. A genome 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. GAlib supplies the two classes, but the objective function must be programmed. 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 developing an application using GAlib are to:

- define a representation
- define the genetic operators
- 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 object is used to represent a possible solution to an optimization problem. The genetic algorithm will create a population of this structure that is supplied. 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 GAGenome type. The programmer must write the new type inherited from the class GAGenome and his or her own data structure class. 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 simple, steady-state, and incremental genetic algorithms. These GA types inherit from the class GAGeneticAlgorithm. They differ from each other in the methods 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 to 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. The population object keeps track of statistics about the population as well. It keeps track of 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 to generate a new solution. 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 genome initialization function is called by the initialization operator in the population object. By default, this calls the initialization operator of each genome, but this can be customized.

The mutation operator defines how a genome is mutated to produce a new solution. 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 genetic 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. This is used to measure the difference between two members of the population and is used for some statistical measures kept by GAlib. The comparator is not a required function.

For most applications, the supplied genome types are more than adequate, but it is a simple process to write one's own genome type. In general, it is not necessary for GAlib 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 GA using GAlib, 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 objective function is passed to the genome so that it can be called from the genome when it 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 genetic algorithm itself, but instead a population of genomes is cloned from it. The GA function "evolve", line 5, can then be called to initiate and

```
/*1*/ void main()
/*2*/ {
        //Declare a single genome object, which will be duplicated by the GA.
/*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

run the GA.

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 a GA that completely replaces the population each generation with a new population created by the crossover and mutation operators. One may also wish to set various parameters that change the operation of the GA, as in Figure 2. These member functions of GA 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 maximization to a minimization.

Writing the Objective Function

The objective function is the only place where the programmer codes in the

```
ga.populationSize(popsize);
ga.nGenerations(ngen);
ga.pMutation(pmut);
ga.pCrossover(pcross);
```

Figure 2: Setting parameters for a GA

```
/*1*/ float Objective(GAGenome & g)
/*2*/ {
          //Type-casting the GAGenome to a GA1DBinaryStringGenome.
/*3*/    GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &) g;

/*5*/    float score = 0.0;

         //This for-loop sums all the ones in the genome in score
/*6*/    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

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 GAlib genome. This is true for all genome types included with GAlib. 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 class before the objective score can be calculated. Figure 3 is an example of a simple objective function. The function gives higher objective scores to those genomes with more ones.

The type cast shown in Figure 3 (line 3) creates a new variable genome of type GA1DBinaryStringGenome, which now contains the correct type for the genome 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 GA types are the GASimpleGA, the GASteadyStateGA, the GAIncrementalGA, and the GADemeGA. Each of these GA types and any customized GA inherit from the class GAGeneticAlgorthm.

The GAGeneticAlgorithm class is an abstract class and can therefore have no instantiations. This class keeps track of GA 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 been met. the function generation returns the current generation the GA is evolving.

To control the evolution of the GA, the programmer can invoke the functions "evolve," "initialize," and "step." "evolve" first initializes the GA, then evolves the population generation by generation until the termination requirements have been met. "initialize" resets the evolution and initializes each member of the population. "step" completes a single generation of the evolution.

The GASimpleGA is the simple GA 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 GA type with the "elitist" function. This flag causes the GA to always keep the single best individual in the population and inserts it into the next 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 population size. By setting the "pReplacement" parameter, the percentage of the population to be replaced each generation is set. The "nReplacement" parameter specifies the exact number of individuals to be replaced. Only one of these parameters can be set at one 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 is two.

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 included with GAlib. 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 GAGenomes 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.

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 the crossover 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.

The classes 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.

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," "depth," and "width" set the size of the array in the first, second, and third dimensions respectively.

Also included with GAlib is the 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 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 made to order. Even a genetic algorithm object can be customized. The following section explains further how the customization process works with GAlib.

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++.

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

```
/*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

Customizing the Mutation Method

Often times, one will desire 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 to assign the custom mutator for use in the optimization.

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 a programmer will want to define his or her

```
/*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;
        //Pick a random node from the neural network to mutate
/*9*/   node = GARandomInt(0, max_nodes - 1);

        //Mutate all the incoming connections to the random node
        //by adding a random number to their weights.
/*10*/  for(i=connection_start[node]; i<=connection_finish[node]; i++)
/*11*/     genome[i] = genome[i] + GARandomFloat(min_weight, max_weight);

/*12*/  return 1;
/*13*/}
```

Figure 5: Example of a Mutator Function

own crossover method. 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, which is 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 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" is 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

```
/*1*/ int CArrayGenome::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;
         //Type-casting GAGenomes into GA1DArrayGenomes for the parents.
/*6*/    CArrayGenome & mom = (CArrayGenome &) p1;
/*7*/    CArrayGenome & dad = (CArrayGenome &) p2;
         //Type-casting GAGenome * into GA1DArrayGenome * for children.
/*8*/    CArrayGenome * bro = (CArrayGenome *) c1;
/*9*/    CArrayGenome * sis = (CArrayGenome *) c2;

