Empirical studies of the genetic algorithm with non-coding segments

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

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

To appear in *Evolutionary Computation*, 3:2, 1995.

Abstract

The genetic algorithm (GA) is a problem solving method that is modelled after the process of natural selection. We are interested in studying a specific aspect of the GA: the effect of non-coding segments on GA performance. Non-coding segments are segments of bits in an individual that provide no contribution, positive or negative, to the fitness of that individual. Previous research on non-coding segments suggests that including these structures in the GA may improve GA performance. Understanding when and why this improvement occurs will help us to use the GA to its full potential. In this article, we discuss our hypotheses on non-coding segments and describe the results of our experiments. The experiments may be separated into two categories: testing our program on problems from previous related studies, and testing new hypotheses on the effect of non-coding segments.

Keywords: genetic algorithms, non-coding segments, non-coding DNA, introns, Royal Road function.

1 Introduction

Nature has successfully used a simple, elegant selection method for millions of years. Natural selection sustains healthy populations because fitter individuals have a greater chance of survival and reproduction. Reproduction then propagates the characteristics of the fitter individuals to the next generation. The *genetic algorithm* (GA) is a computer model of the process of natural selection. The GA has become increasingly popular as a non-biological engineering tool and has been applied successfully to many different types of problems.

The GA operates on a population of individuals or chromosomes. Each individual represents one possible solution to the problem to be solved. A *fitness function* evaluates each individual and returns a value that indicates how well that particular solution solves the problem. The GA exploits individuals with high relative fitness by giving them the chance to have more offspring than less fit individuals. As a result, the better individuals of a generation have a greater chance of propagating their building blocks into the next generation. *Building blocks* are specific bit patterns that contribute to high fitness values. The crossover and mutation operators, which are modelled after biological processes, are used to explore the solution space. Crossover randomly chooses a breakpoint at which two individuals are cut and the two end segments are exchanged. Mutation changes the value of individual bits. The strength of the GA is believed to be its ability to find good solutions to complex problems without needing to test every possible solution. This ability is especially valuable when solving combinatorially explosive problems. The GA achieves this ability by balancing the exploratory and exploitative tendencies described above. According to the *building block hypothesis* (Holland, 1975), the GA finds solutions by searching for building blocks that have high relative fitness and combining them to form even larger building blocks, eventually forming a complete solution.

We would like to extend the analogy between GAs and genetics by studying the effects of another aspect of genetics on the GA's problem solving ability, specifically, non-coding DNA. In natural systems, deoxyribonucleic acid (DNA) is the genetic material that is propagated from generation to generation. The majority of the DNA in an organism, however, is non-coding DNA, DNA that does not code for the ribonucleic acid (RNA) necessary for synthesizing proteins. Though the function of non-coding DNA is still unknown, these structures must not contribute negatively to the genetic process or they would most likely have been eliminated by natural selection long ago. Just as the GA models the process of natural selection, non-coding segments[1]— excess bits — can be added to the individuals of a GA to model non-coding DNA.

Previous studies on non-coding segments have suggested that including these

[1]In this paper, the term *non-coding segment* replaces the term *intron* which has been used in the past (see Section 2.1).

segments in the GA population may speed up the GA's climb from initial, randomly chosen solutions to good, high-fitness solutions. We believe that non-coding segments may also have a stabilizing effect on the preservation of building blocks or partial solutions by the GA. A better understanding of the contributions of non-coding segments would help us to use GAs more efficiently and effectively. We have performed a detailed study on non-coding segments that both replicates previous research and extends it to new experiments.

2 Background

2.1 Non-coding DNA

It is useful to understand the biological inspiration for this work. A *genome* is the complete set of genetic material of a cell, and is composed of several discrete units called *chromosomes*. Chromosomes are sequences of DNA nucleotides. A *gene* is a segment of DNA that codes for a protein, for RNA, or for a regulator product. The DNA of a gene is copied into messenger RNA (mRNA) inside the nucleus of a cell. The mRNA is released from the nucleus into the cytoplasm after preprocessing. In the cytoplasm, transfer RNA (tRNA) match up to the mRNA, three nucleotides at a time. Each tRNA carries an amino acid. The final product is a chain of amino acids whose order is determined by the nucleotide sequence of the mRNA. The synthesis of RNA from a DNA template is called *transcription*. The synthesis of a protein from the mRNA template is called *translation*. A *genotype* is the genetic constitution of an organism. A *phenotype* is the appearance or other characteristics of an organism, resulting from the interaction of its genotype with the environment. A gene that is *expressed* makes some contribution to the phenotype (Curtis, 1983) (Lewin, 1994).

There are several types of non-coding DNA (Nei, 1987). *Intergenic regions* and *intragenic regions* (introns) make up a large part of the non-coding DNA. Intergenic regions are found between genes and are not transcribed into mRNA. Some portions of the intergenic regions regulate the expression of adjacent genes; other portions have no known function. Intragenic regions, also called *introns*, are one or more segments of DNA found within genes (Lewin, 1994). After a gene is transcribed into mRNA, the intron regions are removed from the mRNA chain, and the remaining segments of mRNA ,the *exon* regions, are joined together to become the protein template. Though we are starting to understand the intron removing mechanism (Patrusky, 1992), the function of introns is still uncertain.

A third type of non-coding DNA is the *pseudogene*. A pseudogene is a segment of DNA that is similar to a functional gene, but contains nucleotide changes that prevent its expression (Nei, 1987). Pseudogenes are believed to arise from gene duplication or reverse RNA transcription. Interestingly, pseudogenes produced from reverse transcription do not contain introns. Since pseudogenes are not expressed, they are not subject to selection pressure from the environment. As a result, pseudogenes 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 Schemata

Since building blocks are an intrinsic part of the GA, it is important to have a clear definition of them. Typical GA individuals are strings of binary bits. Building blocks are subsets of these bits that work together, i.e. specific patterns of these bits may result in a significant increase in fitness. Building blocks are defined by their encompassing schemata. A *schema* refers to the set of all strings that contain a particular building block (Forrest & Mitchell, 1993) (Goldberg, 1989). The GA operates on individuals that are strings of length l, defined over the set $\{0, 1\}$. The schemata are strings, also of length l, defined over the set $\{0, 1, *\}$, where $*$ is the *don't care* character. The *order* of a schema is the number of defined bits in the schema. The schema $11 * 01*$ has order, $r = 4$. Since either a 1 or a 0 may be substituted for a $*$, each schema is a set of 2^{l-r} chromosomes. The schema $110 * *$ includes $2^{5-3} = 2^2 = 4$ chromosomes. These chromosomes are $11000, 11001, 11010, 11011$. The *defining length*, $\delta(s)$, of a schema s is the distance between its first and last defined bit. For example, $\delta(*1 * **) = 0$, $\delta(* * 10*) = 1$, and $\delta(*1 * *1) = 3$.

2.3 Other approaches to studying and improving the GA

There have been many attempts to understand how the GA works and to improve the performance of the GA. The hope is that better understanding of the GA will lead to more effective use of this problem solving tool. Past and current research on the GA have addressed topics such as control parameter values, genetic operators, and problem representations.

Much research has gone into searching for control parameter values that are optimal or near optimal for all types of problems. Typical control parameters include crossover rate, mutation rate, population size, and reproduction rate. De Jong (1975) and Grefenstette (1986) both conducted empirical studies on the effect of control parameters on GA performance. The results from these studies have been used as guidelines for initial parameter settings for many GA systems and have spurred further research on control parameter effects. Additional studies (Bäck, 1991) (Lee & Takagi, 1993) (Schaffer & Morishima, 1987) have attempted, with some success, dynamic control of parameter values either by the GA itself or other automated regulator systems. While there are several sets of good "starting point" control parameter values, these sets are not universally optimal and fine tuning of these control parameters to the specific problem is still necessary.

