Improving Computational Performance of Genetic Algorithms: A Comparison of Techniques

Richard J. Povinelli

Department of Electrical and Computer Engineering
Marquette University, P.O. Box 1881, Milwaukee, WI 53201-1881, USA
e-mail: Richard.Povinelli@Marquette.edu; www: http://povinelli.eece.mu.edu

Abstract

A comparison of three methods for saving previously calculated fitness values across generations of a genetic algorithm is made. These methods lead to significant computational performance improvements. For real world problems, the computational effort spent on evaluating the fitness function far exceeds that of the genetic operators. As the population evolves, diversity usually diminishes. This causes the same chromosomes to be frequently reevaluated. By using appropriate data structures to store the evaluated fitness values of chromosomes, significant performance improvements are realized. Several different data structures are compared and contrasted. This paper shows that, for different sets of genetic algorithm parameters, including selection type, population size, and level of mutation, performance improvements are realized.

1 INSTRODUCTION

Although genetic algorithms (GAs) are robust global optimizers (Goldberg 1989; Holland 1992), they are slower to converge than gradient-based methods (Povinelli and Feng 1999b). Hashing provides an effective method for improving a GA's computational performance (Povinelli and Feng 1999b). In this paper, the study of new techniques to improve GA performance is further investigated. Three methods are evaluated for their effectiveness in improving GA computational performance. The first, for comparison purposes, is the hashing technique previously introduced in (Povinelli and Feng 1999b). The second method saves the current generation's fitness values for use by the following generation. The third uses a binary tree to store previously calculated fitness values.

The paper is broken into four sections. The first section presents the problem statement, discusses the optimization problem, and details the testing platform. The second section discusses each of the three potential solutions. The third section presents the results and discusses their significance. The final section summarizes the paper.

2 PROBLEM STATEMENT

The genesis of this research began by profiling the computation time of a GA. For complex, real world problems, most time is spent evaluating the fitness function (Povinelli and Feng 1999b). By studying the convergence criteria and the diversity characteristics of an evolving GA, it is observed that fitness values are frequently recalculated. This suggests an opportunity for performance improvement.

By efficiently storing fitness values with any of the three methods, GA performance can be dramatically improved. For the test problem, hashing provides the best performance improvement, then the keep last generation algorithm, and finally the binary tree technique.

Previously it was shown that hashing can provide a computational performance improvement of more than 50% on a complex real world problem (Povinelli and Feng 1999b). Feedback from the presentation of these results was the impetus to explore alternative methods for storing previously calculated fitness values.

The first method is the hashing algorithm originally presented in (Povinelli and Feng 1999b). Assuming the size of the hash table is initialized appropriately, the computational cost of hashing is constant, but as the search space is explored, the cost degrades to O(n) for both insertion and retrieval (Manber 1989, p. 80). The hashing technique used here avoids the O(n) cost by reinitializing the hash table as its performance degrades.

The second method investigated is a keep last generation algorithm, which stores the previous generation's fitness values in a hash table, which is reinitialized after each generation.

The third method is a binary tree algorithm, which has an average insertion and retrieval cost of O(log n) and a worst case cost of O(n) (Manber 1989, p. 73). On average the cost of insertion and retrieval grows logarithmically as the search space is explored.

Because of the theoretical comparative computational effort of the three techniques and the amount of previously calculated fitness value storage, it is assumed that hashing provides the greatest benefit. This is experimentally confirmed. But surprisingly, the binary tree algorithm delivers the poorest performance of the three methods.

The test problem for this paper is an application of Time Series Data Mining (TSDM) (Povinelli 1999; Povinelli and Feng 1998; Povinelli and Feng 1999a) to identifying events in a financial time series, where an event is an important occurrence. The TSDM technique characterizes and predicts such events in time series by adapting data mining concepts for analyzing time series. Based soundly in dynamical systems theory (Takens 1980), the TSDM method reveals hidden temporal patterns in time series data by taking advantage of the event nature of many problems.

The search mechanism at the heart of the TSDM method is a GA. The details of the GA are briefly discussed. A simple GA composed of the following steps is used.

