# Testing the Robustness of the Genetic Algorithm on the Floating Building Block Representation.

**Robert K. Lindsay**
Mental Health Research Institute
University of Michigan
Ann Arbor, MI 48109-0720
lindsay@umich.edu

**Annie S. Wu**
Artificial Intelligence Laboratory
University of Michigan
Ann Arbor, MI 48109-2110
aswu@eecs.umich.edu

## Abstract

Recent studies on a floating building block representation for the genetic algorithm (GA) suggest that there are many advantages to using the floating representation. This paper investigates the behavior of the GA on floating representation problems in response to three different types of pressures: (1) a reduction in the amount of genetic material available to the GA during the problem solving process, (2) functions which have negative-valued building blocks, and (3) randomizing non-coding segments. Results indicate that the GA's performance on floating representation problems is very robust. Significant reductions in genetic material (genome length) may be made with relatively small decrease in performance. The GA can effectively solve problems with negative building blocks. Randomizing non-coding segments appears to improve rather than harm GA performance.

## Introduction

Intelligent natural systems are complexly structured, modular *organisms*. Acting over billions of years, evolution has created organisms by combining simple components into more complex systems through selection from a diverse population of varied individuals. The Genetic Algorithm (GA) was pioneered by Holland (Holland 1975) as an abstract model of evolution based upon only a few of its then-known salient features. In the standard GA representation, individuals are represented as binary strings, usually a few hundred digits in length, recombination is limited to *crossover* (swapping of long segments between pairs of individuals) and *mutation* (random changes of single bits), adaptation is modeled by a real-number valued fitness function that (typically, but not necessarily) treats substrings of the individual as parameters in an algebraic formula, and higher fitness results in a higher *probability* of reproduction. The bulk of research on the GA has been directed toward studying its ability to find function optima, a problem of interest to AI. However, a far more fundamental AI problem is how ever more complex organisms can evolve, whether or not they are optimizers.

In nature, the mechanisms are more complex than in the GA: there is a distinction between individuals that carry the genetic code (genotypes) and their subsequent development as organisms (phenotypes) that interact with the environment and each other; genotypes are strings of base-pairs, each of which can assume one of four values, and may be several billion in length; the range of genotype length among different species is very large; genotype length is not strictly correlated with organism complexity; there are several restructuring mechanisms, some of which are still not well-understood; particular patterns of base-pair strings (genes) yield particular products, usually proteins; in the more complex organisms (eukaryotes) most of the genotype – strangely – does not code for any product (*non-coding segments*); the genes may lie anywhere on the genotype, since the product is determined solely by the base-pair pattern (*location independence*); genes that are functionally related are often but not always nearby, and though at no fixed distance are in relatively similar order in same-species individuals; genes may contain short base-pair sequences having no known purpose (introns) that are removed during the translation process; the genotype may comprise several disjoint strings (chromosomes); the genes and gene products interact in complex ways including switching the expression of one another on and off; phenotypes develop over extended periods of time through complex interactions of gene products; success means that an individual survives long enough to reproduce many times, which may take anywhere from a few seconds to more than a decade depending on the species; successful species persist over millions of years even after more complex species have evolved.

Although gene products interact in complex ways, they are inherently modular because biochemical reactions require only the presence of proteins in suf-

ficient concentrations rather than in precise arrangements such as are required in typical mechanical systems or computer programs. Presumably, individual genes whose products serve useful functions in a variety of contexts act as *building blocks*. Building blocks may form larger building blocks when they become copresent in single individuals thus producing a combination of components that is more fit than any taken individually. Hence over many generations a functional hierarchy is formed, and the components are preserved because in combination they are highly fit and hence more successful, on average, at reproduction. In both nature and the GA, adjacency of base-pairs or bits is an important carrier of information, which is why crossover, which preserves most adjacency information, is fundamentally different from extensive mutation, which does not. The Building Block Hypothesis (BBH) of the GA states, in effect, that crossover is a mechanism that finds successful new combinations more rapidly than does random change because crossover takes advantage of the ordered structure of the fit fragments. The BBH is the central working hypothesis and motivation for GA research.

