An Experimental Comparison of Binary and Floating Point Representations in Genetic Algorithms

Cezary Z. Janikow*
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599, USA

Zbigniew Michalewicz
Department of Computer Science
University of North Carolina
Charlotte, NC 28223, USA

Abstract

Genetic Algorithms (GAs) are innovative search algorithms based on natural phenomena, whose main advantages lie in great robustness and problem independence. So far, GAs were most successful in parameter optimization domains; however, even there certain problems, as lack of fine local tuning capabilities and severe time complexity, prohibit their wider use on most moderately and highly complex problems. Recently, there has been a growing interest in the floating point (FP) representation for genetic algorithms. In this paper we empirically study both FP and binary based GAs using a dynamic control problem — highly complex and quite difficult for any method. Results suggest that the well known advantages of low cardinality alphabets can be compensated for by designing new operators, and that such approach provides means for overcoming some of the mentioned disadvantages.

1 INTRODUCTION

The binary alphabet offers the maximum number of schemata per bit of information of any coding [6] and consequently the bit string representation of solutions has dominated genetic algorithm research. This coding also facilitates theoretical analysis and allows elegant genetic operators. But the 'implicit parallelism' result does not depend on using bit strings [1] and it may be worthwhile to experiment with large alphabets and (possibly) new genetic operators.

*Present address: Department of Mathematics and Computer Science, University of Missouri, St. Louis, Missouri 63121–4499

In [7] the author wrote:

> "The use of real-coded or floating-point genes has a long, if controversial, history in artificial genetic and evolutionary search schemes, and their use as of late seems to be on the rise. This rising usage has been somewhat surprising to researchers familiar with fundamental genetic algorithm (GA) theory ([6], [10]), because simple analyses seem to suggest that enhanced schema processing is obtained by using alphabets of low cardinality, a seemingly direct contradiction of empirical findings that real codings have worked well in a number of practical problems."

The same paper presents a theory of convergence for real-coded GAs that use floating point codings in their chromosomes, and discusses it further:

> "Although the theory helps suggest why many problems have been solved using real-coded GAs, it also suggests that real-coded GAs can be *blocked* from further progress in [some] situations."

However, we argue that some modifications of genetic operators on float point representation may result in a much better performance; these may be also very useful when the problem to be solved involves non-trivial constraints ([12], [13], [14]), and they can help in avoiding such "blocked" situations. Moreover, the search space of the floating point representation is (to a very close degree) equivalent to the problem space. This, in turn, allows a more conscious design of problem specific operators, and actually extends the idea of using a special coding (as Grey) to bring the two spaces together.

Subsequently, we empirically compare a binary implementation with a floating point implementation using various new operators. As a test case we selected a non–trivial, non–decomposable dynamic control problem [15]. The results, due to the limited context of such experiments, should be looked at as a case, rather than generalizable, study; more systematic experimentations must be performed to draw more convincing conclusions.

2 THE TEST CASE

For experiments we have selected the following dynamic control problem:

$$min \left(x_N^2 + \sum_{k=0}^{N-1} (x_k^2 + u_k^2) \right)$$

subject to

$$x_{k+1} = x_k + u_k,\ k = 0, 1, \ldots, N-1,$$

where x_0 is a given initial state, $x_k \in R$ is a state, and $\vec{u} \in R^N$ is the sought control vector. The optimal value can be analytically expressed as

$$J^* = K_0 x_0^2$$

where K_k is the solution of the Riccati equation

$$K_k = 1 + K_{k+1}/(1 + K_{k+1}) \text{ and } K_N = 1$$

During the experiments a chromosome represented a vector of the control states $\vec{u}$. We have also assumed a fixed domain $\langle -200, 200 \rangle$ for each u_i (actual solutions fall withing this range for the class of tests performed). For all subsequent experiments we used $x_0 = 100$ and (unless otherwise stated) $N = 45$. Therefore, a chromosome was represented by a vector $\vec{u} = \langle u_0, \ldots, u_{44} \rangle$, having the optimal value J^* 16180.4.