Another area of study that has improved our understanding of the GA is the study of genetic operators. Crossover and mutation are the most commonly studied operators. Many different types of crossover have been used with the GA, ranging from the basic one-point crossover to uniform crossover. Though theory predicts that one-point and two-point crossover should perform the best (Syswerda, 1989), experimental results have shown that multi-point crossovers perform significantly better. Other studies have concentrated on the effects of mutation (Fogarty, 1989) (Bäck, 1991); both static and dynamic mutation rates have been tested. Eshelman and Schaffer debated the usefulness of crossover versus mutation (Eshelman & Schaffer, 1993) (Schaffer & Eshelman, 1991). The results indicate that a combination of crossover and mutation is generally more effective than either crossover or mutation alone. While crossover is a good exploratory operator, it is "less effective than mutation at maintaining gene pool variation in the face of selection pressure" (Schaffer & Eshelman, 1991) and thus makes the GA prone to early convergence when used alone. Mutation, which has often been considered a secondary operator, performed surprisingly well on its own because of its ability to introduce variation into the population.

Instead of changing the parameters, operators, or other parts of the GA, studies have also looked into alternative problem representations in the GA. The typical method of representing a problem solution in the GA is to separate the bits of an individual into groups. Each group then codes for one part of the total solution. While this method is both simple for the user to set up and easy for the fitness function to evaluate, it may not always produce the best results. The user may not be aware of the best arrangement of problem parts for creating good building blocks and there are some problems which are simply not suited for the typical GA representation. Goldberg *et al.* (1989) (1990) studies the placement of bits with the messy GA (mGA). By representing each bit of an individual as a pair of numbers – the name and the value of the bit – the mGA is able to arrange the relative placement of bits dynamically during a run. This gives the mGA the flexibility to test and rearrange bits into groups that make good building blocks. Levenick (1991) and Forrest and Mitchell (1992) also investigate the placement of information on an individual. Their work will be discussed in detail in section 5. There are some problems which seem to be good candidate problems for the GA to solve that simply do not work well with the typical binary string representation. GA users often need to tailor problem representations to their specific problems. For example, scheduling problems (Oliver, Smith & Holland, 1987) often use numerical string representations to avoid generating illegal solutions. Though the new representations may require the modification of some steps or operators in the

GA, the basic GA principles of selection and reproduction remain intact.

3 Motivation

Though the function of their biological counterparts is still uncertain, there are a number of arguments for studying the effect of non-coding segments on the GA. The most compelling of these arguments is that non-coding DNA has existed in nature for millions of years. In addition, previous studies on non-coding segments suggest that including these structures in a GA may be beneficial to the GA. Levenick's (1991) research shows that including non-coding segments in the GA may improve the GA's ability to find the optimum individual within a given number of function evaluations.

We believe that non-coding segments may also have a stabilizing effect on the preservation of building blocks by the GA. A key ingredient to using the GA effectively is balancing the exploratory and exploitative tendencies of the GA (Eshelman, Caruana & Schaffer, 1989). The GA explores the solution space to find new building blocks and to combine existing building blocks into larger units. The GA exploits highly fit individuals to preserve their presumably highly fit building blocks in the population. Excessive exploration causes the GA to destroy existing building blocks and excessive exploitation results in early convergence. Thus, too much of either tendency cancels out the effect of the other. A stable GA must maintain a good balance of exploration and exploitation.

There are a number of statistical arguments for including non-coding segments in a GA. These arguments include the following hypotheses.

- Non-coding segments encourage the genetic recombination of existing building blocks. Because non-coding segments increase the number of possible crossover locations between building blocks, they also increase the probability that crossover will occur in between building blocks rather than within a building block. It is important to distinguish between crossover rate and the chance of crossover at any location on an individual. Crossover rate is a control parameter that refers to the chance that an individual will undergo crossover. Once it has been decided that an individual will undergo crossover, a crossover location must be chosen. The chance that crossover will occur at any location on an individual is typically the same for all locations. As a result, adding non-coding bits in between the building blocks increases the chance of crossover occurring in between building blocks which improves the recombination rate of building blocks.

  This increase in the chance of crossover in between building blocks suggests that adding non-coding segments to an individual is similar to using variable crossover probabilities — where the chance of crossover occurring at any given location on an individual differs from location to location — on an individual without non-coding segments. The advantages of the non-coding segment method include the fact that the GA does not need to be modified to handle variable crossover probabilities and that crossover location calculations are much simpler. In addition, the variable crossover probabilities are easily changed or removed by adding or deleting non-coding bits.

- Non-coding segments may reduce the hitchhiking effect (Forrest & Mitchell, 1992). The *hitchhiking effect* occurs when non-optimal bits in the *don't care* region of highly fit schemata benefit from the attached highly fit schemata. As the non-optimal schemata increase in the population, courtesy of the attached highly fit schemata, the non-optimal schemata may slow down the discovery of or wipe out existing lower order building blocks at their same location on the individual. Because of the genetic operators used in the GA, in particular crossover, the hitchhiking effect is expected to be stronger in those parts of the individual that are physically close to the highly fit schemata. Inserting non-coding segments in the individual separates the building blocks and moves them farther apart from each other. Thus, hitchhiking is more likely to be contained in the non-coding segments adjacent to highly fit schemata, and less likely to impede the discovery of neighboring building blocks.

- Non-coding segments maintain variation in the individual. Variation is necessary for exploration of the search space. Since the mutation rate is the probability that each bit of an individual will be complemented, the expected total number of mutations on an individual is equal to the length of the individual times the mutation rate. Mutation as a discovery tool for the coding regions is not altered by the addition of non-coding segments because the lengths of the coding regions are not changed. Nevertheless, non-coding regions are also subject to mutation. Because of lack of selection pressure, mutations should accumulate easily in the non-coding segments. This variation could be very valuable in some situations, such as in experiments where the building block locations are not pre-specified and in experiments where the fitness function is not constant (the environment changes). Should a non-coding segment ever become a building block, variation in that region in the population will prevent early convergence to a sub-optimal solution at that location.

- Non-coding segments could reduce the chance of crossover breaking up an existing building block. This depends on our definition of a simple individual (an individual without non-coding segments). Figure 1 compares two versions of a simple individual with an individual that has non-coding bits. Levenick (1991) places additional extraneous bits at the end of the significant bits so that the simple individual has the same length as the non-coding individual. Forrest and

```
A = aaaaaabbbbbbccccccddddddeeeeee;
B = aaaaaabbbbbbccccccddddddeeeeee******************************;
C = aaaaaa*******bbbbbb*******cccccc*******dddddd*******eeeeee**;
```

Figure 1: A comparison of individuals with and without non-coding segments. Letters indicate significant bits; * indicates non-coding bits. Each building block is represented by a different letter. C is an individual with non-coding segments from Levenick's experiments. B is an individual without non-coding segments from Levenick's experiments. A is an alternative individual without non-coding segments.

> Mitchell (1992) define the simple individual as an individual consisting of only significant bits.
>
> In the first case (`B`), the chance of crossover combining two adjacent building blocks is reduced in comparison to `C` because there is only one crossover point between each building block on `B`. Assuming traditional one-point crossover, the probability of crossover occurring at any location on the chromosome is 1/*(length of chromosome)*. This probability includes the "null" crossover — when the crossover location selected is at the end of the individual and, in effect, no crossover occurs. Since both `C` and `B` are the same length and have the same number of significant bits, the chance that a building block will be broken by crossover is the same. When using one-point crossover, the extra bits at the end of the individual simply increase the chance of the "null" crossover.
>
> In the second case (`A`), the chance of crossover combining two adjacent building blocks is also reduced. Again, this is because there is only one crossover point between each building block on `B`. Since the probability of crossover occurring at any bit depends on the length of the chromosome, and `A` is shorter than `C`, the chance that a building block will be broken by crossover on `A` is greater than on `C`. We expect `C` to lose (and rediscover) fewer building blocks in the process of finding the optimum chromosome.

4 Experimental design

4.1 What are we testing?

- Does including non-coding segments improve the performance of the GA?
- Do non-coding segments combat the hitchhiking effect?
- Do non-coding segments stabilize the GA?

- Under what conditions (what fitness landscapes) do the non-coding segments provide the most improvement?
- What is the best placement of the non-coding segments or of the building blocks?
- Can we confirm previously reported beneficial effects of non-coding segments?

4.2 Program/system details

Though the steps of a typical GA are well known, implementation details vary from system to system. In replicating previous research, we found that small differences in program structure can produce significant changes in the results. Thus, to successfully replicate a GA experiment, we need to know all of the implementation details for that experiment.

The following is a description of the steps of our GA program. Any changes that were necessary for our replication experiments will be specified in the appropriate sections.

