IMPROVING GENETIC ALGORITHMS PERFORMANCE BY HASHING FITNESS VALUES

RICHARD J. POVINELLI AND XIN FENG
Department of Electrical and Computer Engineering
Marquette University, P.O. Box 1881, Milwaukee, WI 53201-1881, USA
E-mail: Richard.Povinelli@marquette.edu
Ph: 414.288.7088
Fx: 414.288.5579

ABSTRACT:
This paper presents a method for improving genetic algorithm (GA) performance. Typically, zero diversity in the population's fitness values signals the stopping point for a GA. As the population evolves, diversity diminishes, causing the same chromosomes to be frequently reevaluated. For real world problems, the computational effort spent on evaluating the fitness function far exceeds that of the genetic operators. By using a hash table to store the most recently evaluated chromosomes, significant performance improvements are realized. Several examples demonstrate the improvements.

Keywords: Genetic Algorithms, Hashing, Data Mining

This paper develops a new way to improve GA performance. Profiling the computation time of a GA reveals that most time is spent evaluating the fitness function. A study of the convergence criteria and diversity characteristics of an evolving GA finds fitness values are frequently recalculated. This suggests an opportunity for performance improvement.

The paper shows that efficiently storing fitness values in a hash table can dramatically improve GA performance. Several examples demonstrate that the method increases in value as the fitness function evaluation cost increases.

PROBLEM STATEMENT

Although GAs are robust global optimizers (Goldberg 1989; Holland 1992), they are slower to converge than gradient-based methods. Table 1 compares the optimization performance of MATLAB's *fmin* (1998, p.7-7) and GA code for three polynomials.

$$10x + 5 \tag{1}$$
$$-x^2 + 10x + 10000 \tag{2}$$
$$x^3 + 150x^2 + x + 100 \tag{3}$$

Both *fmin* and the GA find the maximum for x between –128 and 127. The benchmarks were performed with MATLAB 5.3 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. Since *fmin* falls below the precision of the profiling tool, it was run 100 times for each trial. The computation time was then recomputed for one execution of *fmin*. Each data point in the table represents the mean of 100 trials.

The hardware environment was a dual Pentium II 350MHz with 256MB 100MHz SDRAM, 10.2GB ultra-IDE hard drive, and a 16MB AGP video card. Although the hardware contains two processors, MATLAB was executed on only one processor.

Table 1 - Comparison of *fmin* and GA Performance

	fmin			**GA**		
Polynomial	**Time (s)**	**Flops**	**Optimum**	**Time (s)**	**Flops**	**Optimum**
1	0.023	1065	1,275	4.52	172,901	1,271
2	0.0060	249	10,025	4.60	142,621	10,022
3	0.024	1263	4,467,956	4.06	166,017	4,297,209

As expected, *fmin* outperforms the GA both in computation time and in locating a better solution. It is interesting to note how *fmin* performs much better on the quadratic than on either the first or third degree polynomials. Of course GAs excel in other types of search problems. They have the advantage when the search space is multi-dimensional and contains many local minima. It would be beneficial to find a way to improve the computational performance of a GA. Improved performance allows larger spaces to be searched and more complex problems to be addressed.

ALGORITHM

Now that the goal has been outlined – to find a method to improve the performance of a GA – the details of the implementation will be discussed. The GA provides optimization for a time series data mining problem (Povinelli and Feng 1998). Time series data mining discovers temporal patterns that are characteristic and predictive of events.

The GA is composed of the following steps:

1. Create an elite population.
    a) Randomly generate large population (10 times normal population size).
    b) Calculate fitness.
    c) Select the top one-tenth of the population to continue.
2. While all fitness have not converged
    a) Perform roulette selection, save elite individual.
    b) Crossover population.
    c) Calculate fitness.

The first step, a Monte Carlo search, improves the optimization performance for the data mining problem. The second step is a binary GA with roulette selection, population of 30, random locus crossover, and single individual elitism. The stopping criterion for the GA is convergence of all fitness values.