         //Select the location to crossover with the two parents.
/*10*/   cross = GARandomInt(1,mom.GetSize()-2);

         //Check if bro is nil before he gets the genetic material.
/*11*/   if(c1)
/*12*/   {
/*13*/      nc++; //Increment number of new children.
/*14*/      bro->copy(mom); //Copy mom's genetic material into bro.
            //Insert the selected genetic material from dad into bro.
/*15*/      for(i=cross;i<mom.GetSize();i++)
/*16*/      {
/*17*/         (*bro)[i] = dad[i];
/*18*/      }
/*19*/   }

         //Check if sis is nil before she gets the genetic material.
/*20*/   if(c2)
/*21*/   {
/*22*/      nc++; //Increment number of new children.
/*23*/      sis->copy(dad); //Copy dad's genetic material into sis.
            //Insert the selected genetic material from mom into sis.
/*24*/      for(i=cross;i<mom.GetSize();i++)
/*25*/      {
/*26*/         (*sis)[i] = mom[i];
/*27*/      }
/*28*/   }
         //Return the number of children generated.
/*29*/   return nc;
/*30*/}
```

Figure 6: Example of a Crossover Function

```
/*1*/ class CArrayGenome :
/*2*/    public CArray<int, int>,
/*3*/    public GAGenome
/*4*/ {
/*5*/ public:
/*6*/   GADefineIdentity("CArrayGenome", 201);
        //Declaration of genome operators and evaluator
/*7*/   static void Init(GAGenome&);
/*8*/   static int Mutate(GAGenome&, float);
/*9*/   static float Compare(const GAGenome&, const GAGenome&);
/*10*/  static float Objective(GAGenome&);
/*11*/  static int Cross(const GAGenome&, const GAGenome&,
/*12*/                   GAGenome*, GAGenome*);

         //Constructor that assigns the initializer, mutator,
         //crossover, comparator, and objective functions.
/*13*/   CArrayGenome();

         //The copy constructor.
/*14*/   CArrayGenome(const CArrayGenome& orig) { copy(orig); }

         //The destructor member function.
/*15*/   virtual ~CArrayGenome() {}

         //Definition of assignment operator =.
/*16*/   CArrayGenome& operator=(const GAGenome& orig);

         //The clone function which allocates memory for a new genome
/*17*/   virtual GAGenome* clone(CloneMethod) const;

         //The copy function duplicates the contents of a genome.
/*18*/   virtual void copy(const GAGenome& orig);

         //Declare any other member functions and variables here.
/*19*/};
```

Figure 7: Example of a Custom Genome Class Header

```
/*1*/ CArrayGenome::CArrayGenome() : GAGenome(Init, Mutate, Compare)
/*2*/ {
/*3*/    crossover(Cross);
/*4*/    evaluator(Objective);
/*5*/ }

/*6*/ CArrayGenome& CArrayGenome::operator=(const GAGenome& orig)
/*7*/ {
/*8*/    if(&orig != this) copy(orig);
/*9*/    return *this;
/*10*/}

/*11*/GAGenome* CArrayGenome::clone(CloneMethod) const
/*12*/{
/*13*/   return new CArrayGenome(*this);
/*14*/}

/*15*/void CArrayGenome::copy(const GAGenome& orig)
/*16*/{
/*17*/   GAGenome::copy(orig);

/*18*/   CArrayGenome& new_genome = (CArrayGenome&) orig;
/*19*/   SetSize(new_genome.GetSize());
         //The Copy member of CArray copies the contents of the array.
/*20*/   Copy(new_genome);
/*21*/}
         //Code genome operators, objective function, etc.
```

Figure 8: Example of a Custom Genome Class Implementation

copy the data by the appropriate method for the genome's datatype 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 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 with GAlib. MyGA in Figure 9 inherits from GASteadyGA, but the functions in that class can be overridden to do whatever the programmer desires.

Cellular Automaton Example

In Appendix A is an example of a complete program using GAlib. The GA is used to evolve a one-dimensional cellular automaton (CA) that can determine if an initial condition 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

```
/*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

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 condition.

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 conditions to score the genome with. To ensure that solutions are evolved which work on all initial cases, new initial conditions are generated each generation and the whole population is re-evaluated. The initial conditions 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 conditions used to score the population slowly become more difficult at a specified rate, with more initial conditions 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 function initialize and step are used. Before these two functions are called, however, the GA parameters have been set.

The function "GetIC" was written to generate sets of ICs that increase in difficulty each generation. Within the set of ICs are strings that range in difficulty as well. "RunIC" takes on CA rule set and on IC and applies the rules to the IC. If the CA works correctly and solves the majority problem for that IC, the function returns 1. 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 GAlib. 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 GAs, it frees the programmer to concentrate on the more important aspects of the GA.