Comparative Analysis of Genetic Algorithm Implementations

[Experience Report]

Robert Soricone
Northern Arizona University
S. San Francisco St.
Flagstaff, Arizona 86004
rbs2@dana.ucc.nau.edu

Dr. Melvin Neville
Northern Arizona University
S. San Francisco St.
Flagstaff, Arizona 86004
Melvin.Neville@nau.edu

ABSTRACT

Genetic Algorithms provide computational procedures that are modeled on natural genetic system mechanics, whereby a coded solution is "evolved" from a set of potential solutions, known as a population. GAs accomplish this evolutionary process through the use of basic operators, crossover and mutation. Both the representation of the population and the operators require careful scrutiny, and can change dramatically for different classes of problems. Initial tests were conducted using a GA written in Ada95, and required substantial modifications to handle the changing domains. Subsequent testing was done with a toolbox constructed for Matlab, but the class of problems it can solve is restrictive. Ada95's generic mechanism for parameterization would allow for reuse of existing structures for a broader range of problems. This paper describes the tests performed thus far using both approaches, and compares the performance of the two approaches with regards to optimization.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Algorithms, Performance, Design

Keywords

Genetic Algorithms, Parameterized Types

1. INTRODUCTION

The implementation of Genetic Algorithms (GA) typically requires that the parameters and data structures be tailormade for a particular problem domain. Developing a generalized framework for GA would effectively reduce the work involved in writing completely new material by abstracting away the operators in a GA, which would leave the implementor with simply developing an appropriate objective function.

The reuse of both data structures and operators in a GA offers a challenge due to the dramatic difference in how the individual solutions are represented and how they are evaluated. Traditionally, binary strings are used to represent potential solutions to optimization problems, which can be mapped to a real value. The Traveling Salesman Problem, on the other hand, can use permutation encoding to represent a particular Hamiltonian path of a graph.

This report describes the process of converting an original GA, written in Ada95, as well as a comparison of the same process using a pre-made GA kit for use with Matlab. A comparison is made in the amount of work required for such conversions, as well as the performance of the two approaches on a specific optimization problem.

2. GA ELEMENTS

GAs are computational procedures inspired by Darwin's theory of evolution. GAs mimic biological processes that have been studied in natural evolution, namely that of natural selection and survival of the fittest. A problem solved by a GA is said to have its solution evolved by this evolutionary process.

The GA begins with a population, which is a set of possible solutions to a given problem. Each individual solution contained within the population is referred to as a chromosome. Selecting the chromosome encoding scheme for a GA is usually the first and most important step in solving a particular problem, and is greatly dependent on the type of problem at hand. The classical GA representation of a chromosome is a bit string, usually contained in an array or vector. Alternative encoding strategies, such as integer or real values can also be used.

The goal of the GA is to have the initial population reproduce by combining genetic material of two parent chromosomes to form a new unique solution. The primary operators at this stage are crossover and mutation. Again, the implementation of both of these operators depends on the particular encoding scheme, which is dictated by the problem. The simplest form of crossover is single point crossover. With single point crossover, a position is chosen in each of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGAda'04, November 14–18, 2004, Atlanta, Georgia, USA.
Copyright 2004 ACM 1-58113-906-3/04/0011 ...$5.00.

the parent strings. If the chromosome is a binary string, half of one parent's genetic material is combined with the opposite half of the other parent. Permutation encoding would require a slightly different implementation of single point crossover. If the values within the chromosome are to be unique, then simply copying the other half from the second parent may introduce duplicate values, which would render the chromosome useless. This underlies the importance of understanding the problem domain in advance of designing a GA.

The purpose of the mutation operator is to introduce genetic diversity into the population[3]. Mutation works by randomly selecting a position of a randomly selected chromosome, and replacing the value with another value from the same alphabet. Permutation encoding typically employs swapping two random positions within the string, thereby changing the order of the chromosome's permutation. The reproductive cycle will populate a new set of solutions, which will then be ranked in terms of how well each individual performs with respect to the given problem. The new population can be stored in a priority queue.