PERFORMANCE EVALUATION

To find opportunities for performance improvement, the GA is profiled on the data mining problem. Because of the stochastic nature of a GA, 100 trials are run. Table 2 summarizes the results.

Table 2 - GA Computation Profile

	Time (s)		Calls	
Routine	**Mean**	**Std**	**Mean**	**Std**
GA	57.03	17.42	1.0	0.0
CalculateFitness	55.04	16.97	39.5	15.8
RouletteSelection	0.92	0.31	38.5	15.8
Crossover	0.30	0.12	37.5	15.8
Other	0.76	0.04		

The profiling shows that more than 96% of the processing time is spent calculating fitness values. A small fraction (2%) of the processor time is actually used to run the GA. The Calls column of Table 2 shows the number of times each subroutine is called. This means that, on average, it takes 38.5 generations for the algorithm to converge.

SOLUTION

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 can be seen that the diversity of the population decreases as the algorithm runs. The fitness values for the same chromosomes are repeatedly recalculated. If previously calculated fitness values can be efficiently saved, computation time will drop significantly.

The data mining problem used in this paper searches for temporal patterns in a time series (Povinelli and Feng 1998). 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 a manageable 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.

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 three methods. The first is a *create* method which takes as a parameter the size of the underlying data structure to be constructed. The second is *put*, which takes two parameters – the key and an element. The *put* method stores the element with the associated key. The last method is the *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 3 shows how the data structure is formed.

Table 3 - 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. This method creates the hash from the first n bits of the chromosome, where 2^n is the size of the hash table.

The *put* method has three cases. In the first case, the hash table does not contain the key and the hash is not in use. A hash is generated from the key, and the key and element are inserted into the table at the index given by the hash. In the second case, the key is not stored in the hash table, but the hash is already used. This is a collision. A linear probe, which linearly searches for an open location in the hash table to store the key and element, resolves the collision. In the final case, both the key and the hash are already stored in the table. Nothing is done.

The *get* method has four cases. In the first case, the key and its corresponding hash are not in the hash table. The method returns that the key cannot be found. In the second case, the key is not stored in the hash table, but the hash is already used. The *get* method uses a linear probe to discover that the key is not stored in the table. In the third case, the key is associated with its hash. This is the most efficient storage, and the element is quickly returned. In the final case, the key and the hash are in the hash table, but are not associated with each other. This is a collision. The linear probe is used to find the appropriate key and element.

As collisions mount, the efficiency of the hash table degrades to that of a linear search. When the cumulative number of collisions exceeds the size of the hash table, a rehash is performed. Traditionally, a rehash involves two steps. The first step creates a larger hash table – four times larger for this method. The second step copies the elements from the smaller hash table to the larger – a computationally expensive process.

This method does not include the second step. The diversity reduction argument justifies the exclusion of the rehash copy step. Only the first step of creating a larger hash table is done. The smaller hash table is simply thrown out. This seems counter-intuitive because all of the known fitness values are lost. However, since the diversity decreases as the GA runs, many of the eliminated key-element pairs will not be used and the most frequently used chromosome values will be quickly recalculated. The hash table will fill up again with the most used key-element values.

RESULTS

This section presents the results of applying the hash table to the GA. The first two results are for polynomials. The last is the time series data mining problem.

For the polynomial optimization, the random search step is removed. The first experiment maximizes the first order polynomial, $10x + 5$. The second experiment maximizes the second order polynomial, $-x^2 + 10x + 10000$. Both use the range of -128 to 127. Table 4 and Table 5 provide a comparison of the timing profiles for the versions with and without hashing. Each data point is the summary of 100 trials.

The difference of two independent means statistical test is used. The test hypothesis is:

$$H_0: \quad m_{no\ hash} - m_{with\ hash} = 0$$

$$H_A: \quad m_{no\ hash} - m_{with\ hash} \neq 0$$

This test uses a 1% probability of Type I error (a = 0.01), which means when α is less than 0.01 H_0 is rejected.