1. Create initial population (generation 0). The individuals of the initial population are generated randomly. Each bit in each individual is equally likely to be initialized to a 1 or a 0.
2. Evaluate the fitness of each individual in the current generation. Also, calculate the average fitness and standard deviation of the fitness of the current population.
3. If the stopping condition is satisfied, terminate the program; otherwise, continue. The stopping condition may be a minimum fitness value to be achieved or a maximum number of generations to run the GA.
4. Calculate the expected number of offspring for each individual in the current generation. This value will be proportional to the fitness of the individual: more fit individuals are expected to have more offspring. The expected number of offspring is scaled to a real value x where $0 \leq x \leq 1.5$. In practice, this means that the expected number of offspring will range from zero to two.
5. Select the parents of the next generation. An individual may be selected more than once; multiple selections simply indicate that the individual will sire multiple offspring. Parents are selected using the *roulette wheel* selection method. To select a parent:
    - Choose a random real number $n : 0 \leq n \leq \mathit{population\_size} - 1$.

- Starting from the first individual in the current generation and iterating through as many individuals as necessary, sum the expected number of offspring for each individual. Stop when the sum becomes greater than n. The individual which stops the iteration is selected as the parent.

The number of parents selected should be equal to the number of new offspring expected in the next generation (remember that an individual may be selected to be a parent more than once). Thus, the number of parents selected depends on the reproduction rate. For 100% reproduction rate, there should be *population_size* number of parents. For lesser reproduction rates, the number of parents will be some fraction of the total population size. The remaining fraction of the next generation will consist of individuals that survive the current generation. An individual that *survives* from one generation to the next is copied from the current generation to the next generation with no change. In this algorithm, the survivors of the current generation are the most fit individuals of the current generation.

6. Reproduce. Starting from the top of the list of parents, pairs of individuals undergo one-point crossover and mutation to produce offspring. The crossover rate is the probability that any pair will undergo crossover and the mutation rate is the probability that a single bit will be complemented. The offspring together with the survivors of the current generation form the next generation of individuals.

7. Go to step 2.

The program that we are using is algorithmically the same as the program used in (Forrest & Mitchell, 1992).

4.3 Testbed

With the exception of the replication experiments, we ran all of our experiments on the Royal Road functions. The Royal Road functions (Forrest & Mitchell, 1992) (Forrest & Mitchell, 1993) (Mitchell & Holland, 1993) are a class of fitness landscapes designed for studying building block interactions. Royal Road functions are defined by their component schemata. The building blocks of a Royal Road function are specified by schemata and assigned fitness values before a run begins. Higher level building blocks consist of two or more lower level building blocks. An example of a Royal Road function from (Forrest & Mitchell, 1992), $RR2_{exponential}$, is shown in Figure 2.

Pre-defining all the building blocks of a function allows us to trace the GA's progress in combining short building blocks to form larger building blocks to eventually

s_1 = 11111111**; $c_1 = 8, l_1 = 0$
s_2 = ********11111111**; $c_2 = 8, l_2 = 0$
s_3 = ****************11111111**; $c_3 = 8, l_3 = 0$
s_4 = ************************11111111********************************; $c_4 = 8, l_4 = 0$
s_5 = ********************************11111111************************; $c_5 = 8, l_5 = 0$
s_6 = **11111111****************; $c_6 = 8, l_6 = 0$
s_7 = **11111111********; $c_7 = 8, l_7 = 0$
s_8 = **11111111; $c_8 = 8, l_8 = 0$
s_9 = 1111111111111111**; $c_9 = 16, l_9 = 1$
s_{10} = ****************1111111111111111********************************; $c_{10} = 16, l_{10} = 1$
s_{11} = ********************************1111111111111111****************; $c_{11} = 16, l_{11} = 1$
s_{12} = **1111111111111111; $c_{12} = 16, l_{12} = 1$
s_{13} = 11111111111111111111111111111111********************************; $c_{13} = 32, l_{13} = 2$
s_{14} = ********************************11111111111111111111111111111111; $c_{14} = 32, l_{14} = 2$

s_{opt} = 1111111111111111111111111111111111111111111111111111111111111111;

Figure 2: The Royal Road function, $RR2_{exponential}$. The fitness of x, an input individual, is computed by summing the coefficients or fitnesses, c_i, corresponding to each schemata s_i of which x is an instance. The subscript *exponential* refers to the fact that the coefficients, c_i, increase exponentially. In this function, $c_i = order(s_i)$ and l_i indicates the level. $RR2_{exponential}$ has two levels of intermediate building blocks between the basic building blocks and the complete optimum individual.

```
11111111 11111111 11111111 11111111 11111111 11111111 11111111
** 11111111 **** 11111111 **** 11111111 **** 11111111 **** 11111111 **** 11111111 **** 11111111 **
```

Figure 3: Examples of $RR2_{exponential}$ individuals without and with non-coding segments. The "$*$" represents a non-coding bit.

finding the optimum. We can easily modify the fitness landscape by changing either the building block definitions or the fitness values of the Royal Road function. For example, Forrest and Mitchell also test a Royal Road function called $RR1_{exponential}$, which is $RR2_{exponential}$ minus schemata s_9 thru s_{14}.

Non-coding segments are added to the Royal Road functions by inserting additional bits in between the basic building blocks. For all experiments described in this paper, the non-coding bits were distributed evenly among the basic building blocks. That is, the same number of extra bits were inserted in between each pair of basic building blocks. This number is the *non-coding segment length.* Figure 3 shows an example of $RR2_{exponential}$ individuals with and without non-coding bits, where "*" represents a non-coding bit. Since $RR2_{exponential}$ has eight basic building blocks, eight segments of non-coding bits are added (one segment is split and placed at the ends of the individual). Thus, for a non-coding segment length of 4, the length of the individual increases by (8 building blocks) x (4 bit long segments) = 32 bits.

4.4 Performance criteria

The most commonly used performance measure for evaluating GA performance is the number of generations or function evaluations required before the GA finds an acceptable solution to the problem. In the worst cases, this measure becomes "does the GA find a solution at all?" The obvious goal is to minimize the number of generations or function evaluations required to find the solution since this also minimizes the time and computing power used. This measure, however, is not the only way to evaluate the performance of a GA, and does not necessarily paint a complete picture of the GA's abilities. Other performance measures that we will look at include the number of generations the GA takes to move up each level of the Royal Road function and the number of times building blocks are found (discovered or rediscovered after being lost).

Observing the number of generations between levels will provide insight on how the GA progresses when solving a problem. Does the GA find building blocks in evenly spaced intervals of time or are there periods when the GA is more productive? The hierarchical structure of the Royal Road functions makes them ideal for studying this topic. Each level brings the GA closer to the optimum. In our experiments, we record the first generation that a building block from each level is found.

The number of times building blocks are found gives some indication of the stability of the GA. The definition of *found* includes the first time a building block is discovered in the population and any later re-discoveries of the building block if it is lost from the population. A stable GA is one which saves, in the population, the building blocks that have been found, while encouraging search for the remaining building blocks. The stability of the GA depends on the balance of exploration and exploitation. According to the building block hypothesis, the GA should save the good building blocks that it finds while continuing to search for other building blocks. Thus, we prefer the number of times each building block is found to be as close to one as possible. Let $s_i, i = 1, 2, ..., num\_building\_blocks$, equal the number of times building block i is found. Then, for each run, we record

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

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

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. A *basic building block* is a building block from level 0 of the Royal Road function. Basic building blocks are the smallest building block units that contribute to the fitness.

Building blocks occurring	Fitness value
none	1
a or b or c or d or e	5
d and e	50
a and b and c	100
a and b and c and d and e	1000

Table 1: Levenick's fitness function. The value of the best (highest value) building block in each individual provides the individual's fitness worth.

5 Experimental results

5.1 Levenick

Levenick (1991) presents a set of experiments that compare the performance of the GA on individuals whose building blocks are and are not separated by segments of non-coding bits. His results indicate that separating building blocks with segments of non-coding bits can make the GA more effective. The problem to be solved has five building blocks with four levels of building block interaction. The individuals and their component building blocks are shown in Figure 1. Levenick compares the GA's performance on type B and C individuals in his experiments. We decided to add type A individuals to the comparison as well because A truly has no excess bits and should be computationally quicker to process. The fitness function is shown in Table 1. The following parameter settings were used for these runs.