3 THE TWO IMPLEMENTATIONS

For the study we have selected two genetic algorithm implementations differing only by representation and applicable genetic operators, and equivalent otherwise. Such an approach gave us a better basis for a more direct comparison. Both implementations used the same selective mechanism: stochastic universal sampling [2].

3.1 THE BINARY IMPLEMENTATION

In the binary implementation each element of a chromosome vector was coded using the same number of bits. To facilitate a fast run time decoding, each element occupied its own word (in general it could occupied more than one if the number of bits per element exceeded the word size, but this case is an easy extension) of memory: this way, to read gene's values, elements could be accessed as unsigned integers, which removed the need for binary to decimal decoding (it still required representation range $\rightarrow$ domain scaling). Then, each chromosome was a vector of N words, with N equals the number of elements per chromosome (or again a multiple of such for cases where multiple words were required to represent desired number of bits).

The precision of such an approach depends (for a fixed domain size) on the number of bits actually used, and equals $(UB - LB)/(2^n - 1)$, where UB and LB are domain bounds and n is the number of bits per one element of a chromosome.

3.2 THE FLOATING POINT IMPLEMENTATION

In the floating point (FP) implementation each chromosome vector was coded as a vector of floating point numbers, of the same length as the solution vector. Each element was later initialized within the desired range, and the operators were carefully designed (closed) to preserve this requirement.

The precision of such an approach depends on the underlying machine, but is generally much better than that of the binary representation. Of course, we can always extend the precision of the binary representation by introducing more bits, but this considerably slows down the algorithm (see Section 5). In addition, the FP representation is capable of representing quite large domains (or cases of unknown domains). On the other hand, the binary representation must sacrifice the precision for an increase in domain size, given fixed binary length.

4 THE EXPERIMENTS

The experiments were conducted on a DEC3100 workstation. All presented results represent the average values obtained from 10 independent runs. During all experiments the population size was kept fixed at 60, and the number of iterations was set at 20,000. Unless otherwise stated, the binary representation was using $n = 30$ bits to code one variable (one element of the solution vector), needing $30 \cdot 45 = 1350$ bits for the whole vector.

Because of possible differences in interpretation of different operators, we accepted a probability of chromosomes' update as a fair measure of effort between the floating point and binary representations. Then, all experiments were conducted with individual operators probabilities set to achieve the same such update rates. Accordingly, we could compare runs of both implementations with approximately the same rate of function evaluations. (the number of function evaluations was approximately equal to population size $\times$ update rate $\times$ number of iterations).

4.1 RANDOM MUTATION AND CROSSOVER

In this part of the experiment we ran both implementations with operators which were equivalent (at least for the binary representation) to the traditional ones.

4.1.1 Binary

The binary implementation used traditional operators of mutation and crossover. However, for compatibility with the FP implementation, we allowed crossover points to fall between elements only. The probability of crossover was fixed at 0.25, while the probability of mutation varied to achieve desired rate of chromosome update (shown in Table 1).

Table 1: Relation between Probabilities of Chromosome's Update and Mutation Rate

	Probability of chromosome's update				
	0.6	0.7	0.8	0.9	0.95
Bin	0.00047	0.00068	0.00098	0.0015	0.0021
FP	0.014	0.02	0.03	0.045	0.061

4.1.2 FP

The crossover operators was analogous (and actually equivalent) to that of the binary implementation (split points between float numbers) and applied with the same probability (0.25). The mutation, which we call random, applies to a floating point number rather that to a bit, with an appropriate probability as to achieve the same rates of chromosome's updates (same number of function evaluations) as for the binary case (see Table 1); the result of such a mutation is a random value from the domain $\langle LB, UB\rangle$ with a uniform distribution (special non–uniform distributions will be used in so called "dynamic mutation" introduced in Section 4.2.1).

4.1.3 Results

The results (Table 2) are slightly better for the binary case; however, it is rather difficult to judge them better as all fell quite away from the optimal solution (16180.4). Moreover, an interesting pattern that emerged showed that the FP implementation was more stable, with much lower standard deviation.