While all fitness have not converged
a) Perform selection, save elite individual.
b) Crossover population.
c) Mutate population.
d) Calculate fitness.

The GA uses a binary chromosome of length 18, random locus crossover, and single individual elitism. The stopping criterion for the GA is convergence of all fitness values. Both tournament and roulette selection are investigated. The tournament selection uses a tournament size of two.

The benchmarks were performed with *MATLAB 5.3.1 running under Windows NT 4.0 Service Pack 5*. The computation time was obtained with MATLAB's profiling tool, which reports a precision of .016s. The hardware environment was a dual Pentium III 450MHz with 256MB 100MHz SDRAM, 18GB ultra-IDE hard drive, and a 32MB AGP video card. Although the hardware contains two processors, MATLAB runs on only one processor.

Now that the problem statement has been discussed, the potential solutions will be reviewed.

3 POTENTIAL SOLUTIONS

This section discusses the three algorithms used in this paper. The key to improving the performance of the GA is to reduce the time needed to calculate the fitness. By examining the mechanisms of the GA, it is seen that the diversity of the population decreases as the algorithm runs. The fitness values for the same chromosomes are recalculated repeatedly. If previously calculated fitness values can be efficiently saved, computation time will diminish significantly.

The data mining problem used in this paper searches for temporal patterns in a time series. To find a temporal pattern of length two requires chromosomes of length 18. This means that the search space contains 2^{18} or 267,144 members. With this number of members, the fitness values could be stored in an array. Although this is not an unmanageable size, the problem quickly becomes unwieldy for a slightly larger data-mining problem. For example, a search for temporal patterns of length four requires a chromosome of length 30. This yields a search space with more than one trillion members. With current technology, it is not feasible to store a 10^{12} size array efficiently. This leads us to consider alternative methods for storing the fitness values.

3.1 HASHING ALGORITHM

The classic data structure for efficient storage and retrieval is the hash table. A discussion of hashing can be found in (Manber 1989, pp.78-79). A brief description is provided here.

The interface to a hash table provides two methods. The first is put, which takes two parameters – a key and an element. The put method stores the element with the associated key. The second method is get. It takes one parameter, the key, and returns two values – a flag indicating if an element was found and the element.

Internally, the key-element pairs are stored in an array. The array is accessed through a hash, which is based on the key. Table 1 shows how the data structure is formed.

Table 1 – Sample Hash Table Extract

Hash	Key	Element
100	1001001	32.5
101	Null	Null
110	1100010	45.7

A hash is generated from the key. For this problem, the hash is created by taking the first n bits of the chromosome, where 2^n is the size of the hash table.

As more elements are stored in the hash table, the same hash must be used for representing two distinct elements. This is called a collision. As collisions mount, the efficiency of the hash table degrades from O(1) to O(n). To avoid this performance degradation, a count of the number of collisions is maintained. Once the number of collisions exceeds the size of the hash table, a rehash is performed. Normally, a rehash involves two steps. The first step is to create a larger hash table. The second step is to copy the elements from the smaller hash table to the larger. The hashing algorithm used here is modified because too much time is spent on rehashing. The rehashing mechanism is modified by taking advantage of the diversity reduction that occurs as a GA runs. Only the first step of creating a larger hash table is done. The smaller hash table is destroyed. This seems counter-

intuitive because all of the known fitness values are lost. But since the diversity decreases as the GA runs, many of the eliminated key-element pairs will not be needed. The GA diversity is decreasing, so the chromosome values will be quickly recalculated. The hash table will fill up again with the most used key-element pairs.

3.2 KEEP LAST GENERATION ALGORITHM

The next technique is based on the hash table approach, but only the previous generation of fitness values is stored. Because only the previous generation is stored and the size of GA populations used in this paper are 10 and 30, the hash table size is set to 256. This is a significantly smaller hash table than the one used in the hashing algorithm approach, which starts with a size of 4,096 and grows as large as 65,536.