3. ADA95 GA

A GA was constructed for the purpose of testing three different problems. The code utilizes the generic mechanism of Ada95, which allows for the different forms of parameterization at compile time. Using this approach, specific problems could be added into the framework of the algorithm. One of the "lessons learned" from the tests was that changing the problem required restructuring of certain packages, particularly the ones dealing with crossover and mutation.

3.1 Evolution of Coefficients

The first Ada95 GA involved the evolution of the coefficients of the Taylor power series, which calculates the sine from the value of the angle in radians. Evaluating how well the individuals in the GA performed means determining how close the evolving sine values were to the real sine values. The GA was able to consistently evolve coefficient series that deviated less than 1% from the true sine values for all angles within a specified range[1]. The formula for the power series for the sine is:

$$sin(x) = x^1/1! - x^3/3! + x^5/5! - x^7/7! + ...$$

In our GA the coefficient for each power in the series $a_0x^0 + a_1x^1 + a_2x^2 + ...$ is one of the array cell values *a(0), a(1), a(2)*, ... and is allowed to evolve separately; thus ideally all the even coefficients/cells would evolve to 0.0 while a(5) would evolve to be $1/(5!) = 0.0083333$. The chromosome is the sequence of significant coefficients (out to *a(14)*) for a particular formula variant. Crossover was accomplished by swapping equivalent stretches of a chromosome between two variants: the individuals involved, and the starting and ending coefficients of the segment being swapped were randomly selected[2].

3.2 Maximization of a Function

The second project involved a different class of problem. Instead of evolving parameters of a function, a search was made for an optimal solution for a function within a specified range of [-1..2]. Optimizing the function[2] means finding the maximum value:

$$f(x) = (x * sin(10 * \pi * x) + 1.0)$$

The representation of the chromosome was the classical binary string: an array of bits. Crossover involved the swapping of equivalent sections of the chromosome of bits between two variants. Mutation was accomplished by inverting a randomly selected bit within a chromosome[1].

3.3 Traveling Salesman Problem

The third test was that of the Traveling Salesman problem (TSP). The TSP's goal is to find the shortest possible traversal of a weighted, connected graph, where each node in the graph is visited once and only once, and the tour returns to the starting node. The TSP is NP complete: an exhaustive search of all possible tours is only practical for trivially small graphs (a 225 node graph has 224!/2 possible solutions[1]). The GA was tested on graphs ranging from 225 to 662 nodes, using two variants of the GA. The chromosome in the TSP utilized permutation encoding of integer values, where each value mapped to a specific node on the graph. The "path" of the salesman was the order of the integer values, starting with the first index in the array. Summing the associated weights between the nodes yielded the cost of that particular path. The crossover algorithm is a variant of the OX ("order") approach[5]: The beginning-end sequence of a selected section from one parent is added to the sequence from the other parent of those nodes which are not included in the selection to form the genetic makeup of the offspring.

4. MATLAB EXPERIENCE

As stated in the previous section, it was necessary to rewrite specific packages in the GA to accommodate the different interpretations of the elements and operators for differing problems. This raised the question as to whether or not one could implement a generic GA implementation, and if so, how much was left to the implementor to change. Our attention was turned to two different software packages. The first package was a GA written for Matlab, specifically designed to solve the TSP. The second was a toolbox written for Matlab: The Genetic Algorithm Toolbox from the University of Sheffield's Department of Automatic Control and Systems Engineering.

4.1 TSP Demo

The TSP demo was a free download from Natural Selection, Inc. (www.natural-selection.com). It consists of a series of Matlab files using an Evolutionary Algorithm (EA). The population of chromosomes is held in a matrix, for which Matlab has a number of built-in native functions to create and manipulate. This differs from our Ada95 version of the TSP, in which structures were built from scratch. The wide variety of pre-defined functions in Matlab simplifies the coding and hence the implementation details. The driver file prompts the user for the number of cities to traverse and the size of the population.

