A comparison of the fixed and floating building block representation in the genetic algorithm

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

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

December 13, 1996

To appear in *Evolutionary Computation* 4:2.

Abstract

This article compares the traditional, fixed problem representation style of a genetic algorithm (GA) with a new floating representation in which the building blocks of a problem are not fixed at specific locations on the individuals of the population. In addition, the effects of non-coding segments on both of these representations is studied. Non-coding segments are a computational model of non-coding DNA and floating building blocks mimic the location independence of genes. The fact that these structures are prevalent in natural genetic systems suggests that they may provide some advantages to the evolutionary process. Our results show that there is a significant difference in how GAs solve a problem in the fixed and floating representations. GAs are able to maintain a more diverse population with the floating representation. The combination of non-coding segments and floating building blocks appears to encourage a GA to take advantage of its parallel search and recombination abilities.

Keywords: genetic algorithm, non-coding segment, intron, building block hypothesis, Royal Road function, symbolic regression.

*Current email: aswu@aic.nrl.navy.mil

1 Introduction

The key elements which set apart the genetic algorithm (GA) from traditional artificial intelligence (AI) algorithms are its implicit parallelism and the crossover operator. That a GA operates on a population of individuals or potential solutions allows it to perform a parallel search for its solution. The crossover operator is unique in its ability to combine large amounts of information, which is expected to give a GA an advantage over incremental search algorithms. The building block hypothesis (BBH) of how GAs work states that a GA is expected to perform well on problems in which small building blocks or parts of the solution can be combined to form larger and larger building blocks, eventually leading to an acceptable solution. There is evidence that a similar process occurs in natural systems and certain elements of natural systems may encourage this process. For example, genes are not fixed at specific locations on chromosomes. Instead, they are demarcated by start and end patterns of nucleotides. This location independent representation allows for more variety and flexibility in gene arrangement, which in turn allows for the testing of arrangements that may be less likely to be separated by crossover. In addition, non-coding deoxyribonucleic acid (DNA), which refers to DNA that does not code for mature ribonucleic acid (RNA), is thought to improve the preservation and recombination of existing genes in a population. Perhaps the incorporation of similar structures into a GA would improve the GA's ability to find and recombine the building blocks of a computational problem.

Computation models of non-coding DNA and the location independence of genes can be added to a GA by modifying the way problems are represented in a GA. This article describes a *floating representation* which successfully incorporates non-coding segments (segments of an individual that make no contribution to the individual's fitness) and floating building blocks (building blocks that are location independent) into a GA. The key innovation of the floating representation is that building blocks are no longer fixed at specific locations on an individual. As a result, both building blocks and non-coding regions are dynamically arranged during a run. The experiments in this article comprise a systematic comparison of GA performance on the new floating representation and on the traditional "fixed" representation.

The goals of this study are twofold. First, we would like to study the problem solving process of the GA in hopes of better understanding how GAs work and on what types of problems they will excel. Though our goal is not to optimize parameterized problems, the concepts drawn from this study will be most applicable to problems that can be broken down multiple building blocks or parameters. In addition, we would like to investigate whether extending the analogy between genetics and the GA results in improved GA performance. In the process, we could also gain insight into the role of non-coding DNA in biological systems.

2 Biological Background

To better understand the motivations for this work, it is important to understand the biological background. This section describes relevant biological structures and discusses

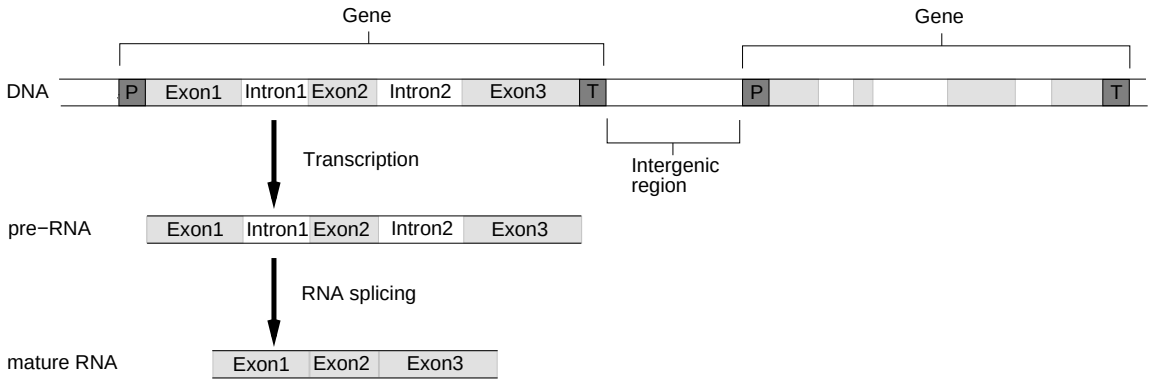

Figure 1: Non-coding DNA: intragenic regions and introns.

some of the parallels between the biological and computational systems. An understanding of the basic concepts of genetics is assumed. For a brief review of basic genetics, the reader is referred to (Wu and Lindsay 1996, Section 2).

2.1 Non-coding DNA

The non-coding segments used in these GA experiments are modeled after non-coding DNA from biological systems. Earlier GA papers (Levenick 1991; Forrest and Mitchell 1993) have used the term "intron" to refer to both the biological and computational structures. A careful study of introns from biological literature, however, suggests that this term is not an accurate description of what is being modeled in the GA. Introns are, in fact, one subset of a more general classification of DNA called non-coding DNA. As a result, the terms "non-coding DNA" and "non-coding segments" will be used in this article to refer to the respective biological and computational structures.

Non-coding DNA refers to all DNA that is not involved in the coding of a mature ribonucleic acid (RNA) product. Though non-coding DNA is prevalent in biological systems, its origin and function are as yet uncertain. Because a great deal of extra energy is required to sustain and process non-coding DNA, it must not contribute negatively to the genetic process or it would most likely have been eliminated by natural selection long ago. There are three types of non-coding DNA: intergenic regions, intragenic regions, and pseudogenes (Nei 1987; Lewin 1994). *Intergenic regions* are the regions of DNA in between genes. These regions are not transcribed into RNA. Some portions of intergenic regions are known to regulate the expression of adjacent genes; other portions have no known function. *Intragenic regions*, also called *introns*, are segments of DNA found within genes. Introns are transcribed into RNA along with the rest of the gene but must be removed from the RNA before the mature RNA product is complete. RNA that still contains the intron regions is often called *pre-RNA*. After the introns are spliced out of the pre-RNA, the remaining segments of RNA, the *exons* or *expressed regions*, are then joined together to become the mature RNA product. Figure 1 shows an example of intergenic regions, introns, and exons on a chromosome. A

pseudogene is a segment of DNA that is similar to a functional gene, but contains changes that prevent its translation. As they are not expressed, pseudogenes are not subject to selection pressure from the environment and consequently accumulate mutations quickly. When a pseudogene mutates enough that its similarity to a functional gene is no longer apparent, it becomes simply non-coding intergenic DNA.

2.2 The intron-exon structure of genes

The existence of the intron-exon structure has been particularly intriguing. Introns are only found in eukaryotic genomes and make up a large portion of the DNA in eukaryotic genomes. Eukaryotic organisms include all multicellular organisms and some single celled organisms. In humans, for example, approximately 30% of the human genome is made up of introns (Bell and Marr 1988). Only about 3% consists of coding DNA and the rest of the genome consists of other non-coding DNA, repetitive sequences, and regulatory regions. The unusual placement of introns, interrupting the coding regions of genes, and the fact that extra energy is needed to maintain and process these structures that have no apparent function, have made introns an important topic of study since their discovery in the 1970's.

The *exon theory of genes* (Blake 1978) suggested that exons are the building blocks of proteins, and genes are created from combinations of these building blocks. This theory lead to the *exon shuffling hypothesis* (Gilbert 1978; Gilbert 1985; Gilbert 1987; Gilbert 1991) which stated that introns increase the rate of recombination of exons and make it easier to move exons around and create new genes. Statistically, "...introns represent hot spots for recombination: by their mere presence and length they increase the rate of recombination, and hence the shuffling of exons, by factors of the order of 10^6 or 10^8 (Gilbert 1987, pg. 901)." Evidence suggests that exons may correspond to both structural and functional subunits of proteins (Blake 1978; Go 1981; Hartl 1991). There are specific examples of the same exon existing in different genes where the same structure or function is required by two different proteins (Gilbert 1985). Thus, the theory is that the intron-exon structure of eukaryotic genes encourages the formation of new genes from structural and functional subunits of existing genes. This process would certainly be more effective and efficient than building new genes one nucleotide at a time.

2.3 The arrangement of genes on a chromosome

A gene is demarcated by specific sequences of DNA called initiation and terminator sites. Since the existence of a gene depends on the existence of these specific sequences, the exact location of a gene on a chromosome need not be fixed. The presence of a gene in a genome is more important than its specific location on a particular chromosome (Curtis 1983, pg.353). Typically, the a specific gene appears at a relatively stable location within the same species, but may vary noticeably across different species.