Table 4 – First Order Polynomial Computation Time Profile (s)

	No Hash		With Hash		
Routine	*m*	*s*	*m*	*s*	*a*
GA	4.52	1.76	5.05	1.96	4.4×10^{-2}
CalculateFitness	2.39	0.93	2.85	1.10	1.4×10^{-3}
RouletteSelection	1.40	0.56	1.44	0.60	6.3×10^{-1}
Crossover	0.55	0.23	0.58	0.24	3.6×10^{-1}
Other	0.18	0.10	0.18	0.05	1.0×10^{-0}

Table 5 – Second Order Polynomial Computation Time Profile (s)

	No Hash		With Hash		
Routine	*m*	*s*	*m*	*s*	*a*
GA	4.60	1.79	5.24	1.63	8.2×10^{-3}
CalculateFitness	2.30	0.91	2.91	0.86	1.1×10^{-6}
RouletteSelection	1.53	0.61	1.55	0.54	8.1×10^{-1}
Crossover	0.59	0.24	0.60	0.22	7.6×10^{-1}
Other	0.18	0.05	0.19	0.05	1.6×10^{-1}

For both experiments, the computation time for *CalculateFitness* is statistically different between the hashing and no hashing versions of the GA. For the second order polynomial, the overall computation time also is statistically different. Hashing has a negative effect on the performance of the GA. The problems are simple and the cost of calculating the fitness is not overwhelming (~50%). Hashing adds no value.

For both experiments, the number of calls to the various routines and the mean optimum are not statistically different between the hashing and no hashing versions of the GA. The introduction of the hashing does not change the GA optimizing characteristics.

The first two experiments suggest that hashing provides no improvement for GA computational performance. For simple problems this is true. But for a simple problem another optimization method, such as MATLAB's *fmin*, would be used. To generalize the impact of incorporating hashing into a GA to more complex problems, the ones at which GAs excel, would be a mistake. The next experiment demonstrates this.

The final experiment is the data mining problem described in (Povinelli and Feng 1998). Table 6 compares the timing profiles. Each data point is the summary of 100 trials.

Table 6 – Data Mining Computation Time Profile (s)

	No Hash		With Hash		
Routine	m	s	m	s	a
GA	52.68	21.55	22.48	3.81	2.6×10^{-43}
CalculateFitness	51.37	21.00	21.10	2.76	3.3×10^{-46}
RouletteSelection	0.86	0.38	0.90	0.80	6.5×10^{-1}
Crossover	0.26	0.16	0.26	0.30	1.0×10^{-0}
Other	0.19	0.04	0.22	0.05	2.8×10^{-6}

With the data mining experiment, the computation time for all the routines but the *RouletteSelection* and *Crossover* are statistically different between the hashing and no-hashing versions of the GA. Here the cost of calculating the fitness is overwhelming (98%). Hashing adds significant value. The computational effort is reduced 57% using hashing. Additionally, the relative computational variation also is reduced by 59%

The number of calls to the various routines cannot be seen as statistically different. The mean optimum for the no-hash version was 0.0410 with a standard deviation of 0.00419 and the mean optimum for the hash version was 0.0415 with a standard deviation of 0.00408. The a equals 0.30, meaning that H_0 cannot be rejected. This shows again that the optimizing character of the GA was not changed by the introduction of the hashing.

This experiment shows the true benefit of hashing. With a complex search problem the computational effort is reduced by over 50%.

CONCLUSION

Changes in a GA's population diversity provide an opportunity for improving its computational performance. Taking advantage of this insight requires an efficient means for storing fitness values. A hash table provides this means. The application of hashing to a GA can improve performance by over 50% and the performance variation is significantly reduced. This method works for complex real-world problems – the ones especially suited to GAs. Care must be taken in extrapolating GA results from simple problems to ones that are more complex.

REFERENCES

(1998). *Using Matlab: Version 5*, The MathWorks, Inc., Natick, MA.

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

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

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

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