Number of significant bits:	30
Number basic building blocks:	5 (each 6 bits long)
Number total building blocks:	8
Crossover rate:	1.0 (one point)
Mutation rate (per bit):	0.0003
Genome length:	A = 30, B = 60, C = 60
Stopping criterion:	20000 function evaluations

Each experiment was run 50 times, and the number of successes (number of times the optimum was found) was recorded. Levenick varied both the population size and the reproduction rate. The reproduction rate indicates how much of each generation was allowed to generate offspring for the next generation. We modified step 5 of our program to add the capability for variable reproduction rate.

	Population size											
Reproduction		16			64			256			1024	
rate	A	B	C	A	B	C	A	B	C	A	B	C
Number of expected offspring = $x : 0 < x < 2.0$												
10%	0	0	0	3	2	1	13	2	22	0	0	0
30%	0	0	0	3	0	4	22	7	23	17	3	36
50%	0	0	0	2	0	7	24	12	33	41	30	49
70%	0	0	0	0	4	6	23	15	30	48	31	50
90%	0	0	0	2	2	15	30	18	30	37	29	44
100%	0	0	0	5	1	3	22	15	24	16	20	40
Number of expected offspring = $x : 0 < x < \mathit{population\_size}$												
10%	0	0	0	4	1	0	5	1	13	8	1	5
30%	0	0	0	2	0	0	11	3	19	17	9	22
50%	0	0	0	1	1	2	11	6	17	19	12	30
70%	0	1	0	5	1	5	21	4	22	26	16	36
90%	0	0	0	2	4	7	22	9	26	29	9	38
100%	0	0	0	8	3	3	16	4	23	18	18	37
Levenick's results (from [Levenick, 1991, Fig.3]):												
10%		0	0		0	2		3	22		4	48
30%		0	0		0	2		2	16		4	28
50%		0	0		0	1		1	9		3	23
70%		0	0		0	0		0	8		1	11
90%		0	0		0	1		0	4		1	12

Table 2: This table lists the number of times the optimum was found in 50 runs. For the runs in the top section, the number of expected offspring for each individual ranges from zero to two. For the runs in the middle section, the number of expected offspring for each individual ranges from zero to the population size. Levenick's results are shown in the bottom section. A, B, and C refer to the types of individuals used in the runs (see Figure 1).

We ran two sets of Levenick's experiments. In our initial attempt, each individual was limited to a maximum of two offspring. The number of expected offspring for each individual depends on the fitness of that individual, but this number was scaled to a value between 0 and 2. These results are shown in the first section in Table 2. Because of differences between our results and Levenick's, we re-ran the experiments without the extra scaling step. This time, the number of expected offspring was still proportional to the fitness of each individual, but ranged from 0 to the population size. The second set of results are shown in the second section in Table 2. Levenick's runs were all performed with unlimited, proportional number of offspring, like our second set.

Our data and Levenick's data are consistent on several major points. The success rate increases as the population size increases. This trend is probably due to the fact that larger populations provide a better sampling of individuals. The larger the population, the more partial or full building blocks the initial population is likely to have. The more building blocks there are, the easier the task is for the GA. In addition, we also confirm that inserting non-coding bits between building blocks improves the GA success rate when there is a limit on the amount of computation that can be made. The GA finds the optimum most often in the type C runs in all three sets of results. The type A runs generally perform better than the type B runs, but still perform worse than the type C runs.

Our data do not agree with Levenick's data on the relationship between success rate and reproduction rate. In Levenick's runs, the success rate decreases as the reproduction rate increases. In our runs, the success rate increases as the reproduction rate increases. According to (Levenick, 1991), the high reproduction rate runs performed poorly because the high rates increase the chance of premature convergence. Premature convergence occurs when all individuals in a population have evolved the same partial set of building blocks but none of the individuals have all of the optimal building blocks. When the fitness of an intermediate level building block is much better than that of the basic building blocks, the individuals containing this intermediate building block may take over the population, eliminating individuals that contain other useful, but less fit, building blocks. A similar effect occurred in our runs. GAs with higher reproduction rates are more likely to converge prematurely because convergence is faster with higher reproduction rates. We also found, however, that a number of other factors overcome or make use of this high convergence rate to benefit high reproduction rates. First, high reproduction rate seems to have more exploratory power. The population remains more diverse overall, even after the convergence period. After a population converges prematurely, the standard deviation of the population fitness at 90% reproduction rate is typically three to four times that at 10% reproduction rate. The GA is more likely to find the optimum during the convergence period for high reproduction rates than for low rates. This relationship between reproduction rate and exploratory power is consistent with the notion that genetic operators are the mechanisms for exploration in the GA and a high reproduction rate would result in more frequent usage of the genetic operators. In addition, since premature convergence occurs much faster in GAs with higher reproduction rates, the GA has more time to rediscover any building blocks that may have been lost during the convergence period. Once a building block exists in the entire population, then all exploratory power can be directed towards finding the remainder of the building blocks.

	RR1		RR2	
	F & M	Our runs	F & M	Our runs
Exponential, 500 runs				
Average	62099	66280	73563	76529
Std. dev.	31081	19484	40115	19066
Std. err.	1390	871	1794	853
Exponential, Population 1024, 200 runs				
Average	37453	47631	43213	66202
Std. dev.	12275	13432	18031	22917
Std. err.	868	950	1275	1620
Exponential, Building block length 4, Genome length 64, 200 runs				
Average	6568	12248	11202	20605
Std. dev.		6906		10007
Std. err.	198	488	394	708
Exponential, segment length 8, 200 runs				
Average		67478	75599	85720
Std. dev.		17124	38141	17123
Std. err.		1211	2697	1211
Flat, 200 runs				
Average		62454	62692	70388
Std. dev.		20770	33814	19349
Std. err.		1469	2391	1368

Table 3: The average number of function evaluations until the optimum is found, the standard deviation of this average, and the standard error of this average for Forrest and Mitchell's experiments. The number of runs performed for each experiment and any changes from standard parameter values are listed above the data for each experiment.

5.2 Forrest and Mitchell

Forrest and Mitchell (1992) performed a series of experiments on the Royal Road functions. Included in these experiments are several investigations into the effect of non-coding segments. We have replicated a number of their experiments. Table 3 lists Forrest and Mitchell's results in the first column and our results in the second column. The default parameters for the runs are the following. Any changes from these parameters are explicitly stated in the table.

Population size:	128
Significant bits:	64
Expected # of offspring:	0 to 2

Name	Building block fitness value
Exponential	*order(building block)*
Flat	1
Power	$base^{level+1}$

Table 4: Royal Road fitness landscapes that we tested with RR2.

Crossover rate:	0.7 (one point)
Mutation rate:	0.005 per bit

Like Forrest and Mitchell's program, our GA showed no improvement by including non-coding segments in the Royal Road function, $RR2_{exponential}$. In fact, the GA performs slightly worse when non-coding segments are included. We also tested $RR1_{exponential}$ with non-coding segments and obtained similar results.

The differences between our results and the original data range from small to approximately a factor of two. Though the results differ noticeably for some runs, the relative values of the runs are almost exactly the same. That is, if we were to order the runs by their performance (average length of time needed to find the optimum), the order of our experiments would be the same as the order of Forrest and Mitchell's experiments. Our program seems to magnify the differences for the extreme valued (shortest and longest) runs.