The structure of the genetic code allows one segment of DNA to potentially code for more than one protein product. One way of to create multiple products is the alternative splicing

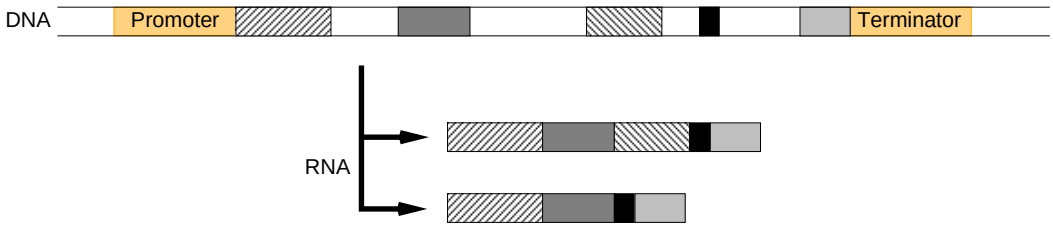

Figure 2: Alternative splicing allows one gene to code for several RNA products.

of introns. Figure 2 shows how one segment of DNA can code for several different proteins by splicing out alternative segments of the mRNA. A second way of coding for multiple products involves overlapping genes (Curtis 1983, pg. 304). The genetic code is stored in codons which are three base pairs long. A *reading frame* is one of the three possible ways of reading a nucleotide sequence as a series of triplets (Lewin 1994). For example, the sequence:

```
ACGACGACGACGACGACGACG
```

can be read in three possible reading frames:

```
ACG ACG ACG ACG ACG ACG ...
 CGA CGA CGA CGA CGA CGA ...
  GAC GAC GAC GAC GAC GAC ...
```

As a result, two completely unrelated genes can overlap or one gene can be contained within another. First discovered in 1977, overlapping genes explained a serious problem that had developed with the bacteriophage $\phi\chi174$. Without the overlaps, the amount of DNA in $\phi\chi174$ had been calculated to be insufficient to code for all of the known protein products of $\phi\chi174$. Thus, overlapping genes are known to provide more efficient storage of information.

2.4 What does this have to do with a GA?

The exon shuffling hypothesis states that introns increase the recombination rate of exons and assist in the creation of genes from exon building blocks. Segments of DNA that code for useful structural and functional protein subunits could be arranged and rearranged to form proteins in the process of evolution. Similarly, the non-coding DNA found in between genes would be expected to aide in the recombination of genes. This theory bears a striking resemblance to the building block hypothesis of GAs. One way to test the exon shuffling hypothesis in a GA would be to observe the effect of introns on GA performance. Earlier work on this topic has suggested that non-coding segments can improve both the speed and stability of a GA. Introducing a floating building block structure in a GA takes the GA model of genetics one step further. This addition not only mimics the location independence of genes and allows for overlapping building blocks, it also allows the GA to decide where to place non-coding and coding segments to create good building blocks. The study of non-coding segments and floating building blocks in a GA, in effect, investigates both the exon shuffling hypothesis and the building block hypothesis.

3 Related Work

Though GAs have been successfully applied to a number of real world problems, our understanding of the GA itself is still far from complete. Research into the theory of GAs continues in many areas including the search for optimal control parameter values, the study of the effects of various genetic operators, and the study of problem representations. The work described in this article is geared toward developing a more effective problem representation style for GAs based on ideas from natural systems.

A number of studies have already incorporated the concept of non-coding segments into a GA. Forrest and Mitchell (1993) and Wu and Lindsay (1995) investigated the effects of non-coding segments on GA performance on the Royal Road (RR) functions (to be described in section 4). Non-coding segments were added to the RR functions by inserting extra bits that made no contribution to the fitness of an individual in between the basic (lowest level) building blocks. These studies found that the addition of non-coding segments resulted in a noticeable increase in the stability of the GA, but produced varying results on the speed of the GA. Levenick (1991) also investigated relative building block placement by inserting non-coding segments into a GA. The artificial evolution problem used by Levenick was similar to the RR function; both consisted of a hierarchical structure of pre-defined building blocks. Results indicated that the runs with non-coding segments are more likely to find a solution than the runs without non-coding segments.

The variable length individuals of genetic programming (GP) systems generate non-coding material as a natural by-product. Initially considered useless material, recent studies suggest that non-coding material may actually improve the performance of GP systems. Haynes (1996) found that the addition of duplicate coding segments significantly sped up the GP learning process on a clique detection problem. Results from Nordin, Francone, and Banzhaf (1995) suggest that non-coding material, both naturally occurring and explicitly defined, may protect the code blocks of a GP system from the destructive effects of crossover.

Goldberg's messy GA (mGA) (Goldberg, Korb, and Deb 1989) investigated the concept of location independent bits instead of location independent building blocks. In the mGA, individuals are strings of ordered pairs. The first element of a pair is the name of the bit; the second is the value. By allowing the bits to be placed in arbitrary orders, the mGA encourages the formation of tightly linked building blocks whose component bits may or may not be numerically contiguous. Thus, in addition to testing various bit values, the mGA also considers the effect of different associations between bits. Results indicate that the ability to dynamically arrange the bits (and consequently, building blocks) of an individual can have a significant effect on the performance of the GA.

i	Schema b_i	Level l_i	Fitness contribution c_i
0	11111111**	0	1
1	********11111111**	0	1
2	****************11111111**	0	1
3	************************11111111**	0	1
4	********************************11111111************************************	0	1
5	**11111111****************************	0	1
6	**11111111********************	0	1
7	**11111111	0	1
8	1111111111111111**	1	1
9	****************1111111111111111**************************************	1	1
10	********************************1111111111111111****************************	1	1
11	**1111111111111111	1	1
12	11111111111111111111111111111111**********************************	2	1
13	********************************11111111111111111111111111111111	2	1
14	1111111111111111111111111111111111111111111111111111111111111111	3	1

Table 1: Definition of 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 schema and a coefficient value c_i which indicates the fitness contribution of that schema.

4 The Royal Road Functions

The Royal Road (RR) functions (Mitchell, Forrest, and Holland 1992; Jones 1994) are a class of functions designed for the study of building block interactions. These functions are characterized as having a pre-defined optimum solution, pre-defined building blocks, and a hierarchical building block structure. The pre-defined nature of these functions allow us to trace a GA's progress as it searches for the known optimum individual. The upper level building blocks in the hierarchy are composed of combinations of lower level building blocks creating an ideal function for investigating the BBH. The short basic (lowest level) building blocks were expected to be easy for a GA to discover and the upper level building blocks were expected to encourage the recombination and preservation of lower level building blocks.

The RR functions used in these experiments were derived from the RR functions from (Mitchell, Forrest, and Holland 1992) and (Forrest and Mitchell 1993). Table 1 shows an example of an RR function. The optimum individual of this function is an individual consisting of all ones. Each schema, b_i, represents all individuals that contain building block i. The `*` is a *don't care* bit whose value can be either `0` or `1`. For example, b_0 represents the set of all individuals that contain building block 0 or all ones in the first eight bits. b_8 represents the set of all individuals that contain building block 8 or all ones in the first sixteen bits; this set includes the sets of individuals containing building blocks 0 and 1. For each b_i, there is a value l_i which indicates the level of that schema and a coefficient value c_i which indicates the fitness contribution of that schema. The fitness of an RR individual is the sum of the fitness contributions of all building blocks whose optimum value is found on that individual. For example, the following example individual,

```
11111111 11111111 00101000 11001111 10010010 00010101 11111111 00100010
```

	Start tag	Identity tag	Optimum pattern
Building block 1	01	001	11001101
Building block 2	01	010	01110011

```
Example individual:         00101010011100110110101001110011101
Potential building blocks:   ^ ^ ^  ^    ^  ^  ^ ^  ^    ^  ^
Existing building blocks:      2 1            2
```

Figure 3: Given two example building blocks, the example individual has one copy of building block 1, two copies of building block 2, and 11 potential building blocks.

contains the optimum value for building blocks 0, 1, 6, and 8. The fitness of this individual is, $f = c_0 + c_1 + c_6 + c_8$.

5 A Floating RR Representation

A floating representation version of the RR function was created by making the basic building blocks of the RR functions location independent. Initially, a start tag and an identity tag were assigned to each basic building block. No end tag was needed because all basic building blocks were of fixed length. The start tag indicated the possible existence of a building block and was the same for all building blocks. The identity tag was found immediately to the right of the start tag; a unique identity tag was assigned to each building block. The body of the building blocks was found immediately to the right of the identity tag. For example, given the two "floating building blocks" in figure 3, the accompanying individual has 11 potential building blocks and three optimum building blocks: one copy of building block 1, two copies of building block 2. Upper level building blocks are still composed of combinations of basic building blocks and exist when all of their component basic building blocks exist. Because the building blocks are location independent, their relative placement is entirely determined by the adaptive process. There may be zero or more copies of a building block and building blocks may overlap each other.