Some form of the BBH is essential to the success of evolution, which would otherwise amount to random hill-climbing and be unlikely to produce complexly structured stable organisms exhibiting intelligence even in geologic time. Recent research has cast doubt on the general validity of the BBH in the standard GA representation (Wu 1995) (Wu & Lindsay 1996). What else is needed in an evolutionary model to result in the beneficial effects predicted by the BBH? We have explored this question by modifying the representation used by the GA to reflect two of the above mentioned features of natural evolution that may plausibly contribute to this: location independent building blocks and non-coding segments. We argue that the combination of these two features may lead to GA behavior that better maintains diversity, is robust in changing environments, better preserves building blocks that have been discovered, and increases the rate of successful recombination that produces structured organisms.

Initial studies on this new *floating building block representation* have produced favorable results (Wu 1995) (Wu & Lindsay 1996). In this article, we investigate the effectiveness of the floating representation by studying its reaction to three, potentially difficult, situations: (1) a reduction in the amount of genetic material available to the GA during a run, (2) the existence of negative building blocks in the function to be solved, and (3) randomized non-coding segments.

## Royal Road Functions

The RR functions are a class of functions that were designed for studying building block interactions (Mitchell, Forrest, & Holland 1991). Building blocks in the GA are groups of bits that affect the fitness of an individual. The RR functions have multiple levels of hierarchically arranged building blocks. All building blocks are defined and assigned optimum patterns and fitness contributions before a run begins. The optimum individual is also defined in advance. The basic or lowest-level building blocks are shortest in length and upper-level building blocks consist of combinations of lower-level building blocks. The fitness of an individual is the sum of the fitness contributions of all building blocks that exist on that individual. Table 1 shows an example of an RR function. This RR function has eight basic building blocks, fifteen total building blocks, and four levels. The additional fitness support provided by the intermediate-level building blocks are expected to encourage the recombination and preservation of the basic building blocks, in effect, laying out a "royal road" for the GA to follow to the optimum individual. We chose the RR functions for our investigations because the pre-defined, hierarchical structure of an RR function allows us to carefully monitor the GA's progress in solving the function and also allows for easy modification of the characteristics of the function. For a detailed description of the GA used in these experiments the reader is referred to (Wu 1995).

## Floating Building Block Representation

The floating building block representation introduced in (Wu 1995) (Wu & Lindsay 1996) investigates the idea of location independent building blocks for the GA. Instead of defining building blocks by their location on an individual, the existence and placement of building blocks depend on the existence and placement of predefined tags. Table 2 shows an example of a floating Royal Road (FRR) function. Like the function shown in table 1, this function has eight basic building blocks, fifteen total building blocks, and four levels. In the FRR function, however, basic building blocks may appear anywhere on an individual. Higher level building blocks still consist of combinations of lower level building blocks. At the start of a run, all building blocks are assigned fitness contributions; each basic building block is assigned a unique optimum pattern. Optimum patterns are randomly generated at the start of each run, so building blocks have different optimum patterns from run to run. Everywhere the start tag is found on an individual, a possible building block exists immediately following this tag. For example, suppose we use the function shown in figure 2. The follow-

| $i$ | Schema $b_i$ | Level $l_i$ |
|---|---|---|
| 0 | 11111111************************************************************ | 0 |
| 1 | ********11111111******************************************************** | 0 |
| 2 | ****************11111111************************************************ | 0 |
| 3 | ************************11111111**************************************** | 0 |
| 4 | ********************************11111111********************************** | 0 |
| 5 | ****************************************11111111**************************** | 0 |
| 6 | ************************************************11111111****************** | 0 |
| 7 | ********************************************************11111111********** | 0 |
| 8 | 1111111111111111************************************************************ | 1 |
| 9 | ****************1111111111111111******************************************** | 1 |
| 10 | ********************************1111111111111111************************** | 1 |
| 11 | ************************************************1111111111111111********** | 1 |
| 12 | 11111111111111111111111111111111************************************************ | 2 |
| 13 | ********************************11111111111111111111111111111111********** | 2 |
| 14 | 11111111111111111111111111111111111111111111111111111111111111111111111111111111 | 3 |