3.3 BINARY TREE ALGORITHM

The next algorithm is based on a binary tree. A discussion of binary trees can be found in (Manber 1989, pp.71-74). A brief description is provided here. The binary tree algorithm provides the same two methods as the hashing algorithm. The first is put, which takes two parameters – a key and an element. The put method stores the element with the associated key. The second method is get. It takes one parameter, the key, and returns two values – a flag indicating if an element was found and the element.

The main difference between the binary tree algorithm and the hashing algorithm is the data structure used to store the key-element pairs. The hashing algorithm uses an array, whereas the binary tree algorithm uses a binary tree. An example binary tree is shown in Figure 1.

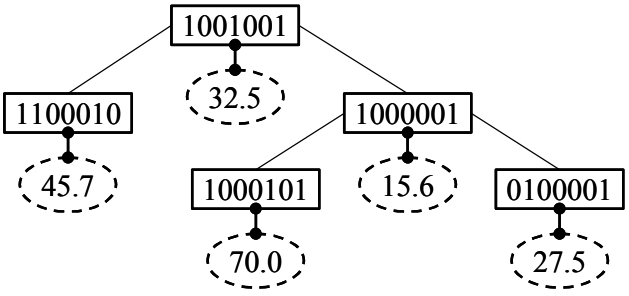

Figure 1 – Sample Binary Tree

The put method starts by setting the current node to be the root of the binary tree. The put method compares the key to current node of the tree. If the key matches the node, the put method stops. If the key is greater than the current node and there is a left branch, the left branch node becomes the current node and the search continues. If there is no left branch, a new node is created as the left branch with a value equal to the key and the element as an attribute. If the key is less than the current node, the right branch node becomes the current node and the search continues. If there is no right branch, a new node is created as the right branch with a value equal to the key and the element as an attribute.

The get method starts by setting the current node to be the root of the binary tree. The get method compares the key to current node of the tree. If the key matches the node, the element stored with that node is returned. If the key is greater the current node, the left branch node becomes the current node and the search continues. If the key is less than the current node, the right branch node becomes the current node and the search continues. If there are no branches off the current node, the get method reports that the search failed.

Now that the three algorithms have been presented, the results of applying these algorithms to the test problem are given.

4 RESULTS

This section discusses results of applying the three algorithms to the test problem. The results show that each of the algorithms provides a performance improvement across population size, mutation, and type of selection. The performance improvements are statistically significant and not artifacts of the number of generations. Each data point in the following tables represents the results of 100 trials.

In Table 2 and Table 3, the mean and standard deviation computational performance measured in seconds is compared for tournament selection. For all combinations of population size (Pop.) and mutation (Mut.), the hashing algorithm is 2.3-3.8 times faster the simple GA. The keep last algorithm is 2.2-3.1 times faster the simple GA. The binary tree algorithm is 1.8-2.6 times faster the simple GA.

Table 2 – Mean Computational Performance for Tournament Selection (s)

Pop.	Mut.	None	Binary Tree	Keep Last	Hashing
10	0	6.35	3.46	2.93	2.73
30	0	39.8	21.5	17.2	16.7
10	0.02	39.4	15.4	12.8	10.3

Table 3 – Standard Deviation Computational Performance for Tournament Selection (s)

Pop.	Mut.	None	Binary Tree	Keep Last	Hashing
10	0	1.49	0.820	0.609	0.497
30	0	9.10	4.81	3.16	3.11
10	0.02	26.5	10.3	7.55	4.55

In Table 4 and Table 5, the probability of a making a Type I Error (incorrectly stating that the computational time is greater when it is not) is shown. The one tail difference of two independent means statistical test is used. In only one case is the statistical significance suspect. That is when the computational time for the keep last algorithm is statistically greater than the computational time for the hashing algorithm when the GA population was 30 and the mutation rate 0.