In the RR functions, because there is only one optimum pattern for each building block and individuals only receive the fitness contributions of the existing building blocks, the GA is essentially searching for an individual containing n specific patterns, where n is the number of basic building blocks. With the traditional fixed representation, the n patterns are fixed at specific locations on an individual. With the floating representation, the patterns can appear anywhere on an individual and identity tags ensure that the n patterns are unique. To find floating building blocks on an individual, we used the search algorithm from Aho and Corasick (1975) which, given a list of character strings (i.e. the list of start tag + identity tag + optimum pattern for each building block), finds all instances of all strings on a separate character string (i.e. an individual) in one pass, including overlapping strings.

Basic building blocks (Level 0)		Upper level building blocks		
Bldg block	Optimum pattern	Bldg block	Level	Component bldg blocks
0	********	8	1	0 1
1	********	9	1	2 3
2	********	10	1	4 5
3	********	11	1	6 7
4	********	12	1	0 1 2 3
5	********	13	1	4 5 6 7
6	********	14	1	0 1 2 3 4 5 6 7
7	********			
	Start tag = 11			

Table 2: Definition of the floating RR function, `8x8.FRR2`. Optimum building block patterns are generated randomly at the start of each run and differ from run to run.

Figure 4: The parts of a basic building block in floating representation.

In initial experiments, building blocks whose optimum patterns consisted of all ones were used. Because of this pattern, many building blocks overlapped easily. In order to ensure that there was no bias from the easily overlapping building blocks, all later experiments, including the ones described here, used randomly generated building block optima. The optimum value of each building block is randomly generated at the start of each run and all optima are required to be unique, eliminating the need for a start tag. A typical "floating Royal Road" (FRR) function is shown in table 2. Figure 4 shows an example of a floating basic building block. The fitness of an individual is still the sum of the fitness contributions of all building blocks whose optimum values exist on that individual. Building blocks whose optimum values have not been found make no contribution to the fitness of the individuals. If there are multiple instances of a building block on an individual, the extra copies also make no contribution to the fitness. All non building block areas are considered to be non-coding segments.

6 Motivations

The floating representation introduces two key structures into a GA: location independent or *floating* building blocks and non-coding segments. Computationally, there are a number of arguments that suggest that these structures may improve GA performance. The purpose of the experiments here is to investigate some of these hypotheses.

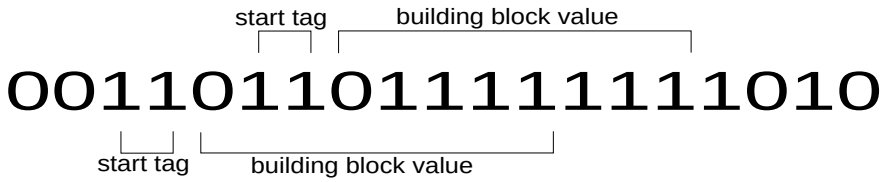

Figure 5: An example of overlapping building blocks in the FRR functions.

- Non-coding segments increase the probability of recombination of existing building blocks and decrease the probability of destroying existing building blocks. Suppose we take an example (fixed representation) RR function which has n basic building blocks, each m bits long where n and m are positive integers. Let $l : l \geq 0$ equal the number of non-coding bits inserted for each basic building block (typically the same number of bits is inserted in between each pair of basic building blocks). When $l = 0$, no non-coding bits are inserted. Let L = length of an individual $= n(m + l)$. Then the probability that crossover will fall in between two building blocks is $\frac{n(l+1)}{n(m+l)} = \frac{l+1}{m+l}$. The probability that crossover will fall within a building block is $\frac{n(m-1)}{n(m+l)} = \frac{m-1}{m+l}$. Comparing $l = 0$ and $l > 0$, we find that $\frac{1}{m} < \frac{l+1}{l+m}$ (probability of crossover falling in between building block boundaries is lower when $l = 0$) and $\frac{m-1}{m} > \frac{m-1}{m+l}$ (probability of crossover falling within a building block boundary is higher when $l = 0$).

  It should be noted that the above two equations basically mean that the addition of non-coding segments lessens the "activity" of crossover within building blocks. While this may reduce the chance of destroying an existing optimum building block, it may also reduce the exploration for the a building block whose optimum has not yet been discovered, potentially increasing the total search time.

- The floating representation allows building blocks to appear more than once on an individual. As long as one copy of the optimum value of a building block exists on an individual, the fitness function will add that building block's contribution to the fitness of that individual. Extra copies of the optimum have no effect on the fitness, but do provide a GA with one or more "free mistakes" – a GA can alter or destroy extra copies without decreasing the fitness of the individual.

- The floating representation allows for overlapping building blocks, as shown in figure 5. Overlapping building blocks reinforce each other – the overlapping segment of the individual contributes in more than one way to the fitness of the individual and, in effect, is worth more. As a result, overlapping building blocks are more difficult to remove from a population. In general, this is advantageous in that it helps to preserve building blocks in the population. There are also situations, however, such as when the arrangement of a group of overlapping building blocks prevents the expression of other building blocks or (with multi-value parameters) when a good value from one building block locks in a not-so-good value for an overlapping building block, where overlapping building blocks may be a disadvantage.

- Another advantage of overlapping building blocks is more efficient use of space or genetic resources. Since floating building blocks can occur at any location on the individual, it is theoretically possible to have a maximum of $L - m' + 1$ building blocks on each individual where L is the length of the individual and m' is the length of the building block including the tags. With fixed, non-overlapping building blocks, the maximum number of building blocks that could fit on an individual is $\lfloor L/m \rfloor$ where $m \approx m'$ is the length of the building block. In effect, a GA seems to have more resources with which to work with the floating representation.

7 Performance Criteria

A number of performance criteria were compared in these experiments. The most common method of evaluating GA performance is to record the number of generations or function evaluations required before a GA finds an acceptable solution to the problem. The obvious goal is to minimize this value as this would also minimize the time and computing power used. A second method is to monitor a GA's progress as it solves a problem, for example, record the first generation in which a building block from each level is found. Does a GA find building blocks in evenly spaced intervals of time or are there periods when it is more productive? The hierarchical structure of the Royal Road functions makes them ideal for studying this topic. A third method is to note the stability of a GA: count the number of times building blocks are found in the population. The definition of *found* includes the first time a building block is discovered in the population, including the initial population, and any later re-discoveries of the building block if it is lost from the population. GA stability depends on the balance of exploration and exploitation. According to the BBH, a GA should save the good building blocks that it finds while continuing to search for other building blocks. Two stability values are recorded for each run. Let
$s_i, i = 1, 2, ..., num\_building\_blocks$, equal the number of times building block i is found. Then, for each run,

$$S_{basic} = \sum_{i=1}^{num\_basic\_bb} s_i \qquad S_{all} = \sum_{i=1}^{num\_bb} s_i$$

where S_{basic} is equal to the sum of the number of times the basic building blocks are found and S_{all} is equal to the sum of the number of times all of the building blocks are found.

8 Royal Road Experiments

This section describes a systematic comparison of a GA's performance on the fixed and floating representations. Experiments were performed on the RR functions. The structure of each function can be determined from its name which is in the form `mxn.f`. `m` indicates the number of basic building blocks in the function, `n` indicates the length of a basic building block in bits, and `f` refers to the structure of the Royal Road function. For this work, `f` will

Non-coding segment length	Genome length (8x8)	Genome length (16x8)
0 (RR2)	64	128
1 (FRR2)	-	144
2 (FRR2)	80	-
10	144	288
20	224	448
30	304	608
40	384	768
50	464	928

Table 3: Genome lengths tested in the 16x8 and 8x8 experiments.

have one of two values: RR2 refers to the fixed representation of the RR function and FRR2 refers to the floating representation of the RR function. The specific functions tested in this study were 8x8.RR2, 8x8.FRR2, 16x8.RR2, and 16x8.FRR2. A detailed description of our GA program can be found in (Wu and Lindsay 1995). The following parameters remained constant throughout all of our experiments:

Crossover type:	one point	Parent replacement:	YES
Crossover probability:	0.7	Mutation probability (per bit):	0.005
Selection method:	roulette wheel	Population size:	256
Range of number of offspring per individual: 0 to 2			

The roulette wheel selection method is a fitness proportional selection method. The selection probabilities for each individual are scaled to fit the expected number of offspring. All building blocks have a fitness contribution of one point, regardless of size or level. Each experiment was run 50 times.

Because the building blocks of the FRR function are location independent, it is impossible to enforce the placement of a given number of non-coding bits in between pairs of building blocks. As a result, these experiments compare on the effects of increasing genome length instead of non-coding segment length on the RR and FRR functions. The *genome length* is the length of the individuals in the GA population and can be calculated with the following equations.

$$\#\ significant\ bits = \#\ basic\ building\ blocks \times basic\ building\ block\ length$$

$$genome\ length = \#\ significant\ bits + \#\ basic\ building\ blocks \times non\text{-}coding\ segment\ length$$

As the genome length increases, the number of non-coding bits found on an individual also increases. Table 3 lists the genome lengths that were tested in the experiments described in this section.