The demo uses a permutation-encoding scheme, and the value of a given permutation is found through a mapping to a distance matrix. Examining the Euclidean distance between each of the cities creates the distance matrix. Each generation, the algorithm sorts the population based on the value found on the distance matrix. A similar approach was

Table 1: 225-Node TSP: Ada95 vs. TSP Demo

Distance from Optimal	Ada95 GA	Matlab GA
less than 15%	40	22
less than 10%	40	8
less than 5%	21	1
less than 1%	3	0
Optimal	1	0

used in the previous Ada95 implementation. This restricts our usage of the algorithm to problems that evaluate their fitness in a similar manner. Table 1 summarizes how well both the Ada95 and Natural Selection GA performed when attempting to solve a 225-node graph over a series of twenty runs. The Ada95 version had a unique solution each time the GA was executed. Originally, the Matlab version however found the identical solution each time it was executed. Results would vary when the input parameters were altered (e.g. the population size), but no variance was found within a series of runs using the same parameters. This was attributed to the fashion in which Matlab implements random number generation. Subsequent testing was done after the code was modified to reset the random number generator using the system clock.

4.2 Matlab GA Toolbox

The GA toolbox is comprised of a suite of packaged, sequenced Matlab commands (called m-files), designed to implement the most important functions in a GA[4]. The benefit from using Matlab as a platform is the time saved in writing the actual algorithm due to Matlab's native functions built around matrices. All data structures can be easily built using row vectors, column vectors, and mxn matrices. The various operators are already provided, it is simply a matter of providing an objective function to evaluate the performance of the chromosomes, and a driver for your particular problem. Our objective was then to test the GA toolbox on a set of problems similar to the ones we used with the Ada95 version. This would allow us to determine if the work involved in switching between problem types was comparable to the effort required to change the Ada95 implementation. Also of interest was that of comparing the performance of the two approaches using the same problem and input parameters.

The test problems we chose involved minimizing the one dimensional variants of De Jong's first, De Jong's third, and Rastrigin's test functions. This provided us with the experience of implementing an objective function in the GA toolbox, as well as a challenging problem to compare relative performance of the two approaches. While all three test functions are designed to test optimization methods, Rastrigin's function is particularly difficult for GAs to solve, due to the large number of local minimums.

The chromosomes for each function are represented as binary strings. Single point crossover was used in the recombination stage of the algorithm. This is the default design of the toolkit. Changes in the representation and/or the operators are left to the implementor of the toolkit. The Matlab GA toolkit was originally designed for multi-dimensional optimization, whereby multiple populations are summed. This was beyond the scope for which we wished to explore the GA. Our interests were in finding the optimum value of a single population. Table 2 shows the functional form and global minimum for each test function, where k is the number of dimensions the function is solved for, which in our case was one. In Table 3, the standard deviation for each data set follows in parentheses the average values in both the bouts and time categories.

Table 2: Test Functions for Optimization

Name	Functional Form	Optimal
De Jong #1	$\sum_{k=1,n} x^2$	0
De Jong #3	$\sum_{k=1,n} \lvert x \rvert$	0
Rastrigin	$\sum_{k=1,n} x^2 - 10 * cos(2 * \pi * x)$	-10

Table 3: GA Performance

F(x)	Avg. Bouts(SD)	Avg. Time(SD) sec.
DJ #1:Ada95	3.8(0.95)	0.13(0.00)
DJ #1:Matlab	28.75(9.68)	14.32(0.22)
DJ #3:Ada95	11.9(2.27)	0.11(0.02)
DJ #3:Matlab	28.05(8.9)	14.37(0.15)
RAST:Ada95	8.85(2.68)	0.12(0.04)
RAST:Matlab	13.6(5.12)	11.93(0.14)

The toolbox is "hardwired" for solving optimization problems, specifically, multi-dimensional functions. In order to evaluate a chromosome for a substantially different problem such as the TSP, the m-files relating to crossover and mutation would have to be re-written. While this is possible, it fails to address the issue of supporting multiple problem classes. This is not to say that the toolkit is not providing the functionality it proposes. However, deviating from optimization will require more than simply writing a new objective function. This leaves us with the same extensive coding dilemma we encountered with the Ada95 approach.