5.3 Our non-coding segment experiments

5.3.1 The effect of non-coding segments on different coefficient types

Our first set of experiments was performed on the Royal Road function RR2 (see Figure 2). We compared the effect of non-coding segments on four different coefficient types: exponential, flat, and power (bases 3 and 5). The coefficient type determines the contribution, c_i, of each building block to an individual's fitness. The coefficient types that we used are described in Table 4. For each coefficient type, we tested the GA with non-coding segment lengths ranging from 0 to 300 bits and evaluated the GA's performance on the criteria described in Section 4.4. The following parameter settings were used for these runs:

Population size:	128
Significant bits:	64 bits
Number of levels:	4

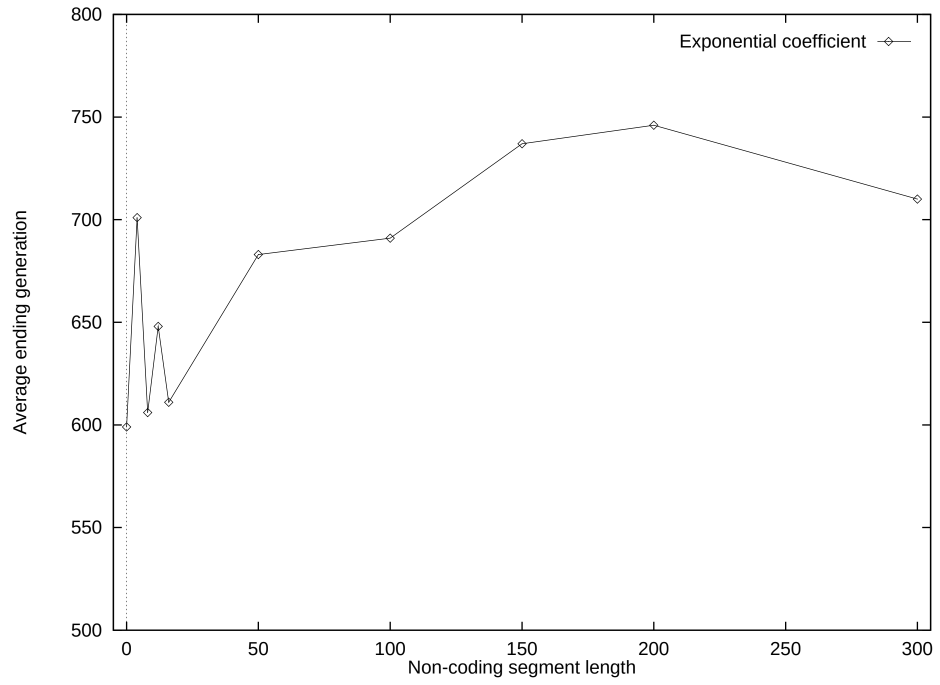

Figure 4: Data from the exponential coefficient runs: the average number of generations needed to find the optimum versus non-coding segment length.

Number of basic building blocks:	8, each 8 bits long
Number of total building blocks:	15
Crossover rate:	0.7 (one point)
Mutation rate (per bit):	0.005
Reproduction rate:	1.0
Parent selection:	Roulette wheel, with replacement
Expected number offspring per individual:	0 to 2
Stopping criterion:	5000 generations

Each experiment was run 100 times and the average values of these runs are reported here.

Figures 4 to 7 show the average number of generations the GA takes to find the optimum with exponential, flat, power(base = 5), and power(base = 3) coefficient types, respectively. A smaller number of generations until the optimum is found (shorter run times) indicates better performance. The exponential coefficient runs show no improvement from adding non-coding segments. On the contrary, the GA takes longer to find the optimum when these segments are included than when they are not. The flat coefficient runs also show little benefit from including non-coding segments, although short segments (less than 20 bits) do result in some improvement. The power coefficient, on the other hand, does result in shortened run times when

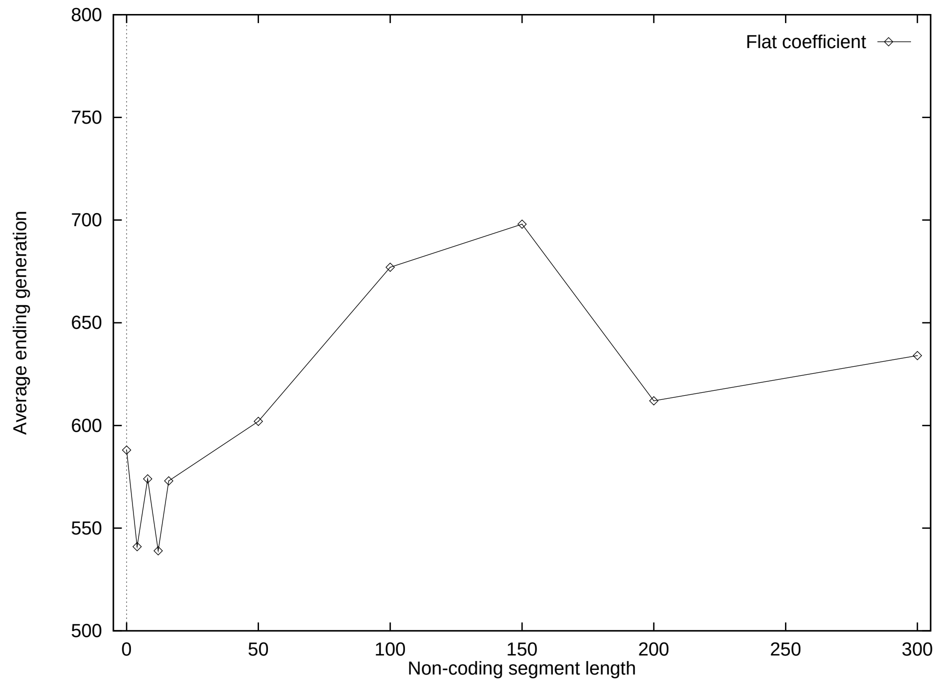

Figure 5: Data from the flat coefficient runs: the average number of generations needed to find the optimum versus non-coding segment length.

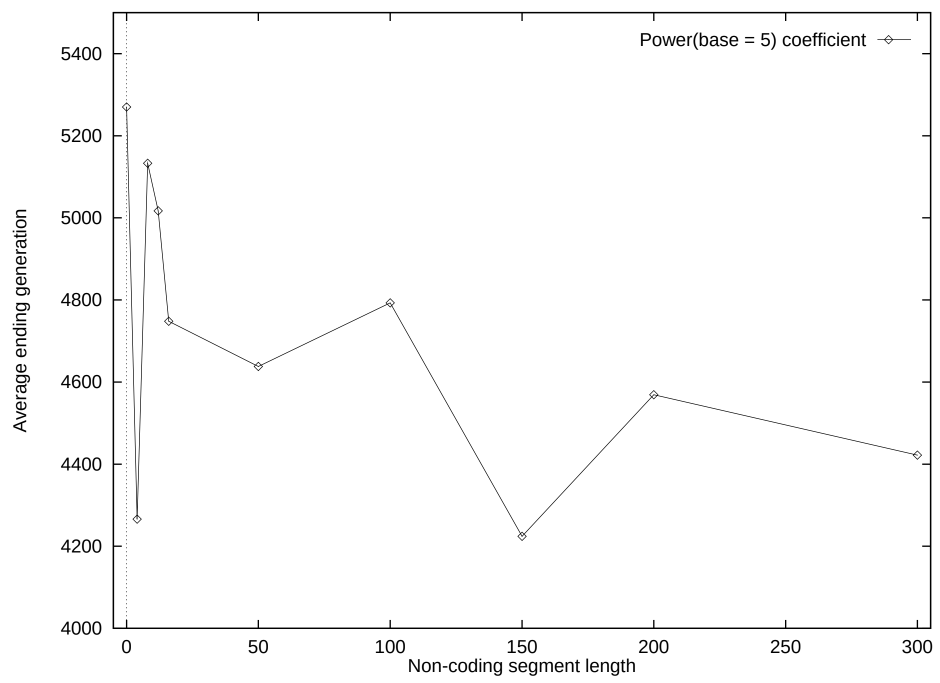

Figure 6: Data from the power(base = 5) coefficient runs: the average number of generations needed to find the optimum versus non-coding segment length.

