GENETIC ALGORITHMS USING GAlib

Richard P. Simpson  Ranette H. Halverson
Department of Computer Science  Department of Computer Science
Midwestern State University  Midwestern State University
Wichita Falls, TX  Wichita Falls, TX
simpson@cs.mwsu.edu  rhalver@nexus.mwsu.edu

ABSTRACT
The recent popularity of genetic algorithms (GA’s) and their application to a wide range of problems is a consequence of their ease of implementation and flexibility. Evolutionary techniques are often applied to optimization and related search problems where the fitness of a possible result is easily established. Problems of this type are generally very difficult if not NP-hard. The ability to find a reasonable solution to these problems within an acceptable time constraint is clearly desirable. Genetic algorithms represent an interesting approach to these complex problems. Since the majority of GA’s are very similar in structure it is possible to build libraries to support construction of GA applications. One such library, GAlib, has been developed at MIT by Matthew Wall. The use of this library in the classroom provides 3 benefits. It allows students to become proficient in the development of GA’s. The use of GAlib provides experience programming with OOP libraries, an important skill. Finally, the use of genetic algorithms provides a variety of opportunities for student research.

INTRODUCTION
The recent popularity of genetic algorithms (GA’s) and their application to a wide range of problems is a consequence of their ease of implementation and flexibility. Evolutionary techniques are often applied to optimization and related search problems where the fitness of a possible result is easily established. Problems of this type are generally very difficult if not NP-Hard. The ability to find a reasonable solution to these problems within an acceptable time constraint is clearly desirable. Genetic algorithms represent an interesting approach to these complex problems. Since the majority of GA’s are very similar in structure it is possible to build libraries to support construction of GA applications. Matthew Wall has developed one such library, GAlib, at MIT. The library is a collection of C++ genetic algorithm components from which it is possible to quickly construct GA’s to attack a wide variety of problems. The use of this library in the classroom allows the instructor to give more interesting problems to the students as well as providing experience using an OOP library developed by someone other than themselves.

This paper provides an overview of the development and use of genetic algorithms. This is followed by a discussion of the features of GAlib and its applications.
to an example problem, the vertex cover problem. Its application in the computer science classroom is then discussed.

BACKGROUND

John Holland developed genetic algorithms (GA’s) in the 1960’s. The original intent was to study biological adaptation and possible ways this could be simulated within computer systems. In his innovative 1975 book *Adaptation in Natural and Artificial Systems* [Holl], Holland introduced population-based algorithms together with the closely associated concepts of chromosomes, crossover operators, and mutation. In this book he made the first attempt to analyze evolutionary computation theoretically by studying the concept of schemas. This original work, together with research by others such as Rechenberg [Rech], resulted in the creation of many different branches of evolutionary computation, which in addition to genetic algorithms includes work in genetic programming [Koza], evolution strategies [Rech], and attempts to justify biological evolutionary theories using computer simulation [Ackl, Hint].

The pioneering work by these researchers as well as many others has resulted in a broad collection of tools and techniques that modern computer professionals can use in addressing some of the hardest problems faced today. Many of these problems have no polynomial time solutions and, consequently, can only be addressed using heuristics, genetic algorithms, and related schemes that generate reasonable but not necessarily optimal solutions. The application of evolutionary computation has been applied in the areas of optimization, automatic programming, machine learning, evolution and learning, population genetics and biochemistry, just to name a few.

Due to the breadth of the area of evolutionary computation, this paper focuses only on genetic algorithms and one of the better known libraries that support the development of GA’s. A short introduction to genetic algorithms is given for clarity, although it would be helpful to read [Meri], which contains a complete example and tutorial. Next the capabilities of GAlib are address and the paper ends with a discussion of an example application of GAlib to the search for optimal vertex covers of a graph.