Table 4 – Probability of Type I Error (α) for Tournament Selection Computational Performance, Part I

Pop.	Mut.	None > Binary Tree	None > Keep Last	None > Hashing
10	0	1.62×10^{-65}	6.33×10^{-101}	3.23×10^{-118}
30	0	1.26×10^{-70}	5.96×10^{-122}	1.27×10^{-127}
10	0.02	1.53×10^{-17}	2.23×10^{-22}	1.53×10^{-27}

Table 5 – Probability of Type I Error (α) for Tournament Selection Computational Performance, Part II

Pop.	Mut.	Binary Tree > Keep Last	Binary Tree > Hashing	Keep Last > Hashing
10	0	1.27×10^{-7}	2.08×10^{-14}	5.91×10^{-3}
30	0	2.35×10^{-14}	1.71×10^{-17}	1.35×10^{-1}
10	0.02	1.98×10^{-2}	3.14×10^{-6}	2.78×10^{-3}

In Table 6 and Table 7, the mean and standard deviation number of generations is presented.

Table 6 – Mean Number of Generations for Tournament Selection

Pop	Mut.	None	Binary Tree	Keep Last	Hashing
10	0	7.72	8.37	8.03	7.83
30	0	15.8	15.8	15.6	16.4
10	0.02	46.0	46.7	40.4	45.8

Table 7 – Standard Deviation Number of Generations for Tournament Selection

Pop	Mut.	None	Binary Tree	Keep Last	Hashing
10	0	1.75	2.27	1.98	1.60
30	0	3.60	4.07	3.49	4.39
10	0.02	30.8	37.2	28.6	30.4

Table 8 and Table 9 show the probability of making a Type I Error in saying that the number of generations is different. Using the two tail difference of two independent means statistical test, none of the number of generations is statistically different from one another. This provides further assurance that the differences in computational performances are statistically significant and not artifacts of the number of generations.

Table 8 – Probability of Type I Error (α) for Tournament Selection Generations, Part I

Pop	Mut.	None ≠ Binary Tree	None ≠ Keep Last	None ≠ Hashing
10	0	0.023	0.240	0.643
30	0	0.985	0.675	0.260
10	0.02	0.890	0.180	0.952

Table 9 – Probability of Type I Error (α) for Tournament Selection Generations, Part II

Pop	Mut.	Binary Tree ≠ Keep Last	Binary Tree ≠ Hashing	Keep Last ≠ Hashing
10	0	0.259	0.052	0.432
30	0	0.682	0.292	0.130
10	0.02	0.180	0.847	0.198

In Table 10 and Table 11, the mean and standard deviation computational performance measured in seconds is compared for roulette selection. As for tournament selection, the order of computation improvement was the hashing algorithm, the keep last algorithm, and the binary tree algorithm. The hashing algorithm is 3.4-4.4 times faster than the simple GA. The keep last algorithm is 2.8-3.8 times faster the simple GA. The binary tree algorithm is 2.3-2.6 times faster the simple GA.

Table 10 – Mean Computational Performance for Roulette Selection (s)

Pop	Mut.	None	Binary Tree	Keep Last	Hashing
10	0	12.7	4.93	3.95	3.73
30	0	89.5	33.8	23.4	20.4
10	0.02	115.	50.3	40.8	28.3

Table 11 – Standard Deviation Computational Performance for Roulette Selection (s)

Pop	Mut.	None	Binary Tree	Keep Last	Hashing
10	0	7.40	1.82	1.32	1.18
30	0	55.4	12.8	7.24	5.89
10	0.02	125.	42.4	36.2	17.6

In Table 12 and Table 13, the probability of a making a Type I Error is shown. Again, in only one case is the statistical significance suspect. That is when the computational time for the keep last algorithm is statistically greater than the computational time for the hashing algorithm when the GA population was 10 and the mutation rate 0.