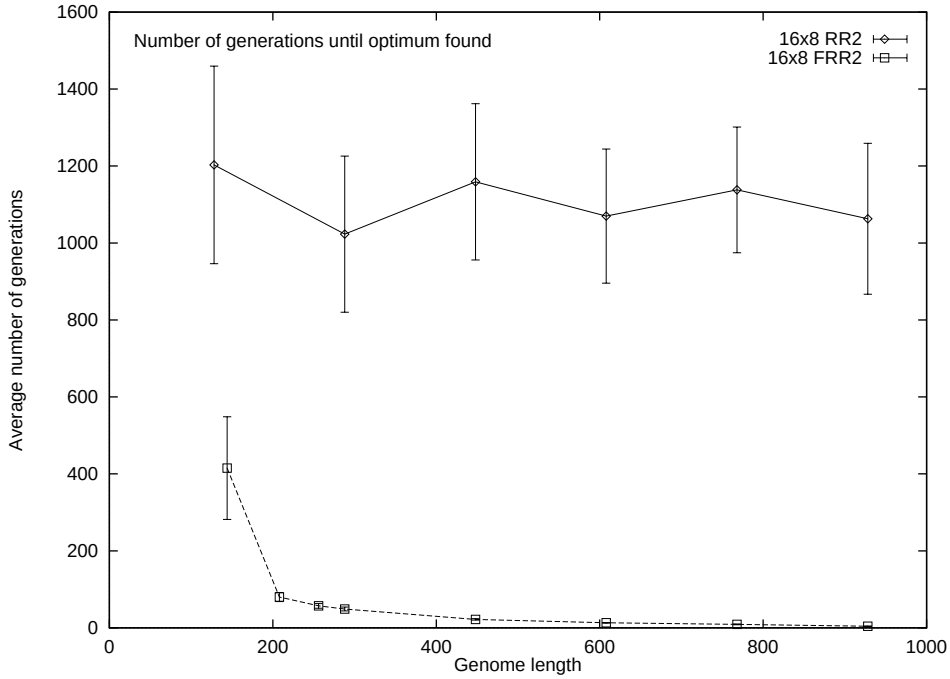

Figure 6: Average number of generations needed to find the first optimum vs. genome length: `16x8.RR2` and `16x8.FRR2`.

8.1 Finding a single optimum individual

We observed the behavior of a GA as it searched for a single optimum individual in both the fixed and floating Royal Road functions. Each experiment was run 50 times and the average values and 95 percent confidence intervals are reported here.

8.1.1 Results

Figure 6 plots a typical comparison of the average number of generations needed to find an optimum individual as genome length increases. Figure 7 plots a typical comparison of the average number of times building blocks are found as genome length increases. Both plots show that, when the genome length is short, GA performance is noticeably better (faster and more stable) on the fixed representation than on the floating representation. As the genome length increases, performance on the floating representation improves significantly and quickly becomes much better than performance on the fixed representation. Performance on the fixed representation shows little or no improvement as genome length increases. Table 4 shows the average generation at which each level of an RR and FRR function is discovered. The data here is consistent with the above results in that there is little change with the fixed representation and significant improvement with the floating representation as the genome length increases. In both the fixed and floating runs, however, the number of generations until the next level is achieved increases significantly with each higher level. These results are

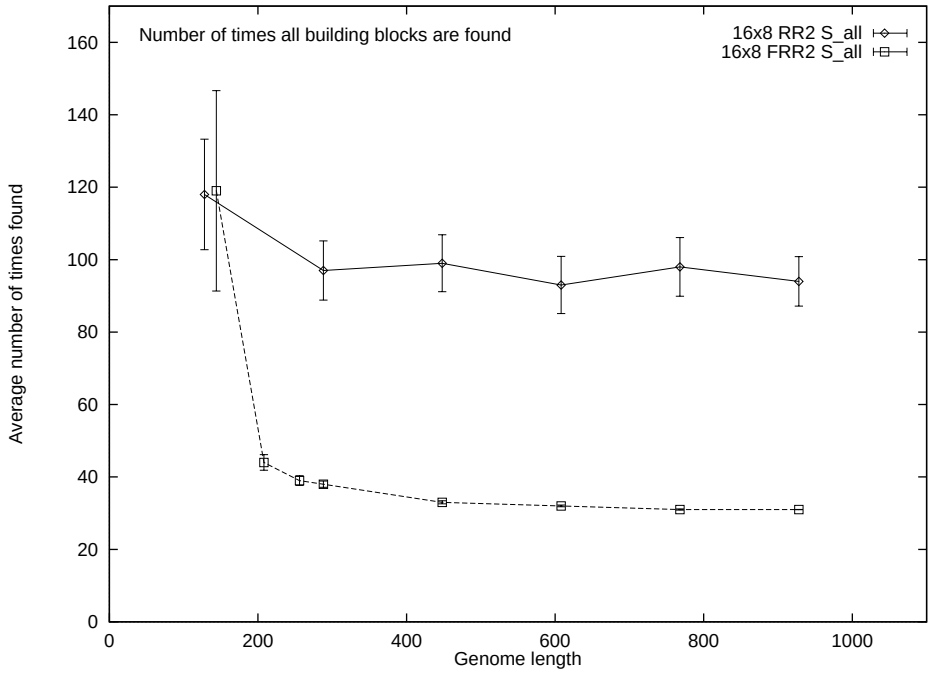

Figure 7: Average number of times building blocks are found vs. genome length: `16x8.RR2` and `16x8.FRR2`.

typical of all of the fixed vs. floating comparisons done, regardless of the specific function used.

8.1.2 Discussion

Closer observation of the individual runs suggests that these differences in performance may be due in part to the demands on a GA as it solved each type of problem. With both representations, a GA must find an individual with at least one optimal copy of each building block. With the floating representation, there is the additional task of arranging the building blocks on the individuals, a non-trivial task with short genome lengths. Bad placement of building blocks discovered early in a run can significantly slow down or prevent the discovery of other building blocks later in the run. With the fixed representation, building block locations are fixed, there is no danger of needing to rearrange building blocks because of an initial bad placement, and a GA needs only to search for the optimum value of each building block.

At longer genome lengths, the floating representation gains a major advantage over the fixed representation. Since a GA is free to use all of the genetic material available, there is now more material with which to explore. Longer genome length result in increased chance that a particular building block will be found on an individual and in increased non-coding material which is thought to provide stability. GA performance on floating representation functions levels off at optimal values when genome lengths become long enough that the

16x8.RR2						16x8.FRR2					
Genome	Levels					Genome	Levels				
length	0	1	2	3	4	length	0	1	2	3	4
128	0	12	80	333	1125	144	0	0	2	28	349
288	0	9	77	307	942	288	0	0	0	9	49
448	0	11	85	375	1159	448	0	0	0	3	2
608	0	12	74	376	1070	608	0	0	0	1	13
768	0	11	107	387	1138	768	0	0	0	0	9
928	0	12	89	337	1063	928	0	0	0	0	4

Table 4: Average generation at which each level is discovered vs. genome length: `16x8.RR2` and `16x8.FRR2`.

chance of finding all building blocks in a randomly generated individual approaches 100%. The fixed representation functions uses the same amount of genetic material to code for a solution regardless of the genome length. Increasing the genome length, however, decreases the chance of crossover at any particular bit of an individual. This decreased activity seems to stabilize the GA but is also thought to reduce the exploration within building block boundaries, two actions that have opposite effects on GA performance. As a result, GA performance on the fixed representation functions shows no significant changes as genome length increases while GA performance on the floating representation functions improves noticeably.

Given the results from figures 6 and 7, one might ask, "why not simply use the floating representation with an extremely long genome length since this combination seems to yield near optimal performance?" There are two main reasons for minimizing genome length. First, longer individuals require more time to process. For the GA program used here, the computational complexity per generation is on the order of $O(p^2 + pL)$ where p is the population size and L is the genome length. The fitness evaluation and reproduction routines are of order $O(pL)$. The roulette wheel selection for parents is of order $O(p^2)$. If the genome lengths is increase by r bits, then the computation complexity for one generation becomes $O(p^2 + p(L + r)) = O(p^2 + pL + pr)$. Over many generations, the extra term will noticeably slow down a GA run, especially when $r \gg L$. Second, the simplicity of the RR functions (in particular, the pre-defined building block optimum patterns and the additive fitness) mean that a GA is essentially searching for a pre-defined set of unique patterns when searching for a solution to these functions. Each pattern being sought is a string composed of the tags and optimum pattern of a building block. Obviously as the genome length increases, the chance that all building blocks will exist on an individual increases. Real world problems, on the other hand, are typically more complex than simply searching for a set of pre-defined patterns and often involve multi-valued building blocks that interact. In these problems, a GA must decide which pattern of each building block works best with the chosen patterns of other building blocks. Longer genome lengths may increase the chance of all of the patterns of all of the building blocks existing on an individual, but have no effect on deciding which of the existing values will work well together (a task which becomes more difficult as the

number of candidate values increase). As a result, when using the floating representation, it would be desirable to be able to calculate the genome length value at the sharp bend, i.e. in figures 6 and 7, approximately 200, for maximum performance and minimum computation.