THE GENETIC ALGORITHM

The basic genetic algorithm involves a population of chromosomes and the operations selection, crossover, and random mutation. The application of genetic algorithms to a problem first involves the representation of solutions to this problem in the form of chromosomes. Each chromosome is a bit string or some other data structure (e.g., tree) that is mapped to a single solution of the problem. A collection of chromosomes, often referred to as a population, defines a temporary set of possible solutions that is continually modified. The quality of a chromosome is referred to as its fitness and it normally is generated by a fitness function. Typically, a GA repeatedly processes the population, creating new generations by applying the operations of selection, crossover, and mutation. The selection operator finds individuals in the population that have high fitness and returns them for crossover. Two fit individuals, called parents, are then combined in a variety of ways to produce offspring. This crossover mechanism is often as simple as swapping sub-strings of bits found in the
parent chromosomes. The last step in this process is called mutation which is usually just a random, low probability, flipping of bits of the offspring chromosome. The main purpose of this operator is to increase the diversity of individuals within a population. Each new population, a generation, will hopefully increase in average fitness over time. At a point determined by a termination criterion, the algorithm terminates and displays the best chromosomes discovered thus far.

To clarify the above concepts let us consider a well-know NP-Hard problem in graph theory known as the optimal vertex cover problem [Gare]. Let $V$ and $E$ represent the vertex and edge sets respectively of a graph. A vertex-cover of an undirected graph $G=(V,E)$ is a subset $V'$ of $V$ such that if $(u,v)$ is in $E$ then either $u$ or $v$ is an element of $V'$. The optimal vertex-cover problem is to find a vertex-cover that is minimal in size. The first problem in the design of a GA for this problem requires the construction of a chromosome scheme that represents solutions. Assuming that the graph in question contains $N$ vertices, an $N$-bit string is used to represent vertex inclusion in the set. In other words, if the $i^{th}$ bit is 1 then node $i$ is a member of the vertex-cover set. The sum of the bits is easily calculated and represents the size of the associated vertex-cover. The vertex sets created by randomly generated chromosomes (N-bit strings) can be easily analyzed to determine if the set is a cover and to determine its size. For this problem and chromosome definition we chose, giving additional weight to the missed edges, the fitness function

$$\text{Fitness} = \#\text{vertices (i.e. 1's in chromosome)} + (\# \text{missed edges}) \times 3.$$  

The missed edges are determined by searching the graph’s adjacency matrix for edges that are not represented in the chromosome. In this case small is better and any chromosome having fitness less than or equal to $N$ is a vertex-cover. The algorithm returns a vertex set with the smallest fitness discovered during its evolutionary search.

The next step in creating a GA for this problem is the design of an appropriate crossover operator. One of the simplest methods, which will work here as well, is called a binary one-point crossover. A random position is selected along the chromosome, cutting it in two sections. Applying this to both parents and swapping the second halves results in two new offspring that are different but genetically related to their parents. In order to guarantee diversity, a mutation operator flips, with low probability, a randomly selected bit of each offspring. Experimentation has shown that a mutation operator probability of .001 often works well in many GA applications.

Combining these steps allows us to derive the following simple genetic algorithm.

**BASIC GA**

Begin with a random population of chromosomes called the *current-population*

Repeat until termination criteria are satisfied

Apply the fitness function to each chromosome in *current-population*

Repeat until $n$ offspring have been created

Select a pair of parent chromosomes based on high fitness

Apply a crossover operator to this pair generating one or more children

Mutate these children with low probability

Add the children to *new-population.*
End-repeat
Set current-population = new-population
End-repeat

The termination criteria can include a number of conditions. For example, the algorithm can stop after \( M \) generations have passed or it may exit when a specific fitness has been discovered.

AN INTRODUCTION TO GAlib

GAlib is a object oriented C++ library created by Matthew Wall of MIT. The software and necessary documentation can be readily downloaded via ftp from ftp://lancet.mit.edu/pub/ga/. GAlib is designed to execute on a large number of UNIX platforms (SGI, Sun, HP, DEC, IBM) as well as Mac and DOS/Windows systems. The source code is available at no cost for non-profit purposes. An extensive manual and associated set of example applications make this library extremely easy to use. In our experience students are able to get the software running with very little help after a short introductory tutorial on the library. In addition, GAlib can be used with PVM (parallel virtual machine) to evolve populations in parallel on a distributed collection of workstations. The system allows users to construct a variety of overlapping and non-overlapping populations, which are all supported by a number of built-in statistics-generating functions. Although chromosomes can be built from any C++ type, there exists a large collection of templated built-in types for immediate use. Examples of these genome types include, but are not limited to:

- GA1DArrayGenome\(<\text{T}>\)
- GA1DBinaryStringGenome
- GAListGenome\(<\text{T}>\)
- GARedGenome
- GASTringGenome
- GATreeGenome\(<\text{T}>\)

The one-dimensional (1D) chromosomes are available in 2D and 3D versions as well. These classes can be the base class of any new derived classes created by the user for special circumstances not met by the built-in types. Associated with each built-in type is a collection of genetic operators such as OnePointCrossover, FlipMutator, UniformInitializer and many others which are dependent on the structure of the chromosome’s base type. Generally, a programmer selects an appropriate chromosome type, the crossover operator, the algorithm style, and the design the objective function. The library heavily supports the first three, but the programmer is basically on his/her own with respect to the design and implementation of the objective function.