Table 12 – Probability of Type I Error (α) for Roulette Selection Computational Performance, Part I

Pop	Mut.	None > Binary Tree	None > Keep Last	None > Hashing
10	0	$1.14\text{x}10^{-24}$	$1.56\text{x}10^{-31}$	$2.90\text{x}10^{-33}$
30	0	$5.37\text{x}10^{-23}$	$1.35\text{x}10^{-32}$	$1.11\text{x}10^{-35}$
10	0.02	$5.14\text{x}10^{-7}$	$6.60\text{x}10^{-9}$	$3.93\text{x}10^{-12}$

Table 13 – Probability of Type I Error (α) for Roulette Selection Computational Performance, Part II

Pop	Mut.	Binary Tree > Keep Last	Binary Tree > Hashing	Keep Last > Hashing
10	0	$6.94\text{x}10^{-6}$	$1.54\text{x}10^{-8}$	$1.02\text{x}10^{-1}$
30	0	$9.62\text{x}10^{-13}$	$9.00\text{x}10^{-22}$	$5.33\text{x}10^{-4}$
10	0.02	$4.35\text{x}10^{-2}$	$7.82\text{x}10^{-7}$	$9.58\text{x}10^{-4}$

Table 14 and Table 15 present the mean and standard devation of the number of generations for roulette selection.

Table 14 – Mean Number of Generations for Roulette Selection

Pop	Mut.	None	Binary Tree	Keep Last	Hashing
10	0	15.4	15.4	15.1	15.7
30	0	35.5	33.8	33.8	34.5
10	0.02	135.	155.	135.	146.

Table 15 – Standard Deviation Number of Generations for Roulette Selection

Pop	Mut.	None	Binary Tree	Keep Last	Hashing
10	0	9.05	7.81	8.31	8.61
30	0	21.8	17.7	20.7	20.2
10	0.02	146.	149.	128.	115.

In Table 16 and Table 17, it is seen that number of generations cannot be said to be statistically different between the various algorithms for roulette selection.

Table 16 – Probability of Type I Error (α) for Roulette Selection Generations, Part I

Pop	Mut.	None ≠ Binary Tree	None ≠ Keep Last	None ≠ Hashing
10	0	0.973	0.782	0.817
30	0	0.548	0.566	0.752
10	0.02	0.336	0.997	0.550

Table 17 – Probability of Type I Error (α) for Roulette Selection Generations, Part II

Pop	Mut.	Binary Tree ≠ Keep Last	Binary Tree ≠ Hashing	Keep Last ≠ Hashing
10	0	0.793	0.777	0.599
30	0	0.988	0.780	0.785
10	0.02	0.304	0.633	0.514

This section showed that each of the three algorithms provides statistically significant performance improvement across population size, mutation, and type of selection. It also shows that the hashing algorithm provides the greatest performance improvement. It is between 2.3-4.4 times faster than the simple GA.

5 CONCLUSION

Modification of the simple GA to save previously computed fitness values provides significant performance improvements. This was demonstrated with three different methods for storing the fitness values including a binary tree algorithm, keep last algorithm, and a hashing algorithm. The hashing technique, at 2.3-4.4 times faster than the simple GA, provides the greatest performance improvement.

References

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization, and machine learning*, Addison-Wesley, Reading, Massachusetts.

Holland, J. H. (1992). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*, MIT Press, Cambridge, Massachusetts.

Manber, U. (1989). *Introduction t o Algorithms: A Creative Approach*, Addison-Wesley, Reading, MA.

Povinelli, R. J. (1999). "Time Series Data Mining: Identifying Temporal Patterns for Characterization and Prediction of Time Series Events," Ph.D. Dissertation, Marquette University, Milwaukee.

Povinelli, R. J., and Feng, X. (1998). "Temporal Pattern Identification of Time Series Data using Pattern Wavelets and Genetic Algorithms." *Artificial Neural Networks in Engineering*, St. Louis, Missouri, 691-696.

Povinelli, R. J., and Feng, X. (1999a). "Data Mining of Multiple Nonstationary Time Series." *Artificial Neural Networks in Engineering*, St. Louis, Missouri, 511-516.

Povinelli, R. J., and Feng, X. (1999b). "Improving Genetic Algorithms Performance By Hashing Fitness Values." *Artificial Neural Networks in Engineering*, St. Louis, Missouri, 399-404.

Takens, F. (1980). "Detecting strange attractors in turbulence." *Dynamical Systems and Turbulence*, Warwick, 366-381.