With the floating representation, the length of the start tags of the basic building blocks determines the frequency of potential building blocks on an individual. The short two-bit tags used in the experiments here causes each location of an individual to have a one in four chance of being the start of a building block. Earlier tests on varying start tag lengths found little change in relative GA performance as genome length increases. Performance is poor with shorter genome lengths, quickly improves, and levels off as the genome length increases. The only noticeable difference to increasing the start tag length is that the performance plot of the GA on the floating representation shifts to the right, that is, the genome length at which performance quickly improves becomes longer as the start tag gets longer. Statistically, longer genome lengths are required to yield the same frequency of discovery of longer start tags.

8.2 Looking within the runs

To better understand the difference in results between the fixed and floating representations, it is necessary to investigate the behavior of a GA during individual runs. In examining individual fixed and floating representation runs, we find some interesting differences that begin to explain the above results.

8.2.1 Results

Figure 8 plots data from a typical fixed representation run in which there are not non-coding bits. Three values are plotted each generation of the run: the fitness of the best individual in the population, the average fitness of the entire population, and the average Hamming distance of the entire population from the best individual in the population. Hamming distance refers to the number of bits that differ between two individuals. The best fitness plot increases in discrete steps because building block fitnesses are discrete. The average fitness follows as the best fitness increases, also increasing in a stair-step fashion, that is, there are times during the run when the average fitness plot has growth spurts and other times when it levels off noticeably. While this is expected in the best fitness graph, it is somewhat surprising in the average fitness graph. One would expect the average fitness of a diverse evolving population to increase gradually since the members of a diverse population should have a variety of good building blocks. The average Hamming distance is also affected by the best fitness value. Each major jump in the best fitness of the population is followed by a slight rise then a noticeable drop in the Hamming distance. Soon after that drop, the Hamming distance levels off until the next jump in the best fitness.

Figure 9 plots data from a typical floating representation run. This run also uses individuals with minimal non-coding bits (segment lengths of 1 or 2). The most noticeable difference of the plots in figure 9 from those in figure 8 is the absence of the stair-step shape

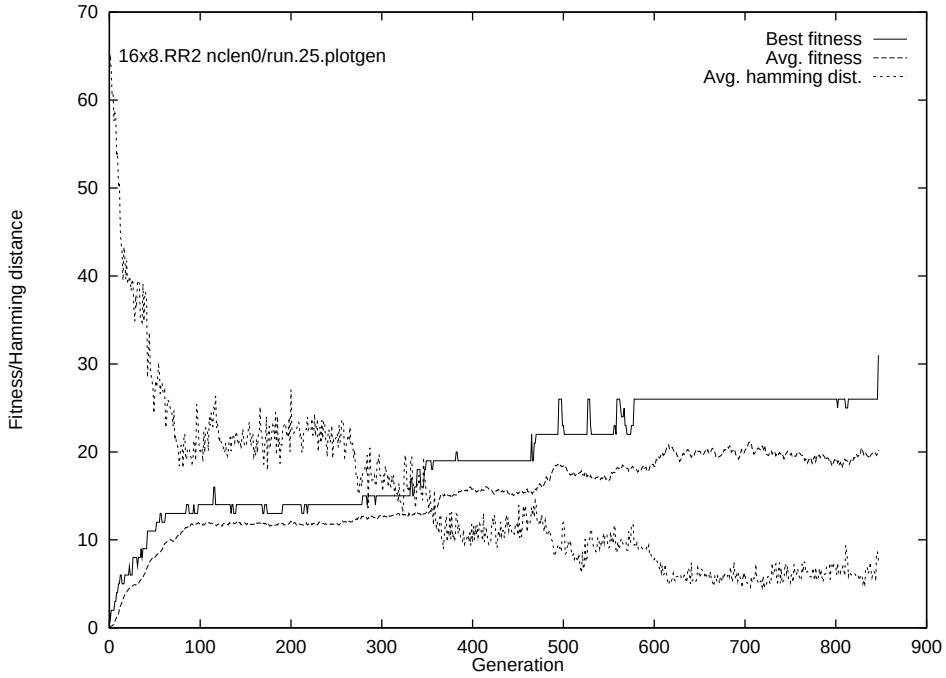

Figure 8: Typical RR2 run searching for a single optimum. For each generation, the best individual fitness, average population fitness, and the average Hamming distance of the population from the best individual are plotted.

in the average fitness and Hamming distance plots. The average fitness still increases slightly behind the best fitness; however, this increase is steady, almost smooth in some runs. The Hamming distance also drops steadily during each run. The Hamming distance plot oscillates more than the average fitness plot, but does not level off noticeably at any point.

Figure 10 shows the density of the individual building blocks in example fixed and floating representation runs. These plots show the number of each building block in the population at each generation. The left plot is from a typical fixed representation run; the right plot is from a typical floating representation run. Each column represents one basic building block, as labeled along the x-axis. In this function, building blocks zero to fifteen are basic building blocks and building blocks sixteen to thirty are upper level building blocks. Each row along the y-axis represents one generation. Generation 0 is at the bottom and the final generation is at the top. Thus, the cell at x=2 and y=5 shows the amount of building block 2 in generation 5. Since these two plots represent different runs which ended at different generations, the heights of the plots have been scaled for easier comparison. The shades of gray at each cell are scaled proportionally to indicate the density of each building block in the population at each generation. A white cell means 0% (zero copies of the building block) and a black cell means 100% (256 or more copies). The floating representation runs may have more than 100% because individuals can have more than one copy of a building block. The fixed representation graph is mostly either white or black, reflecting the tendency of the fixed representation runs to fill the population with new building blocks one at a time. The

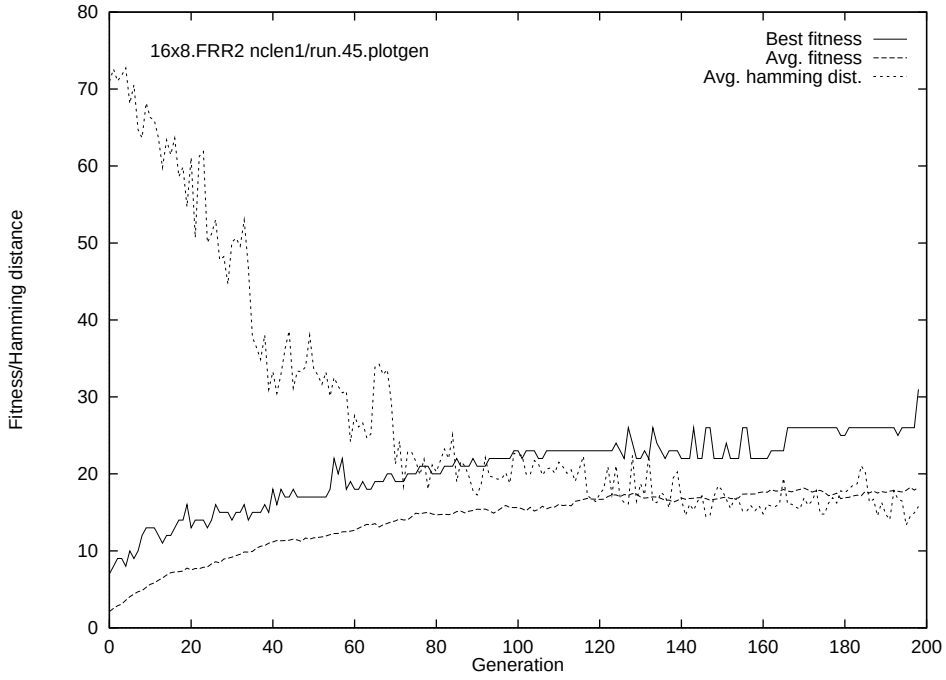

Figure 9: Typical FRR2 run searching for a single optimum. For each generation, the best individual fitness, average population fitness, and the average Hamming distance of the population from the best individual are plotted.

floating representation graph has more medium gray shades, indicating the preservation of at least a moderate number of each building block in the population, throughout the run. Even though a select few building blocks may still fill the population in the floating runs, other building blocks are still relatively well represented in the population.

Figure 11 shows a different perspective of the two runs from figure 10. In these plots, white cells indicate zero copies of a building block; gray indicates one copy; and black indicates two or more copies. In the fixed representation example, the same pattern that is seen in the scaled gray shade plot is seen here. The GA seems to find basic building blocks one at a time and either has zero copies of a building block or many copies. Unless a building block is propagated to the majority of the population, it is unstable and easily lost. In the floating representation example, there is almost always at least one copy of each basic building block in the population at all times, usually two or more copies. Most of the level 1 building blocks are also well represented in the population during the run suggesting that the GA's task is more focussed on testing different combinations of building blocks rather than filling the population with existing building blocks. This testing of different combinations of basic building blocks is the desired behavior from a GA.