Table 2: Average Results as a Function of Probability of Chromosome's Update

	Probability of chromosome's update					std.
	0.6	0.7	0.8	0.9	0.95	dev.
Bin	42179	46102	29290	52769	30573	31212
FP	46594	41806	47454	69624	82371	11275

In addition, it is interesting to note that the above experiment was not quite fair for the FP representation; its random mutation behaves "more" randomly than that of the binary implementation, where changing a random bit (with a uniform distribution) doesn't imply producing a totally random value from the domain. As an illustration let us consider the following question: what is the probability that after mutation an element will fall within δ% of the domain range (400, since the domain is $\langle -200, 200\rangle$) from its old value? The answer is:

FP : Such probability clearly falls in the range $\langle \delta, 2\cdot\delta\rangle$. For example, for $\delta = 0.05$ it is in $\langle 0.05, 0.1\rangle$.

Binary : Here we need to consider the number of low order bits that can be safely changed. Assuming $n = 30$ as an element length and m as the length of permissible change, m must satisfy $m \leq n + \log_2 \delta$. Since m is an integer, then $m = \lfloor n + \log_2 \delta \rfloor$. Again, for $\delta = 0.05$, $m = 25$, and the sought probability is $m/n = 25/30 = 0.833$ — quite a different number.

Therefore, we will try to design a method of compensating for this drawback in the following subsection.

4.2 DYNAMIC MUTATION

In this part of the experiments we ran, in addition to the operators discussed in Section 4.1, a special dynamic mutation operator aimed at both improving a single element tuning and reducing the above disadvantage of random mutation in the FP implementation.

4.2.1 FP

The new operator is defined as follows: if $s_v^t = \langle v_1, \ldots, v_m\rangle$ is a chromosome (t is the generation number) and the element v_k was selected for this mutation, the result is a vector $s_v^{t+1} = \langle v_1, \ldots, v_k', \ldots, v_m \rangle$, where

$$v_k' = \begin{cases} v_k + \triangle(t, UB - v_k) & \text{if a random digit is 0} \\ v_k - \triangle(t, v_k - LB) & \text{if a random digit is 1} \end{cases}$$

The function $\triangle(t,y)$ returns a value in the range $[0,y]$ such that the probability of $\triangle(t,y)$ being close to 0 increases as t increases. This property causes this operator to search the space uniformly initially (when t is small), and very locally at later stages; thus increasing the probability of generating the new number closer to its successor than a random choice. We have used the following function:

$$\triangle(t,y) = y \cdot \left(1 - r^{(1-\frac{t}{T})^b}\right),$$

where r is a uniform random number from [0..1], T is the maximal generation number, and b is a system parameter determining the degree of dependency on iteration number (we used $b = 5$).

4.2.2 Binary

To be more than fair to the binary implementation, we modeled the dynamic operator into its space, even though it was primarily introduced to improve the FP mutation. Here, it is analogous to that of the FP, but with a differently defined v_k':

$$v_k' = mutate(v_k, \nabla(t,n)),$$

where $n = 30$ is the number of bits per one element of a chromosome; $mutate(v_k, pos)$ means: mutate value of the k-th element on pos bit (0 bit is the least significant), and

$$\nabla(t,n) = \begin{cases} \lfloor \Delta(t,n) \rfloor & \text{if a random digit is 0} \\ \lceil \Delta(t,n) \rceil & \text{if a random digit is 1} \end{cases}$$

with the b parameter of Δ adjusted appropriately if similar nonuniformity is desired (for examples see Figure 1).

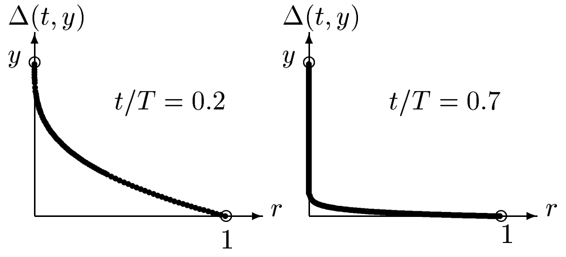

Figure 1: Δ function for two selected times and $b = 4$

4.2.3 Results

We repeated similar experiments to those of section 4.1.3 using also the dynamic mutations applied at the same rate as the previously defined mutations.