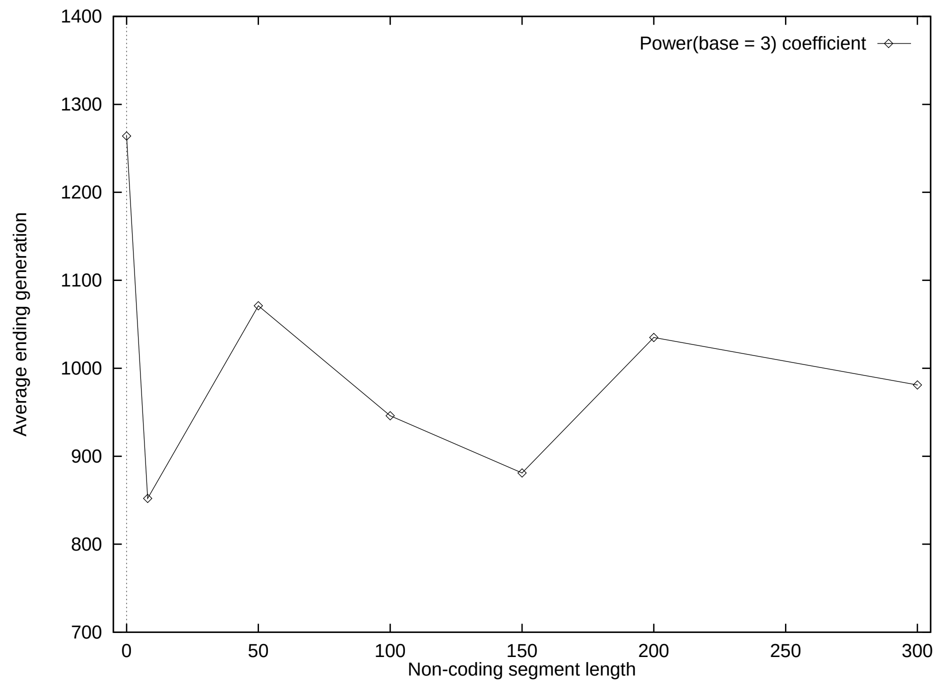

Figure 7: Data from the power(base = 3) coefficient runs: the average number of generations needed to find the optimum versus non-coding segment length.

non-coding segments are added. Though this improvement is not consistent (the number of generations does not always decrease as the segment lengths increase), the average number of generations for runs that include non-coding segments is always less than that for runs without these segments.

Figures 8 to 11 show building block stability data from exponential, flat, power(base 5), and power(base 3) coefficient types, respectively. S_{basic} refers to the sum of the number of times each basic building block is found. S_{all} refers to the sum of the number of times all of the building blocks are found. The vertical bars indicate the 95% confidence intervals for the average values. In all of our experiments, including non-coding segments lowers the average number of times that building blocks are found, i.e. increases their stability. This effect is most noticeable with segment lengths greater than 50 bits. In the majority of our experiments, the 95% confidence intervals do not overlap the 95% confidence interval of runs without non-coding segments. In experiments that used segments of less than 50 bits, the average number of times building blocks are found is still less than the average without non-coding segments, but the 95% confidence intervals may overlap the interval without non-coding segments.

Figures 12 to 15 show the average generation at which each level is discovered for the four coefficient types. We prefer that the average discovery generation of each

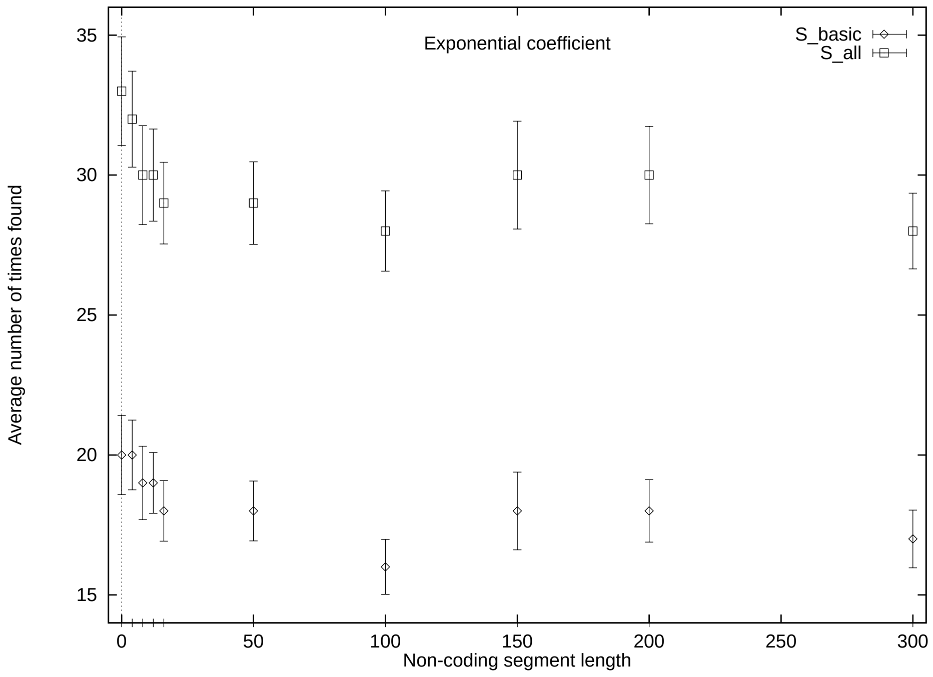

Figure 8: Data from the exponential coefficient runs: the average number times building blocks are found before the optimal solution is discovered.

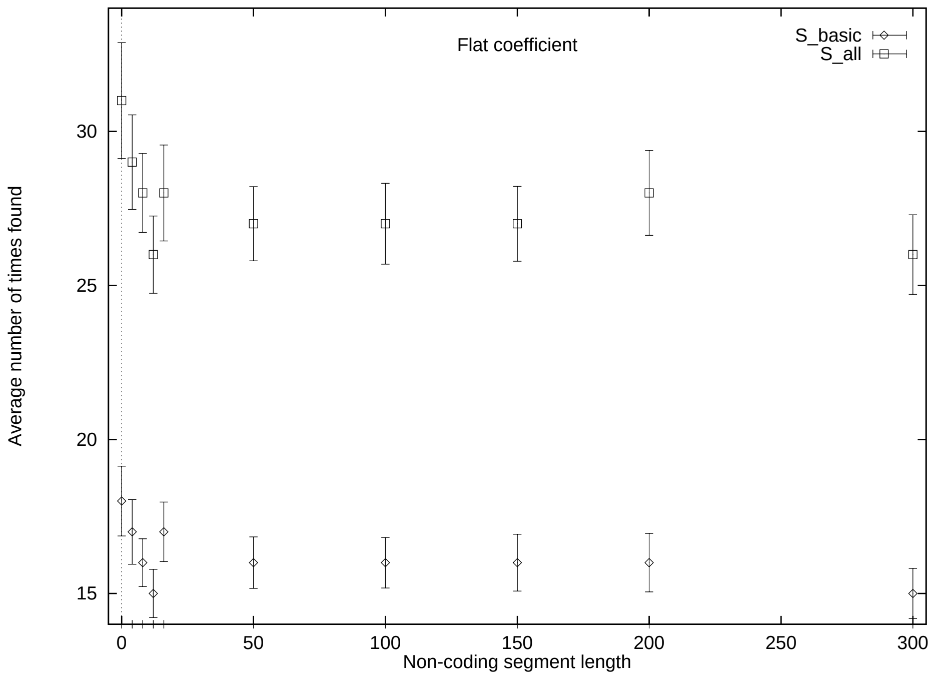

Figure 9: Data from the flat coefficient runs: the average number times building blocks are found before the optimal solution is discovered.

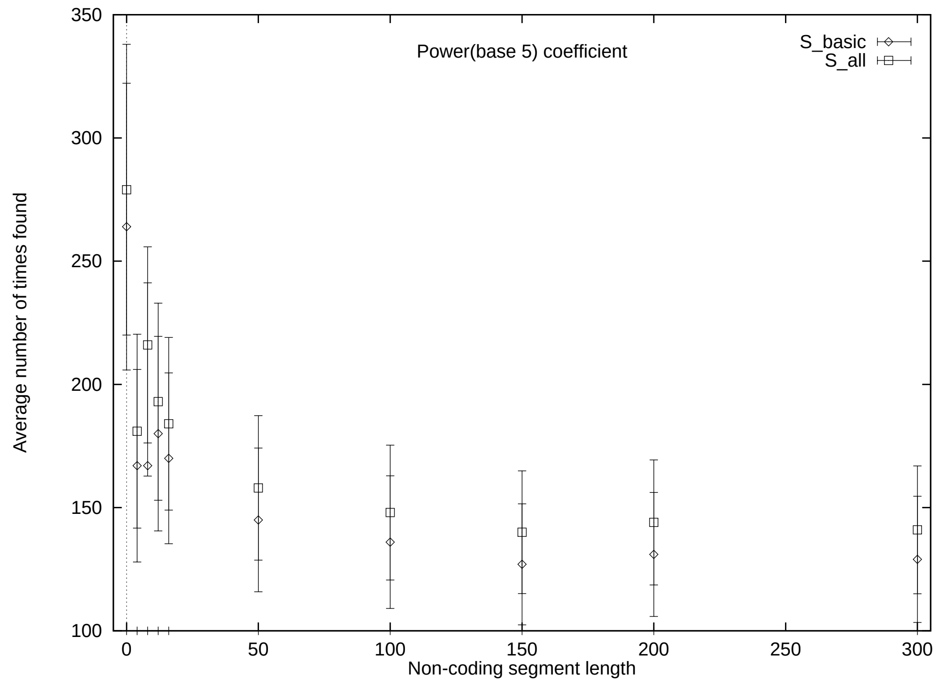

Figure 10: Data from the power(base = 5) coefficient runs: the average number times building blocks are found before the optimal solution is discovered.