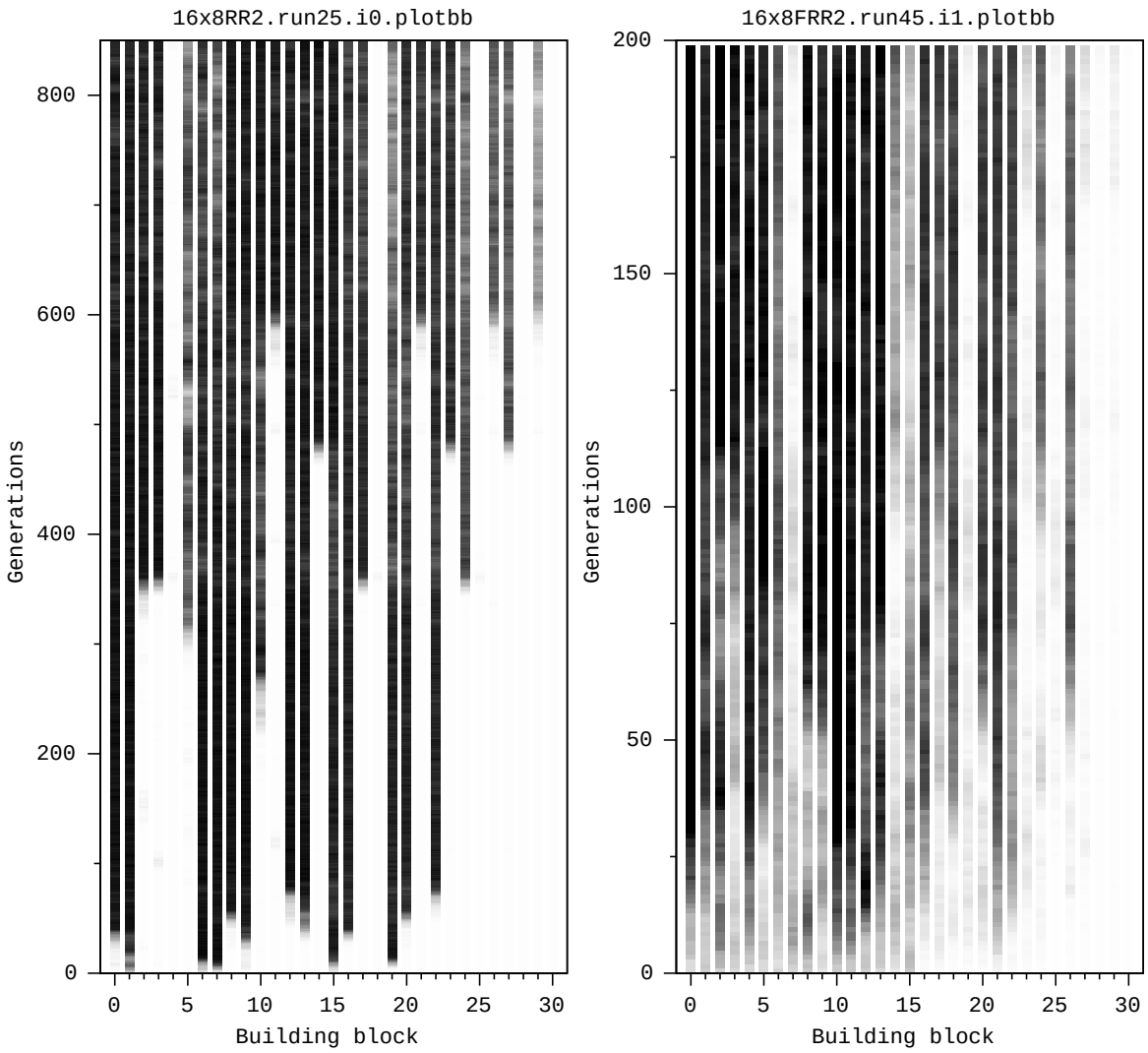

Figure 10: Building block density during fixed and floating Royal Road runs: `16x8.RR2` and `16x8.FRR2`. White cells indicate zero copies of a building block; black cells indicate 256 copies.

8.2.2 Discussion

The differences in behavior seen in this section seem to be due to differences in how a GA searches for building blocks with the fixed and floating representations. As a fixed representation run begins, a few of the basic building blocks quickly fill up the entire population. In the process, many other existing building blocks are completely lost from the population presumably due to the hitchhiking effect (Forrest and Mitchell 1993). As a fixed representation run progresses, each time a new building block is discovered, the GA attempts to fill up the entire population with this new building block before continuing the search for other building blocks. As a result, except for the initial few generations, all individuals in the population have approximately the same optimal bits and the same set of good building

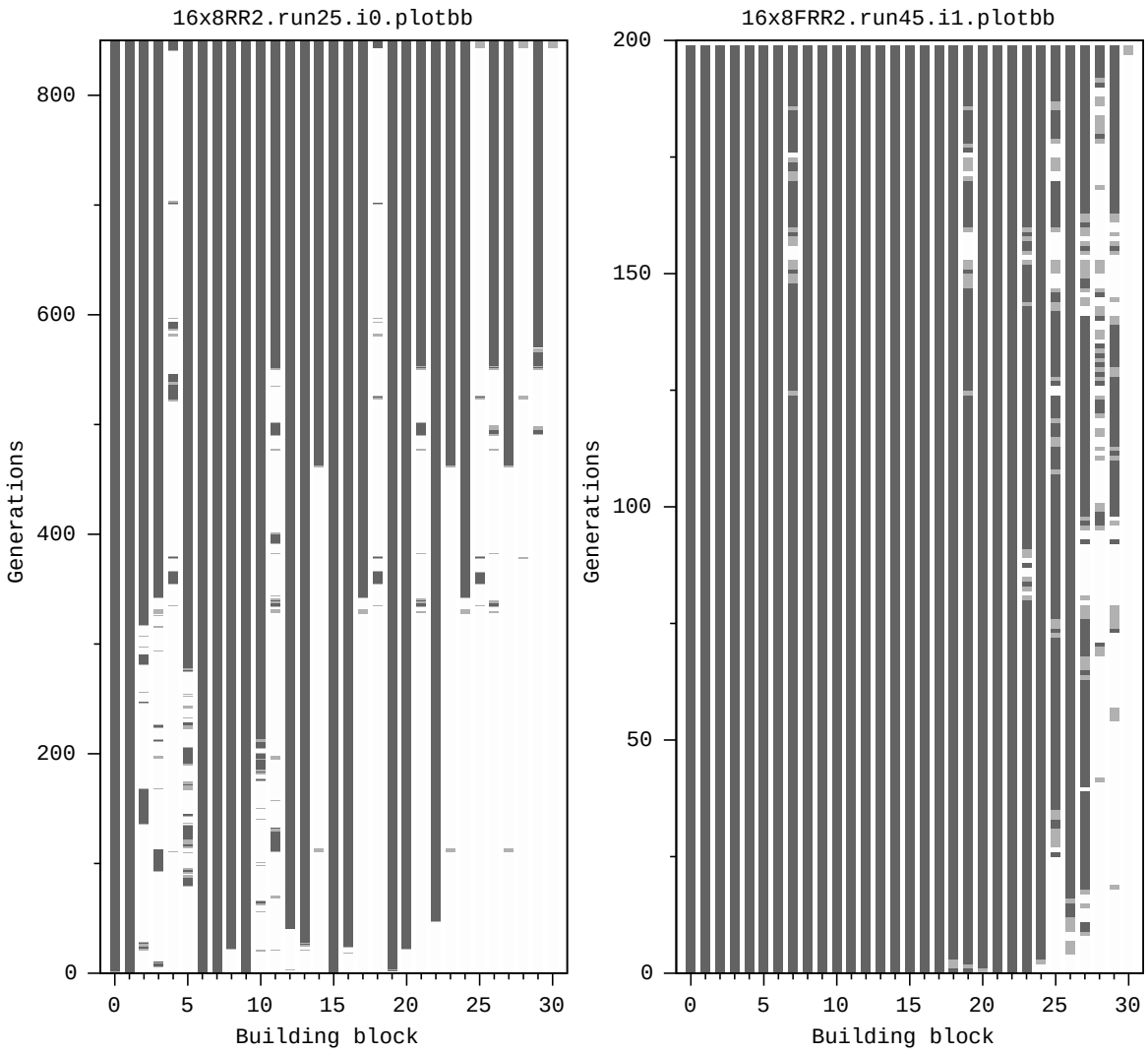

Figure 11: Building block density during fixed and floating Royal Road runs: `16x8.RR2` and `16x8.FRR2`. White cells indicate zero copies of a building block; gray cells indicate one copy; black cells indicate two or more copies.

blocks throughout the entire run. This explains the stair-step shape of the average fitness and Hamming distance plots from Figure 8. The discovery of a new building block results in a new "best individual" that has a building block that no other individual has. This difference causes a temporary rise in the average Hamming distance of the population from the best fitness. As the members of the population all gain one more building block in common, however, the average Hamming distance once again levels off. The increases in average fitness occur at the same time as the decreases in Hamming distance, marking the intervals when the GA is distributing newly discovered information throughout the population. Those intervals when these two plots level off indicate times when the GA is searching for new building blocks. At those times, most of the population have the same

combination of optimal building blocks. Thus, while it is unclear whether crossover or mutation drives the search for new building blocks, a major task of the crossover operator seems to be the propagation of information throughout the population.