THE VERTEX COVER APPLICATION

The previously mentioned vertex cover problem was implemented in GAlib and executed on random graphs. The following graph having twenty nodes and fifty nine
edges was randomly selected for this paper as an example that has a known optimal solution.

\[
\text{Graph}( \{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20\}, \\
\{\{1,10\},\{1,11\}, \{3,4\}, \{3,12\}, \{3,17\}, \{3,20\},\{4,15\},\{4,19\}, \\
\{5,8\},\{6,3\},\{6,7\},\{7,3\},\{7,12\},\{8,5\},\{8,13\},\{9,13\},\{9,15\}, \\
\{9,17\},\{10,20\},\{11,6\},\{11,15\},\{12,9\},\{12,19\},\{13,1\},\{13,8\}, \\
\{13,9\},\{13,12\},\{13,20\},\{14,5\},\{14,11\},\{14,12\},\{14,15\},\{15,8\}, \\
\{15,9\},\{15,10\},\{15,16\},\{15,18\},\{15,19\},\{16,1\},\{16,2\},\{16,7\}, \\
\{16,9\},\{16,12\},\{16,14\},\{16,15\},\{16,19\},\{17,1\},\{17,4\}, \\
\{17,15\},\{18,5\},\{18,7\},\{18,10\},\{19,2\},\{19,8\},\{19,12\},\{20,3\}, \\
\{20,11\},\{20,13\}\});
\]

To determine an optimal solution, the graph was processed using Gaston H. Gonnet’s algorithm \cite{Gonnet}, which uses a combination of a pruned search of the complete tree and a greedy heuristic. According to Gonnet, for random \(n\) node graphs with \(n \ln n\) edges, it runs in time \(O(n^2)\). A lower bound is computed by applying clique theory to a reduced graph. You may find Gaston’s algorithm, which uses the above input format, at \url{http://cbrg.inf.ethz.ch/VertexCover.html}. This algorithm generated an optimal eleven-node vertex set \{3,5,7,10,11,12,13,15,16,17,19\}.

The genetic algorithm designed to solve this problem generated this exact same set after 500 generations each with a population size 100. Multiple runs of the GA using different seeds for the random number generator usually found genomes that were optimal or within one or two nodes of being optimal. This was also true when the size of the graph was increased to 30 and then to 50 vertices. In the 30-node case the GA discovered an optimal cover after six runs with different seeds, although it was different from that returned by the Gonnet program. Gonnet’s algorithm, when fed the large 50-node graph, was able to discover a 38-node cover. It was not able to determine if this graph was optimal and gave a lower bound of 34 for the size of the optimal vertex cover. The GA also discovered 38-node covers after several runs and all of the remaining non-optimal runs were within one to three nodes of 38. Although the above experimentation does not establish the fact that the GA is superior to schemes involving pruning and heuristics, it does indicate that it performs adequately up to reasonably large graphs. Fifty is normally considered quite large when one is processing NP-Hard problems.

Serious students of the Vertex Cover Problem should look at a recent high-speed algorithm that has been developed by R. Balasubramanian and others \cite{Balasubramanian}. According to the authors this algorithm is capable of generating an optimal solution to the above graph \(O(n^2 (1.3247...)^k)\) time, where \(n\) is number of nodes in the graph and \(k\) is the size of the vertex cover.

The GALib implementation for this problem is remarkably easy to develop. Our implementation is a two-page program that was designed, coded, debugged, and executed within an afternoon. The following binary string genome and initializations are done early in the program.

```c
float Objective(GAGenome &); // Name of objective function
int leng = 20; // Number of vertices in graph
int popsize = 100; // number of chromosomes in pop.
```
int ngen = 500; // maximum number of generations
float pmut = 0.001; // probability of mutation
float pcross = 0.7; // probability of crossover
GA1DBinaryStringGenome genome(leng, Objective);

This declares the genome type to GAlib and indicates that Objective is the name of the objective function, which is written by the user. A genetic algorithm object ga is then created using the above define genome by the following statement.