Table 1: The RR function, `8x8.RR2`. Each $b_i, i = 1, 2, ..., 15$ shows the optimum pattern for one building block. *'s represent *don't care* bits. For each $b_i$, there is a value $l_i$ which indicates the level of that building block.

| Basic building blocks (Level 0, start tag:11) | | Upper level building blocks | | |
|---|---|---|---|---|
| BB | Opt. pattern[1] | BB | Level | Component BB |
| 0 | 01101111 | 8 | 1 | 0 1 |
| 1 | 10100110 | 9 | 1 | 2 3 |
| 2 | 00001001 | 10 | 1 | 4 5 |
| 3 | 10010111 | 11 | 1 | 6 7 |
| 4 | 11100010 | 12 | 2 | 0 1 2 3 |
| 5 | 10010111 | 13 | 2 | 4 5 6 7 |
| 6 | 00101010 | 14 | 3 | 0 1 2 3 4 5 6 7 |
| 7 | 11111110 | | | |

Table 2: The FRR function `8x8.FRR2`. [1]Randomly generated at the start of each run.

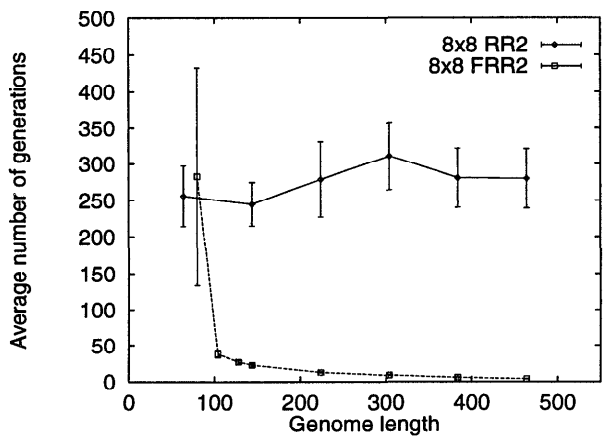


Figure 1: Average number of function evaluations needed to find the optimum vs. genome length: GA on `8x8.RR2` and `8x8.FRR2`.

ing individual has eleven start tags indicating possible building blocks, but only three actual building blocks: 2 copies of building block 0 and 1 copy of building block 1. Start tags of possible building blocks are marked by an `x` while the start of existing building blocks are marked by the corresponding building block numbers.

```
10001101101111010011001101101111 00
    x  x  xxx      x   x  x  xxx
    0     1        0
```

Like the RR functions, FRR building blocks contribute to the fitness of an individual only if their optimum pattern exists on an individual. The goal of the GA, then, is to find at least one copy of each building block on an individual. Note that with the floating representation comes many new issues such as non-coding segments (segments of bits that have no effect on the fitness of the individual), multiple copies of building blocks, and overlapping building blocks. These issues are discussed in greater detail in (Wu 1995).

Figure 1 shows typical results from a comparison of GA performance on the fixed and floating representations. The $x$-axis shows increasing genome length (length of an individual) and the $y$-axis shows increasing generations. In the fixed representation functions, extra bits from the longer genome lengths are distributed evenly between the basic building blocks as non-coding segments. With the floating representation, longer genome lengths simply allowed the GA more resources with which to work. Comparisons of GA performance on the fixed and floating representations clearly indicate that long genome lengths are a big advantage for the floating representation. At short genome lengths, the average GA performance on the FRR functions is worse than the average GA performance on the RR functions. As the genome length

increases, GA performance on the floating representation improves exponentially and quickly becomes significantly better than GA performance on the fixed representation. Once the genome length exceeds a certain point, the performance levels off to ideal values.