The characteristics of the floating representation — basic building blocks can appear anywhere on an individual, can appear multiple times, and can overlap — give genetic operators such as crossover and mutation a more dynamic and less predictable effect on the population. Compared to the fixed representation where one genetic operation affects at most one basic building block at a time, genetic operations on floating representation individuals can create and destroy multiple building blocks at a time. As a result, it is more difficult for a GA to converge prematurely to local maxima and a more diverse population is maintained, resulting in the gradual (not stair step) changes in the average fitness and Hamming distance plots from figure 9. The values change gradually because the population consists of a greater variety of different individuals. Different, in this case, refers to the binary patterns of the individuals of the population, not necessarily to the number of different building block combinations that the individuals have. Since two individuals can easily have the same building blocks in different locations, Hamming distance can only measure bit diversity and not building block diversity. Nevertheless, it still provides a useful measure of diversity — non-homogeneous individuals are be more likely to consist of different building blocks than homogeneous individuals — as well as a good mode of comparison with fixed representation runs. Diversity in the population does not prevent extremely fit building blocks from filling up the entire population. It just means that the population is able to maintain individuals with many different combinations of building blocks in different placement schemes.

Adding non-coding bits (increasing the genome length) results in some changes: GA performance on the floating representation speeds up significantly, GA stability improves on the fixed representation, and the Hamming distance in both representations levels off at higher values, presumably due to the increased number of bits that are not subject to selection pressure. The overall GA behavior on each type of representation, however, remains consistent. With the fixed representation, the GA tends to find and propagate one building block at a time and maintains a relatively homogeneous population. With the floating representation, the GA is able to maintain a moderate number of copies of each building block throughout a run and maintains a more diverse population.

These differences are further reflected in GA behavior when searching for more than one optimum individual. There are many types of problems where one might expect more than one solution; problems where a set of solutions or good alternatives is desired rather than a single solution. Several sets of experiments were performed in which the runs were extended until 50% of the population consisted of optimum individuals. Figure 12 plots, for both the fixed and floating representations, the average percentage of the time of a run that is spent searching for the first optimum. With the fixed representation, between 70% and 85% of the time of a typical run is spent searching for the first optimum individual and less than 20% of the time is spent filling the population to 50% optima. These values concur with our earlier observations that by the time a GA finds one optimum in the fixed representation runs, most of the rest of the population are already of near optimal fitness. All that the GA really needs to do then is to propagate the most recently found building block to half of the population to

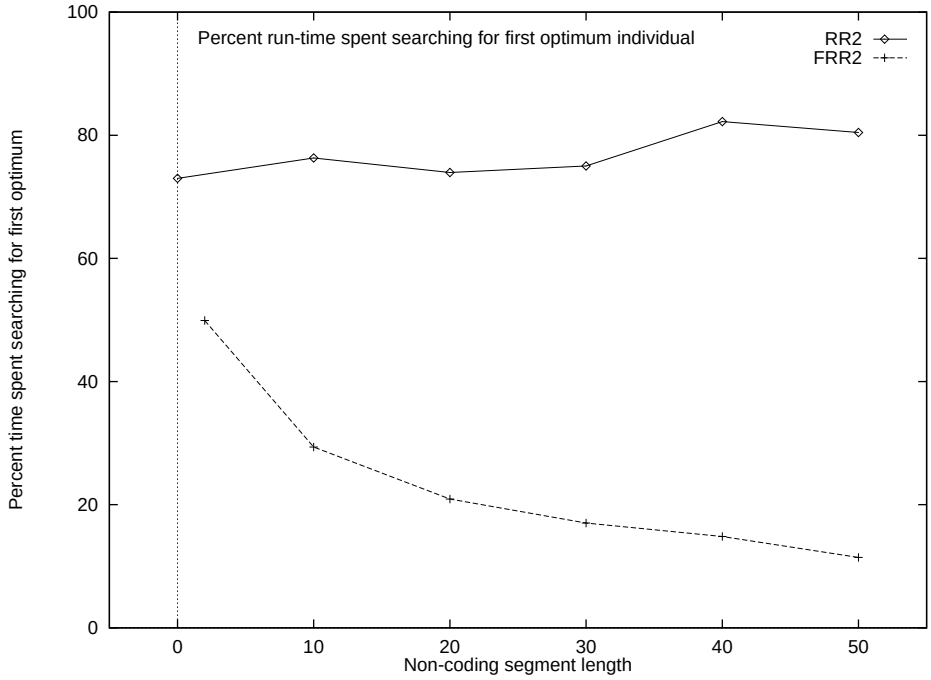

Figure 12: Comparison of the percentage of the run time spent searching for the first optimum individual in the `8x8.RR2` and `8x8.FRR2` functions. The stopping condition of the runs is 50% optima in the population.

meet the goal. With the floating representation, 50% or less of the time of an average run is spent searching for the first optimum. The rest of the time is spent filling the population. As non-coding segment lengths increase, a GA spends less and less of the run time searching for the first optimum and more time filling up the population. We know that the discovery of one optimum individual in a floating representation run does not mean that all other individuals are only one building block away from the optimum. On the contrary, it is not uncommon to find different individuals with equal (including optimal) fitness in a floating representation population. As non-coding segment length, and hence genome length, increases, the time required for a GA to find one optimal individual decreases, giving it less time to propagate useful partial solutions throughout the population. As a result, with increasing genome length, the population with the first optimum will be more and more diverse and a GA will have to do more work to fill up the population to 50% optima.

9 Symbolic Regression

To test the effectiveness of the floating representation on a more realistic problem, we observed GA performance using both the fixed and floating representations on a symbolic regression problem. Give a set of data points or values, a GA is used to discover a set of numerical coefficients for a polynomial function that most closely fits the given points. The

solution to such problems can readily be represented as a set of building blocks: one building block per coefficient. Though the maximum degree of the polynomial (and consequently, the number of terms to be summed) must be specified in advance in our algorithm, the actual number of terms in each candidate solution may vary. To make the problem more challenging, sine and cosine terms are also included in the function. The goal of the GA is to minimize the sum of the differences between each data point and the corresponding function value.

The experiments to be described here attempt to fit a function to the set of data points given by the function

$$g(x) = 2\sin(x) + 1 - 2x$$

for values of x in the range $0 \leq x \leq 20$. Given a maximum second degree polynomial, the function that the GA attempts to fit to this data is

$$f(x) = a\cos(x) + b\sin(x) + c + dx + ex^2$$

where a, b, c, d, and e are the coefficients whose values are to be evolved. Coefficient values may be positive or negative integers or zero, allowing for the adaptation of both the values and the number of terms in the function (a zero coefficient effectively eliminates a term). Each individual of the GA represents a candidate function. The fitness of an individual is inversely proportional to

$$E = \sum_{x=0}^{20} |g(x) - f(x)|$$

where E is the sum of the differences between each given data point and the corresponding value of the candidate function. Lower values of E indicate less error, which results in higher fitness.

The parameter values from the beginning of section 8 are used in these experiments with the exception of population size which is set to 100 here. Genome lengths ranging from 70 to 220 are tested. Five building blocks each four bits long are used to represent the five coefficients from $f(x)$. Coefficient values must be integers and are coded in binary format. With the fixed representation, building blocks are assigned to specific locations on the individuals and there may be zero or more non-coding bits between building blocks. With the floating representation, each building block is assigned a unique, randomly generated tag which, when found on an individual, precedes the actual building block value. The issues of over-specification and under-specification of floating building blocks and the tag length of building blocks are research topics in themselves, and will not be extensively studied here due to time constraints. In these experiments, building block tags are six bits long; when there are multiple building blocks for a single coefficient on an individual, the average integer value of all building blocks is used; and when a coefficient is missing on an individual, its value is zero by default.