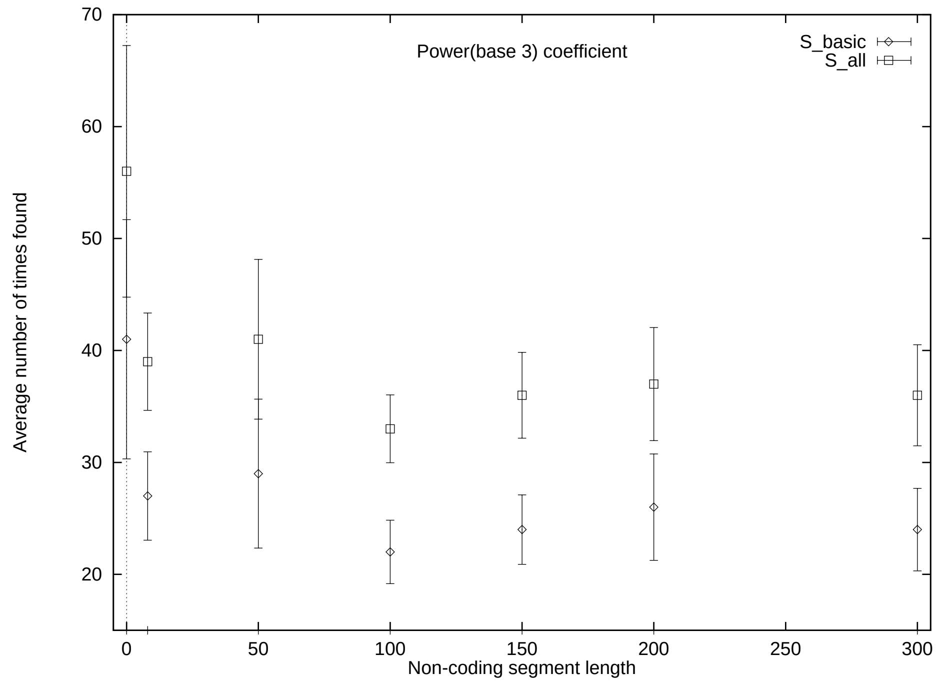

Figure 11: Data from the power(base = 3) coefficient runs: the average number times building blocks are found before the optimal solution is discovered.

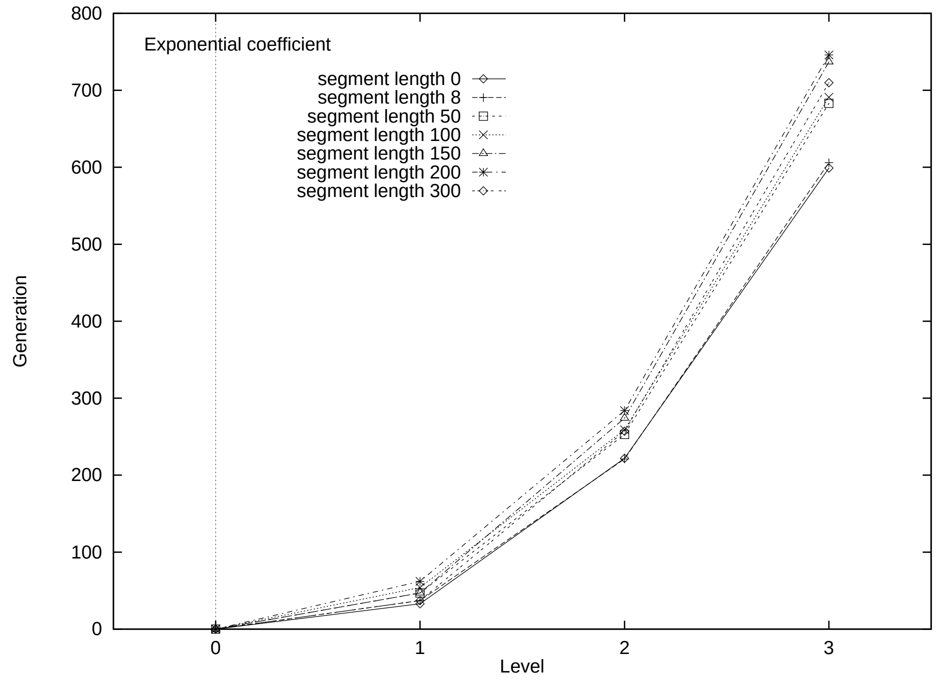

Figure 12: Data from the exponential coefficient runs: the average generation at which each level is discovered.

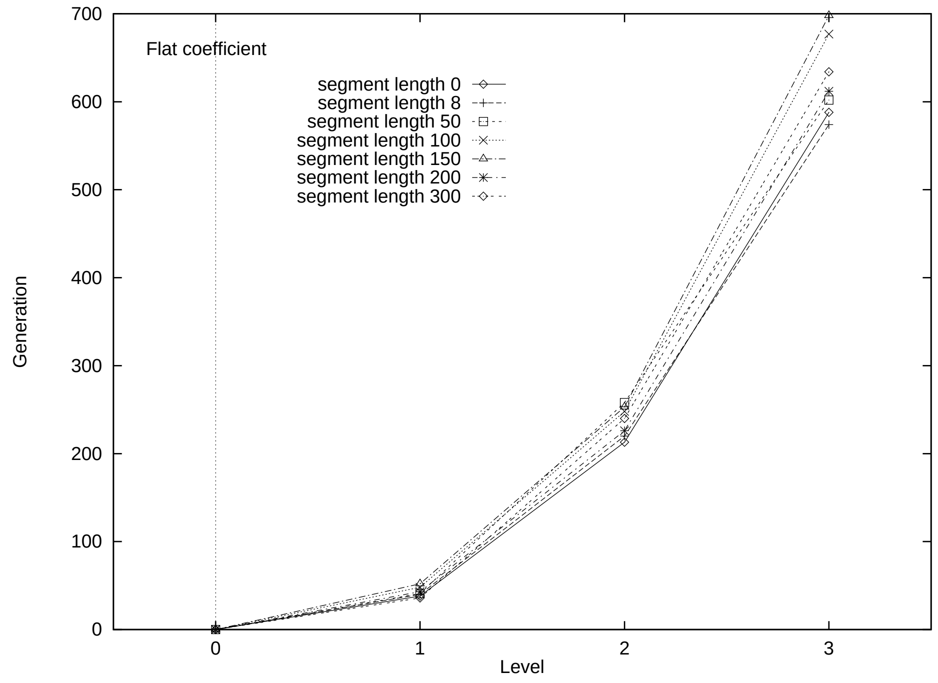

Figure 13: Data from the flat coefficient runs: the average generation at which each level is discovered.