## Density of Genetic Material

Given the behavior shown in figure 1, why not simply run the GA on floating functions using as long a genome length as possible? The amount of "genetic material" used in each generation is not a trivial issue. The more genetic material there is to be analyzed, the more computation time is required. As a result, there is a tradeoff between the better performance of the floating representation on longer individuals and the extra computation time required to generate and process the longer individuals. The minimum genome length that a fixed RR function needs to code a solution is equal to the sum of the lengths of the basic building blocks. The minimum genome length for an FRR function depends on the arrangement of building blocks with the greatest overlap. This arrangement differs for each set of building block optima. Because of the ability to overlap building blocks, FRR runs on average use fewer bits to code for all of the building blocks of a function than corresponding RR runs. What then is the minimum genome length necessary for the GA to be able to find an optimum individual for an FRR function?

We attempt to empirically estimate this minimum length. A series of experiments was carried out on the FRR function, `8x6.FRR2`, which resembles the one described in table 2 except that basic building blocks are six instead of eight bits long. The longest genome length tested was 64 bits, and genome lengths were shortened by four bits in each consecutive experiment until the GA stopped finding optimum individuals. Each experiment was run 100 times and the average results reported in figure 2. The $x$-axis shows increasing genome length. The $y$-axis values may be interpreted in one of three ways. The percentage of successful runs refers to the number of runs (out of a total 100) in which an optimum individual was found. The average generation found is calculated from only those runs where an optimum individual was found. The percentage of coding bits, also calculated only from runs where an optimum was found, shows the average proportion of an optimum individual that was used to code for the solution. As genome length decreased, GA performance grew progressively worse. The GA was less successful at finding optimum individuals, required more time to find the optimum in those runs where it was successful, and used up a larger percentage of the bits

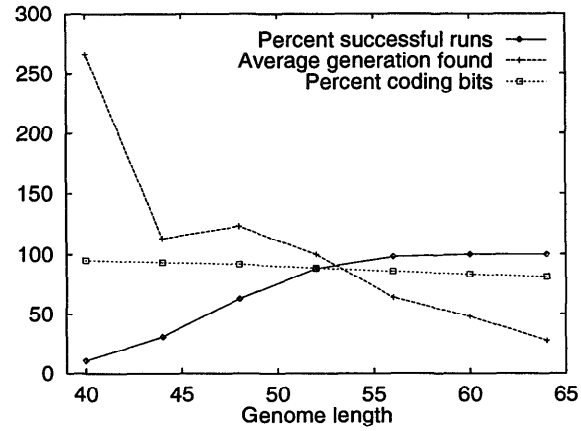


Figure 2: Results from shrinking the genome length of the `8x6.FRR2` function.

of an individual to code for the optimum. The "percent coding bits" graph shows a tendency for the GA to use as much space as is available. As genome length shrank, the GA packed building blocks more compactly until the percentage of coding bits approached 100%. At this point, the GA had compacted the building blocks as tightly as possible and was unable to find optimum individuals at any shorter genome lengths. Studies on other FRR functions produced similar results.

## Negative Building Blocks

A key phrase associated with evolutionary systems is "survival of the fittest." Positive qualities are rewarded with increased chance of survival and reproduction, and negative qualities are discouraged and eliminated. The complex mechanisms of natural systems, however, have resulted in gene products that may interact and affect their organism in many ways, possibly producing both positive and negative effects. This overlap in functionality is paralleled in the FRR function by overlapping building blocks. As in natural systems, one segment of an individual can affect the fitness of the individual in multiple ways.

In using negative building blocks in the FRR functions, we introduce a situation where building blocks may contribute both positively and negatively to an individual's fitness. *Negative building blocks* have negative fitness contributions that decrease an individual's fitness. However, a negative building block may be part of a larger, positive building block. This situation is similar to the *deceptive* problems described in (Goldberg 1989) where building blocks that are discouraged/encouraged early in a run may/may not lead to the optimum individual. Initial experiments showed

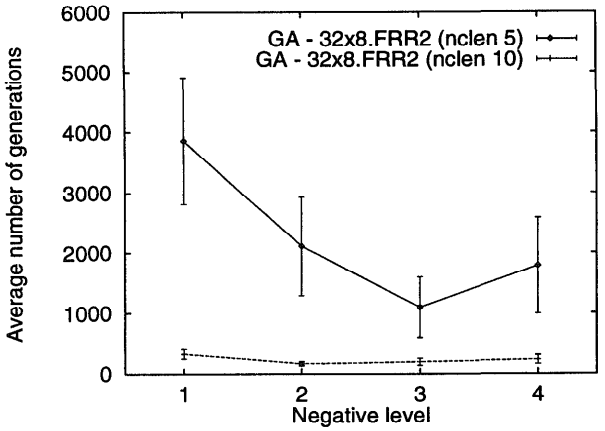


Figure 3: Number of function evaluations needed to find the optimum vs. level of negative building blocks: `32x8.FRR2`.