GASimpleGA ga(genome);

The main section of the program is represented by the following short sequence of calls to the ga object.

ga.elitist(gaTrue);// tells the ga to copy the best individual from previous pop to
// to the current population.
// This is the name of the function that defines the stopping criteria of the program.
ga.terminator(GATerminateUponGeneration);
ga.populationSize(popsize);// Defines pop size
ga.nGenerations(ngen);// Defines the max number of generations to run
ga.pMutation(pmut);// Mutation probability
ga.pCrossover(pcross); // Crossover probability
ga.evolve(); // Run the GA!

// Complete the program by printing out the important information
cout << "The GA found:" << ga.statistics().bestIndividual() << "\n";
cout << "Other stats:\n" << ga.statistics();

When applied to the 20-node graph, the vertex cover ga program generated the string (0010101011110111011011), which represents the cover \{3,5,7,10,11,12,13,15,16,17,19\}.

The statement ga.statistics() automatically generates several statistical values, including the number of crossovers since initialization, the number of genome evaluations since initialization, and the maximum and minimum fitness of the initial and final population. These allow the user to determine if the population fitness is increasing over time and consequently converging as expected.

The main function the user writes is the Objective function, which calculates the fitness. The Vertex Cover fitness function as defined for this application is defined as follows.

float Objective(GAGenome& g) {
    GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome & )g;
    float score = 0.0;
    for(int i=0; i<genome.length(); i++){
        score += genome.gene(i); // count number of 1 bits
        for(int j=0; j<genome.length(); j++)
            if((graph[i][j]==1) && (genome.gene(i)==0) && (genome.gene(j)==0))
                score += 3.0; // count the number of bad edges by threes.
    }
    return ((500.0 – score)?500-score:0);
This function is called to evaluate every chromosome. The built-in method
genome.gene(i) returns the \textit{i}th bit of the chromosome. This function calculates and
accumulates the number of one bits plus the number of missed edges multiplied by three.
The multiplier is required to guarantee that all missed edges are removed before node
reduction is attempted. Since the default \textit{ga} assumes that larger fitness values are the
better chromosomes, fitness is inverted by returning the value 500.0–score. The library
does not allow negative fitness values so a zero is returned if the result is negative.

\textbf{USING GALIB IN THE CLASSROOM}

The use of GAlib in a recently offered course in genetic algorithms provided a
variety of ideas for classroom application and for student research. The genetic
algorithms course was taught to a combination of graduate students and junior/seniors. A
variety of projects was assigned to the class. These ranged from simple (generation of a
string of all ones) to quite difficult (generation of magic squares). We found that the
students quickly became proficient and productive with GAlib.

The use of the OOP library, GAlib, in the genetic algorithms class proved to be a
very educational experience for the students. Using such libraries forces students to pay
close attention to parameter definitions, object structure, and return values. Because
many of these libraries do not have tutorial texts written for them, it also forces students
to read a manual. Thus, the authors believe that this is a very good approach for use in an
object oriented programming class or a course in which OOP techniques are emphasized.
Libraries of this type are used heavily in industry where programmers are required to use
MFC, DirectX, and other object oriented API’s.

The area of genetic algorithms opens a wide range of opportunities for student
research projects as well. There are many open problems in this area and are of the
difficulty level that good students can be successful. One student is working on a project
in an attempt to generate new sorting sequences for the age-old shell sort algorithm.
Thus far, some of the sequences generated with the GA have equaled the previously
developed best known sequences. Another student is developing genetic algorithms to
execute under PVM.

\textbf{CONCLUSION}

The use of GAlib in the classroom and the lab environment has proven to be a very
valuable experience in several ways. It provided students with a tool which allowed them
to rapidly enter the world of genetic algorithms and to solve reasonably complex
problems. It gives the students much needed experience using OOP libraries and
provides opportunity for serious students to do useful research.

\textbf{REFERENCES}

\textit{Artificial Life II}, C. G. Lanton, C. Taylor, J. D. Farmer, and S. Rasmussen, eds.,
Addison-Wesley, 1992.