Table 3: Average Results as a Function of Probability of Chromosome's Update

	Probability of chromosome's update			std.
	0.7	0.8	0.9	dev.
Bin	32275	35265	30373	40256
FP	21098	20561	26164	2133

Now the FP implementation shows a better average performance (Table 3). In addition, again the binary's results were more unstable. However, it is interesting to note here that despite its high average, the binary implementation produced the two single best results for this round (16205 and 16189).

4.3 OTHER OPERATORS

In this part of the experiment we decided to implement and use some additional operators — those easy to implement in each space. Therefore, the purpose of this part of the experiment was not to compare both implementations in exactly the same context, but rather to see what level of quality could be obtained by using a set of easily implementable operators. Actually, this is where we start to distinguish between problem–independent and problem–dependent operators, to show that problem–specific operators are superior.

4.3.1 Binary

In the binary representation, the space is that of binary strings. This provides for the highly acclaimed operator problem–independence, since all operators can be defined in this space regardless of the underlying problem space. In addition to those previously described operators, we implemented a multi–point crossover, and also allowed for the classical crossovers (crossover points within bits of an element). The multi–point operator was introduced to set aside the single vs. muilt–point crossover debate; the probability of a crossover split was controlled by a system parameter (set at 0.3).

4.3.2 FP

In the floating point representation we deal directly (disregarding finite precision) with the problem space. Therefore, we can easily define new operators acting on real space vectors rather that some artificial agents. Accordingly, in addition to those previously described operators, we also implemented an analogous multi–point crossover, and single and multi–point arithmetical crossovers; They average values of two corresponding elements (rather that exchange them), at selected points. Such operators have the property that each element of the new chromosomes is still within the original domain. A version of such an arithmetical crossover averages two whole chromosomes along all dimensions, and simulates finding a midpoint between two points of a real space. For more details on these operators an interested reader is referred to [11], [13], [12], [18].

4.3.3 Results

Table 4: Average Results as a Function of Probability of Chromosome's Update

	Probability of chromosome's update			std. dev.	Best
	0.7	0.8	0.9		
Bin	23814	19234	27456	6078	16188.2
FP	16248	16798	16198	54	16182.1

Here the FP implementation shows an outstanding superiority (Table 4); Even though the best results are not so much different, only the FP was consistent in achieving that.

5 TIME PERFORMANCE

Many complain about the high time complexity of GAs on nontrivial problems. In this section we compare the time performance of both implementations using the mutation and crossover as defined in Section 4.1.

Table 5: CPU Time (sec) as a Function of Number of Elements

	Number of elements (N)				
	5	15	25	35	45
Bin	1080	3123	5137	7177	9221
FP	184	398	611	823	1072

Table 5 compares CPU time for both implementations on varying number of elements in the chromosome. The FP version is much faster, even for the moderate 30 bits per variable in the binary implementation; Both times are linear in the chromosome's length. Since we executed approximately the same number of function evaluations, and there was no need for binary decoding other than domain scaling (see Section 3.1), the major factor for these differences had to be the operator selection mechanisms. Furthermore, since the crossover operators were, in fact, identical, the mutation mechanisms had to contribute mostly. Actually, the reasons for such time disparities in these mechanisms are easily visible from Table 1: while seeking applicable mutation antities, the binary implementation iterates over all bits of a chromosome, but the floating point implementation iterates only over all elements of a chromosome. In other words, if other operations of the implementations were neglected, one would expect the binary one to run slower by the factor equal to the number of bits required to represent one gene (30 in the above run). The above fact was confirmed by measuring time spent in various parts of each algorithm. Note that the above holds only for our definitions of floating representation operators; if one wished to model these operators on the classical ones, the outcome might be quite different.

From the above discussion follows that the time disparity between binary and floating point implementations is directly proportional to the number of bits per one gene of the former. This implies that for problems which, due to some problem specific goals, require high precision, and, therefore, a longer bitwise gene representation, the difference should increase. This claim is exemplified in Table 6.