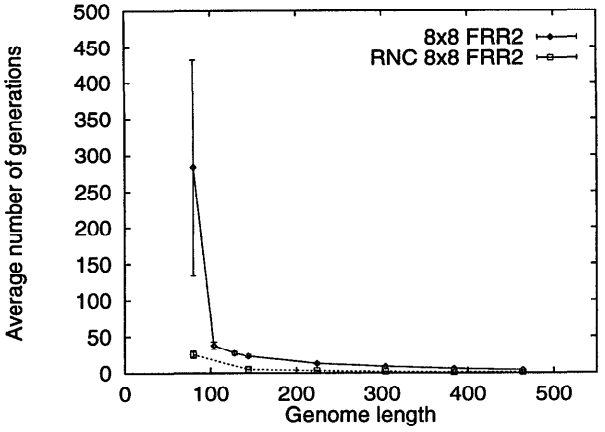


Figure 4: Average number of generations needed to find the optimum vs. genome length: GA and RNC-GA on `8x8.FRR2`.

that one or two negative building blocks had little effect on GA performance. Our next test, then, was to investigate the effects of setting entire intermediate levels of building blocks to negative values.

For each function tested, the intermediate level building blocks were assigned a fitness of -1 (while all other building blocks had fitness 1) one level at a time. The performance of the GA was evaluated to determine at which levels of a function is negative reinforcement most detrimental. Each experiment was run 50 times and the average values reported here.

Figure 3 plots the data from a six level function called `32x8.FRR2` using genome lengths of 416 and 576. The $x$-axis indicates the level at which building blocks were set to negative fitnesses and the $y$-axis indicates the generation at which the optimum was found. Setting the lower level building blocks to negative fitnesses seemed to cause the most difficulty for the GA; more time was needed to find an optimum. This behavior supports the building block hypothesis' implication that it is important to encourage the GA to retain short building blocks because they are easy to find and can be easily combined to form larger building blocks. Performance also degrades when upper level building blocks are set to negative fitnesses. Because of the hierarchical nature of the RR functions, a tendency to lose very large building blocks can result in significant setbacks in the search for solutions. This suggests that it is also important to encourage the GA to keep upper level building blocks in the population. The best performance occurred when the middle-most levels were set to negative fitnesses. Tests on other RR and FRR functions produced similar results. The need for positive reinforcement at the lower levels is even more important with the RR functions.

## Randomized Non-coding Segments

One of the unique features of the floating representation is the fact that floating building blocks allow the bits of an individual to switch back and forth between coding and non-coding status. As mentioned earlier, non-coding bits are bits that are completely ignored by the fitness function and make no contribution to an individual's fitness. Thus, when an existing building block is destroyed, parts of that building block may still be retained as non-coding regions. Creating a building block – whether one that was just destroyed, or a different one – out of these parts should be easier than creating a new building block from scratch. In addition, there are some situations in which the fitness function may vary over time. The ability to save old building blocks in the population could reduce search time later on in a run when the old building blocks once again become "fitness contributing" building blocks. Thus the floating representation is thought to provide the GA with a limited "memory" capability that could improve its rediscovery abilities.

To test this hypothesis, an additional step was added to the GA in between the evaluation and selection steps to "randomly initialize" the non-coding bits of each individual. This step should erase any partial building blocks that may have been saved in non-coding regions. Coding regions were left untouched.