The performance results of the two approaches shows the Ada95 implementation finishing in considerably less time than the Matlab version. The primary reason for this can be attributed to manner in which selection is carried out in the two approaches. The Matlab version employs the roulette wheel method, often found in a "classical" GA. Individuals are probabilistically selected based on some measure of performance. An individual has a better chance of passing on its genetic material if the individual has a higher performance value, with respect to the other individuals in the population. The Ada95 GA simply inserts the offspring into a priority queue. A specified number of offspring are selected from the front of the queue, assuring that the best performers survive to pass on their winning combination. While this approach can lead to early convergence, particularly in the case in a function with multiple local extremes, this did not occur with the functions tested.

5. ADA95 GA OPERATORS

The main issue we had with the Matlab GA was the amount of effort required to make it possible for the toolkit to solve alternate classes of problems. This applies to both the representation of the chromosome, and the main operators of the GA. With the previous tests conducted to date on the Ada95 GA, we have already accumulated a much broader range of ways to interpret the representation of the aforementioned components. The structural representation of the chromosome has already included bit strings, floating

point reals, and integer values. Procedures have also been constructed for dealing with each of these representations.

Ada95 has the ability to compose generic packages through the use of package parameters. This method was used previously on a number of different utility packages for the GA. A generic linked list, used to store an initial population, as well as a generic binary heap priority queue, used to "rank" the offspring from each generation. The parameterized packages can be instantiated to handle access to arbitrary data types.

One way to implement generic instantiation is through the use of an appropriate generic template. Another approach to generic instantiation is through use of a separate compilation unit, external to the package that wishes to make use of it[6]. An alternate method would be to implement a formal package. A formal package facilitates the creation of generic packages by permitting one package to be used as a parameter to another package, so that a consistent hierarchical structure of related packages can be generated[2,7].

The application of formal packages is also useful when one wishes to bundle together a number of related types and operations, and treat them as a whole[2,7]. We can make each type in the bundle a parameter of an otherwise null generic package. The net effect of the instantiation of the generic package is an assertion that the individual components have the required relationship and the group can then be referred to though using the instantiated package, also known as a signature[7]. This would allow specific types required for a particular problem class to be assembled as a unique package:

```
generic

type GenObject is private;
type GenObjectPtr is access all GenObject;
type GenMutator is private;
type GenMutatorPtr is access all GenMutator;
type GenCrossover is private;
type GenCrossoverPtr is access all GenCrossover;
type GenAlgs is private;
type GenAlgsPtr is access all GenAlgs;

package GA_Pack is end GA_Pack;
```

We can then compile this two line library unit to construct a new package, and place information about it into the program library. Any other compilation unit can then make use of it by a with clause. A similar approach could be used to create a generic GA, which serves as a driver for the combined packages. The implementor would simply change the generic instantiation to reflect which of the pre-defined chromosomes and operators they wish to use, and recompile.

6. FUTURE ENDEAVORS

Following the same design as the Matlab GA toolbox, a generic GA could be constructed by expanding on the parameterized paradigm already in place for data structures in the existing software. We already have the functionality needed for solving various optimizations, as well as the TSP. We just have to move along the directory hierarchy and perform different instantiations. If the chromosome, crossover, and mutation packages were generic, as the linked list and binary heap currently are, it would be possible to specify which variations one would desire and edit the package instantiation file. A GUI interface could even simplify the selection process, and allow for altering GA parameters. The only new code that would be required is changing the objective function. Once the changes are complete, the source code will be located at the following site: http://astrogeology.usgs.gov/projects/flexible_ga.

7. REFERENCES

[1] Neville M., Soricone R., Waslo J.: On the Automaticity of Genetic Programming, CONIELECOMP 2004, Veracruz, Mexico

[2] Neville, M., Sibley, A.: Developing a Generic Genetic Algorithm, SIGAda '02, Houston, Texas, USA.

[3] Pal, S. and Wang, P.: Genetic Algorithms for Pattern Recognition, CRC 1996

[4] Chipperfield, A., Fleming, P., Polheim, H., Fonseca, C.: Genetic Algorithm Toolbox Users Guide, University of Sheffield, 1996

[5] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, 3rd Ed., Springer 1996

[6] Dale, N., Lilly, S., McCormick, J.: Ada Plus Data Structures An Object Based Approach, Houghton Mifflin College Div., September 1996

[7] Barnes, J.: Programming in Ada95, Addison-Wesely, 1996