Each experiment is allowed to run for 200 generations and the *online error* and *offline error* are recorded. Given the best individual from each generation of a run and their error values, the online error is the average of these error values and the offline error is the minimum error value (the single best individual) of the entire run. An offline error of zero indicates that a perfect solution has been found, that is, $f(x) = g(x)$. Figure 13 plots the

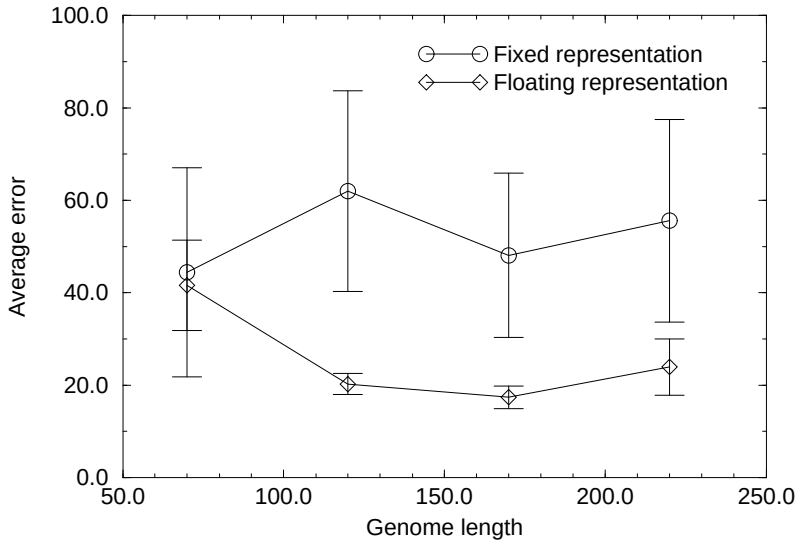

Figure 13: Average online error as genome length increases.

average online error as genome length increases. Figure 14 plots the average offline error as genome length increases. In both plots, each data point is the average value from ten runs. Once again, results indicate that GA performance is better on functions that use the floating representation that on functions that use the fixed representation. In fact, figure 13 looks similar to the plots comparing the fixed and floating RR functions: GA performance on the fixed representation does not change significantly while GA performance on the floating representation improves with increasing genome length. In examining individual runs, we found that once again, the GA has a tendency towards early convergence with the fixed representation functions. While many of the fixed representation runs had online and offline errors that were comparable to the floating representation values, a few of the fixed representation runs converged too early on coefficient values with high error rates and became stuck at those values, resulting in unusually high online and offline errors for those particular runs. The dynamic nature of the floating representation makes it more difficult for a GA to converge prematurely giving it more time to lower the error rates of its candidate solutions. While both the floating and fixed runs were able to find "perfect fits", figure 14 indicates that a GA is much more likely to find an ideal solution using the floating representation than the fixed representation.

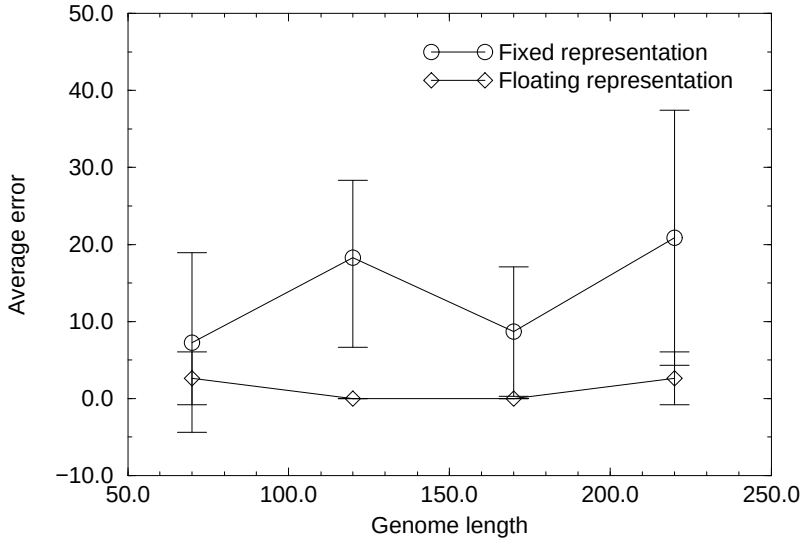

Figure 14: Average offline error as genome length increases.

10 Summary

The experiments described in this article investigate the effects non-coding segments and location independent building blocks on a GA. The performance of a GA on the fixed and floating representations of the RR functions and of a symbolic regression problem are compared. The results suggest that there is a symbiotic relationship between the floating building block representation and non-coding segments and encourage the use of floating building blocks in the GA. With shorter individuals, GA performance tends to be better on the fixed representation than on the floating representation. As the genome length of individuals increases, GA performance on the floating representation improves and becomes noticeably better than the performance on the fixed representation

Analysis of GA performance on the RR functions indicate that differences in performance are due in part to the demands on a GA as it solves each type of problem. With the fixed representation, a GA needs only to find one optimal copy of each building block. As genome length increases, the stability of a GA improves, but the advantage in speed gained from this stability is cancelled out by the reduced exploration within building blocks. With the floating representation, a GA must find optimal building blocks and also arrange the building blocks on the individuals. Arrangement can be difficult at short genome lengths, trivial at long genome lengths, as reflected in our results. Analysis of individual runs reveals differences in the problem solving procedure as well. With the fixed representation, a GA finds building blocks one at a time and quickly propagates new building blocks throughout

the entire population. With the floating representation, a GA maintains a more diverse population which includes a moderate number of copies of each basic building block. As a result, a GA can spend more energy in combining various existing building blocks rather than searching for new ones. These observations support the importance that the building block hypothesis places on a GA's parallel search and recombination abilities. Parallel search for different building blocks results in a more diverse population which, in turn, allows a GA to take full advantage of crossover's recombination abilities.

Because the fixed representation runs are so good at filling the population with each new building block, most of the population at any given time contains the same combination of building blocks. As a result, if a GA run is extended past the first optimum individual, the amount of time required to fill the population, for example to 50 percent optima, after the first optimum is found is minimal compared to the amount of time spent looking for the first optimum. The floating representation runs, on the other hand, maintain a more diverse population throughout the entire run. Consequently, the amount of time spent filling the population to 50 percent optima after finding the first optimum is typically equal to or greater than the amount of time spent searching for the first optimum.

Initial experiments on a symbolic regression problem yield comparable results. Both the online and offline performance of a GA is better on the floating representation than on the fixed representation of this problem. The ability of the floating representation to maintain diversity is again an important factor.

The results of these experiments encourage the inclusion of non-coding segments and the use of the floating building block representation in a GA. The combination of non-coding segments and floating building blocks appears to be an excellent pairing for taking advantage of a GA's problem solving abilities. This seemingly symbiotic relationship creates a better balance of exploration and exploitation and results in increased stability of building blocks, and greater diversity of the population.

Acknowledgements

This research was funded by NASA Johnson Space Center under grant NGT-51057 and by the National Research Council Research Associateship Program and the Naval Research Laboratory. The authors would like to thank John Holland for many interesting discussions and suggestions relating to this work and the reviewers of this article for many helpful comments.

References

Aho, A. V. and M. J. Corasick (1975). Efficient string matching: an aid to bibliographic search. *Communications of the ACM 18*(6), 333–340.

Bell, G. I. and T. G. Marr (Eds.) (1988). *Computers and DNA*. Addison-Wesley.

Blake, C. C. F. (1978, May). Do genes-in-pieces imply proteins-in-pieces? *Nature 273*, 267.

Curtis, H. (1983). *Biology*. New York: Worth Publishers.

Forrest, S. and M. Mitchell (1993). Relative building-block fitness and the building-block hypothesis. In D. Whitley (Ed.), *Proceedings of the Foundations of Genetic Algorithms Workshop*, San Mateo, CA, pp. 109–126. Morgan Kaufmann.

Gilbert, W. (1978, February). Why genes in pieces? *Nature 271*, 501.

Gilbert, W. (1985, May). Genes-in-pieces revisited. *Science 228*, 823–824.

Gilbert, W. (1987). The exon theory of genes. *Cold Spring Harbor Symposia on Quantitative Biology 52*, 901–905.

Gilbert, W. (1991). Gene structure and evolutionary theory. In L. Warren and H. Koprowski (Eds.), *New perspectives on evolution*, New York, pp. 155–163. Wiley-Liss.

Go, M. (1981, May). Correlation of DNA exonic regions with protein structural units in haemoglobin. *Nature 291*, 90–92.

Goldberg, D. E., B. Korb, and K. Deb (1989). Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems 3*, 493–530.

Hartl, D. L. (1991). New perspectives on the molecular evolution of genes and genomes. In L. Warren and H. Koprowski (Eds.), *New perspectives on evolution*, New York, pp. 123–137. Wiley-Liss.

Haynes, T. (1996). Duplication of coding segments in genetic programming. In *Proceedings of the 13th National Conference on Artificial Intelligence*, Cambridge, MA, pp. 344–349. MIT Press.

Jones, T. (1994). A description of holland's royal road function. *Evolutionary Computation 2*(4), 409–415.

Levenick, J. R. (1991). Inserting introns improves genetic algorithm success rate: Taking a cue from biology. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pp. 123–127.

Lewin, B. (1994). *Genes 5*. John Wiley & Sons.

Mitchell, M., S. Forrest, and J. H. Holland (1992). The royal road for genetic algorithms: Fitness landscapes and GA performance. In F. J. Varela and P. Bourgine (Eds.), *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, Cambridge, MA. MIT Press.

Nei, M. (1987). *Molecular Evolutionary Genetics*. Columbia University Press.

Nordin, P., F. Francone, and W. Banzhaf (1995). Explicitly defined introns and destructive crossover in genetic programming. In P. Angeline and K. Kinnear (Eds.), *Advances in Genetic Programming II*, pp. 111–134. Cambridge, MA: MIT Press.

Wu, A. S. and R. K. Lindsay (1995). Empirical studies of the genetic algorithm with non-coding segments. *Evolutionary Computation 3*(2), 121–147.

Wu, A. S. and R. K. Lindsay (1996). A survey of intron research in genetics. In *Proceedings of the 4th International Conference on Parallel Problem Solving from Nature*.