Figure 4 plots the average number of generations needed to find an optimum individual as genome length increases for the `8x8.FRR2` function. "RNC" or randomized non-coding segments refers to the modified GA runs. Figure 5 plots the average number of times

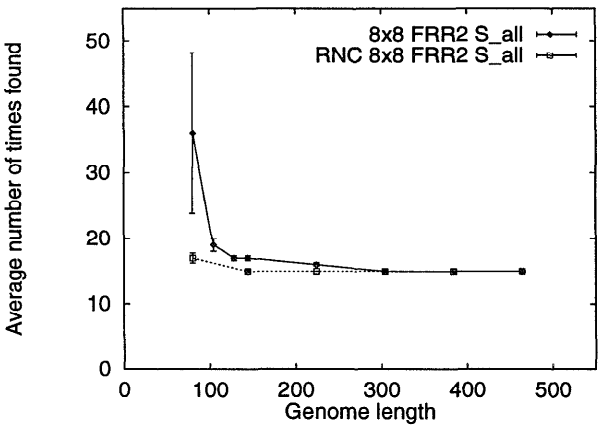


Figure 5: Average number of times building blocks are found vs. genome length: GA and RNC-GA on `8x8.FRR2`.

building blocks are found for the `8x8.FRR2` function. Both plots compare the performance of the modified GA with the performance of the original GA. Contrary to expectations, the modified GA performed better than the original GA in terms of both speed and stability. Similar behavior was seen in runs on other FRR functions. Analysis of these runs suggests the following reason for these unexpected results. While randomizing the non-coding bits may destroy previous building block parts, it also increases the mutation rate, and hence the exploration, in the non-coding regions. In effect, this modified GA saves existing building blocks with minimal risk of loss while continuing high-powered exploration in the as yet unused regions of the individuals. Though the randomization step may destroy previous building block parts, the detrimental effects of this action appear to be far outweighed by the benefits of increased exploration.

## Conclusions

The fact that the GA was able to solve most of the experiments described in this paper makes a strong statement for the effectiveness of the floating representation. Nevertheless, these are still preliminary results; additional studies are needed to fully understand the implications of the floating representation on the GA.

The first experiment looked into the density of building blocks in floating representation solutions. Results showed that the GA can effectively solve floating representation problems even with significantly reduced genetic material. The GA tends to use as much material as it is given: runs with longer genome lengths will scatter building blocks randomly all over the individuals while runs with shorter genome lengths will pack building blocks more densely and have more overlaps on the individuals. As expected, the probability that a solution will be found at all decreases as genome length shrinks and eventually becomes zero when it is no longer physically possible to fit all building blocks on an individual.

The second experiment studied the effects of negative building blocks on GA performance. Entire levels of building blocks were assigned negative fitness contributions and the average performance of the GA compared. The results suggest that the middle-most levels of building blocks are the least important to the GA. Lower level building blocks are the most important because the "pieces of the puzzle" must exist before the puzzle can be built. Upper level building blocks must be encouraged because they consist of such a large proportion of the entire solution.

The third experiment studied a modified GA in which non-coding bits were randomized during each generation of a GA run. This extra step was expected to diminish GA performance because it would remove any "memory" capabilities (for saving parts of previously existing building blocks) that non-coding segments may have provided the GA. Instead, results indicated that performance improved with randomization and this improvement was due to increased exploration in the non-coding regions.

## Acknowledgments

This research was sponsored by NASA/JSC under grant NGT-51057. The authors would like to thank John Holland for many discussions and suggestions relating to this work and Leeann Fu for many helpful comments on this article.

## References

Goldberg, D. E. 1989. Genetic algorithms and walsh functions: Part II, deception and its analysis. *Complex Systems* 3:153–171.

Holland, J. H. 1975. *Adaptation in Natural and Artificial Systems.* University of Michigan Press.

Mitchell, M.; Forrest, S.; and Holland, J. H. 1991. The royal road for genetic algorithms: Fitness landscapes and GA performance. In *Toward a Practice of Autonomous Systems: Proc. of 1st ECAL.*

Wu, A. S., and Lindsay, R. K. 1996. A comparison of the fixed and floating building block representation in the genetic algorithm. Forthcoming.

Wu, A. S. 1995. *Non-coding DNA and floating building blocks for the genetic algorithm.* Ph.D. Dissertation, University of Michigan.