Table 6: CPU Time (sec) as a Function of Number of Bits Per Element; $N = 45$

	Number of bits per binary element					
	5	10	20	30	40	50
Bin	4426	5355	7438	9219	10981	12734
FP	1072 (constant)					

6 SUMMARY

The results of the conducted experiments indicate that the floating point representation is faster, more consistent from run to run, and provides higher precision (especially with large domains where binary coding would require prohibitively long representation). At the same time its performance can be enhanced by special operators to achieve high (even higher than that of the binary representation) performance accuracy. The design of such operators is, however, easy in the representation space approximately equivalent to the problem space. This approach abandons the idea of problem–independent operators; however, the floating point representation was introduced especially to deal with real parameter problems and we see no drawbacks of tailoring the operators to such domains.

These results support other studies praising the floating point representation, *e.g.* [7] gives the following reasons for such a preference: (1) comfort with one-gene-one-variable correspondence, (2) avoidance of Hamming clifs and other artifacts of mutation operating on bit strings treated as unsigned binary integers, (3) fewer generations to population conformity.

Acknowledgments: This research was supported by a grant from the North Carolina Supercomputing Center.

References

[1] Antonisse, J., *A New Interpretation of Schema Notation that Overturns the Binary Encoding Constraint*, in [17], pp. 86–91.

[2] Baker, J.E., *Reducing Bias and Inefficiency in the Selection Algorithm*, in [9].

[3] Bosworth, J., Foo, N., and Zeigler, B.P., *Comparison of Genetic Algorithms with Conjugate Gradient Methods*, Washington, DC, NASA (CR–2093), 1972.

[4] Davis, L., (Editor), *Genetic Algorithms and Simulated Annealing*, Pitman, London, 1987.

[5] De Jong, K.A., *Genetic Algorithms: A 10 Year Perspective*, in [8], pp.169–177.

[6] Goldberg, D.E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison Wesley, 1989.

[7] Goldberg, D.E., *Real-coded Genetic Algorithms, Virtual Alphabets, and Blocking*, University of Illinois at Urbana-Champaign, Technical Report No. 90001, September 1990.

[8] Grefenstette, J.J., (Editor), Proceedings of the First International Conference on Genetic Algorithms, Pittsburg, July 24–26, Lawrence Erlbaum Associates, Publishers, 1985.

[9] Grefenstette, J.J., (Editor), Proceedings of the Second International Conference on Genetic Algorithms, MIT, Cambridge, July 28–31, Lawrence Erlbaum Associates, Publishers, 1987.

[10] Holland, J., *Adaptation in Natural and Artificial Systems*, Ann Arbor: University of Michigan Press, 1975.

[11] Janikow, C., and Michalewicz, Z., *Specialized Genetic Algorithms for Numerical Optimization Problems*, Proceedings of the International Conference on Tools for AI, Washington, November 6–9, pp.798–804, 1990.

[12] Michalewicz, Z. and Janikow, C., *GENOCOP: A Genetic Algorithms for Numerical Optimization Problems with Linear Constraints*, to appear in Communications of ACM, 1991.

[13] Michalewicz, Z. and Janikow, C., *Genetic Algorithms for Numerical Optimization*, Statistics and Computing, Vol.1, No.1, 1991.

[14] Michalewicz, Z. and Janikow, C., *Handling Constraints in Genetic Algorithms*, Proceedings of the 4th International Conference on Genetic Algorithms, San Diego, July 13–16, 1991.

[15] Michalewicz, Z., Krawczyk, J., Kazemi, M., Janikow, C., *Genetic Algorithms and Optimal Control Problems*, Proceedings of the 29th IEEE Conference on Decision and Control, Honolulu, pp.1664–1666, December 5–7, 1990.

[16] Schaffer, J., Caruana, R., Eshelman, L., and Das, R., *A Study of Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization*, in [17], pp.51–60.

[17] Schaffer, J., (Editor), Proceedings of the Third International Conference on Genetic Algorithms, George Mason University, June 4–7, 1989, Morgan Kaufmann Publishers, 1989.

[18] Vignaux, G.A. and Michalewicz, Z., *A Genetic Algorithm for the Linear Transportation Problem*, IEEE Transactions on Systems, Man, and Cybernetics, Vol.21, No.2, 1991.