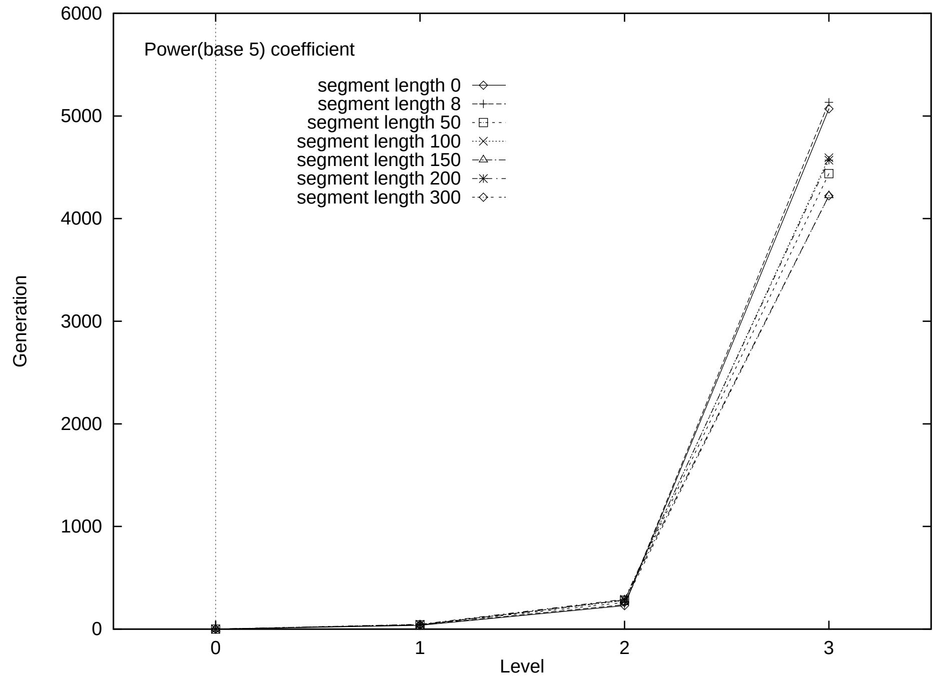

Figure 14: Data from the power(base = 5) coefficient runs: the average generation at which each level is discovered.

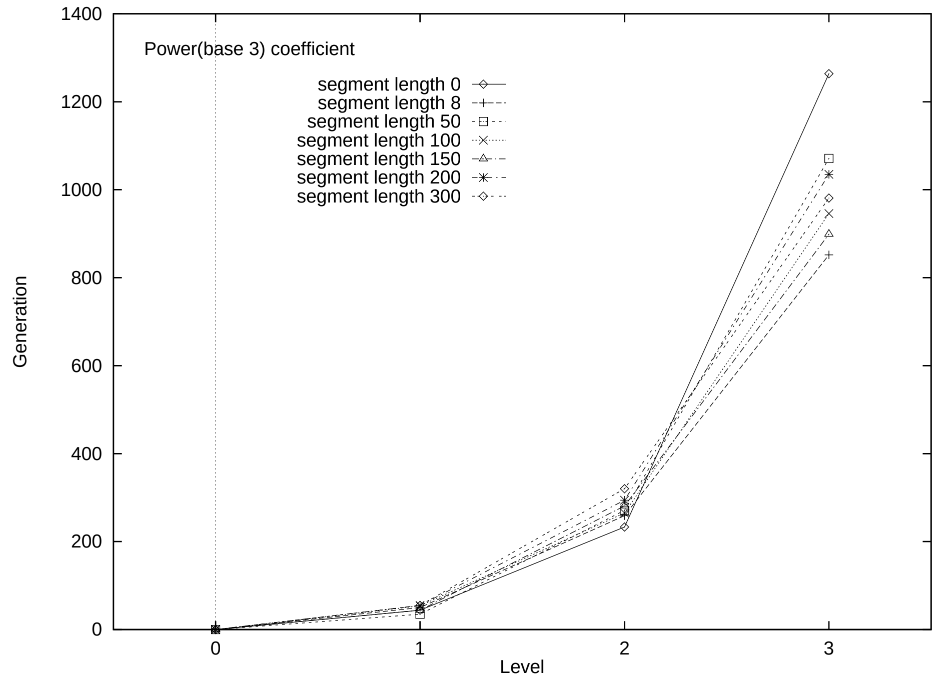

Figure 15: Data from the power(base = 3) coefficient runs: the average generation at which each level is discovered.

s_1 = 1111**; $l_1 = 0$

s_2 = ****1111**; $l_2 = 0$

s_3 = ********1111**; $l_3 = 0$

s_4 = ************1111**; $l_4 = 0$

s_5 = ****************1111**; $l_5 = 0$

s_6 = ********************1111**; $l_6 = 0$

s_7 = ************************1111************************************; $l_7 = 0$

s_8 = ****************************1111********************************; $l_8 = 0$

s_9 = ********************************1111****************************; $l_9 = 0$

s_{10} = ************************************1111************************; $l_{10} = 0$

s_{11} = **1111********************; $l_{11} = 0$

s_{12} = **1111****************; $l_{12} = 0$

s_{13} = **1111************; $l_{13} = 0$

s_{14} = **1111********; $l_{14} = 0$

s_{15} = **1111****; $l_{15} = 0$

s_{16} = **1111; $l_{16} = 0$

s_{17} = 11111111**; $l_{17} = 1$

s_{18} = ********11111111**; $l_{18} = 1$

s_{19} = ****************11111111**; $l_{19} = 1$

s_{20} = ************************11111111********************************; $l_{20} = 1$

s_{21} = ********************************11111111************************; $l_{21} = 1$

s_{22} = **11111111****************; $l_{22} = 1$

s_{23} = **11111111********; $l_{23} = 1$

s_{24} = **11111111; $l_{24} = 1$

s_{25} = 1111111111111111**; $l_{25} = 2$

s_{26} = ****************1111111111111111********************************; $l_{26} = 2$

s_{27} = ********************************1111111111111111****************; $l_{27} = 2$

s_{28} = **1111111111111111; $l_{28} = 2$

s_{29} = 11111111111111111111111111111111********************************; $l_{29} = 3$

s_{30} = ********************************11111111111111111111111111111111; $l_{30} = 3$

s_{opt} = 1111111111111111111111111111111111111111111111111111111111111111; $l_{opt} = 4$

Figure 16: A five level Royal Road function, $RR3$. Additional levels are added by doubling the length of the individual and, consequently, doubling the number of schemata for each level.

level to be as low as possible. At the lower levels, there is very little variation in the average discovery generation as the non-coding segment lengths vary. There is a significant spread at the highest level, but the ordering seems to be completely arbitrary. There is no obvious correlation between segment length and discovery generation. We should note that with the power coefficient, including segments of length 50 or larger does result in a significant improvement over short or no segments.

5.3.2 The effect of non-coding segments on different leveled Royal Road functions

Our second set of experiments was performed on the Royal Road function, RR3, shown in Figure 16. The goal of this set of experiments was to see if the effects of non-coding segments become more apparent when the number of levels in the Royal Road function increases. Figure 16 shows a five level Royal Road function. The number of levels is changed by doubling or halving the length of an individual and the number of schemata in each level. We decided to pursue this work because of the data that we collected in Figures 12 to 15. The higher the level, the larger the range of the average discovery generations. If this trend continues as the number of levels increase, then the differences in behavior due to different non-coding segment lengths should be more obvious in functions with more levels. In our experiments, we test the function from Figure 16 with four, five, and six levels. The following parameter settings were used for these runs:

Population size:	128
Crossover rate:	0.7
Mutation rate (per bit):	0.005
Reproduction rate:	1.0
Parent selection:	Roulette wheel, with replacement
Expected number offspring per individual:	0 to 2
Stopping criterion:	10000 generations

Number of levels:	4	5	6
Significant bits:	32	64	128
Number of basic building blocks:	8	16	32
Number of total building blocks:	15	31	63

Each experiment was run 100 times and the average values of these runs are reported here.

Initially we planned to run these experiments on RR2. After some testing, however, we found that adding levels to RR2 made the problem much too big. Running all of the runs we needed would take too much time. We decided to switch to a Royal Road function with building blocks of length four instead of eight. On small problems (individuals with eight or less building blocks), the short length of the building blocks makes the problem almost trivial for the GA. This is evident in the results from our four level runs. Little information was gained from these runs because there is very little variation in the data values with changing non-coding segment lengths. As a result, we will list these values in Table 5 rather than graphing them as we do for the other data. We found that when the number of building blocks, and consequently levels, are increased, this problem becomes more challenging for the GA and produces interesting results.

Figures 17 and 18 show the average number of generations the GA uses to find the optimum with the five and six level RR3 functions, respectively. In both sets of experiments, adding non-coding segments reduces the number of generations required to find the optimum. The effect is especially clear in the six level experiment where increasing the segment lengths result in steadily decreasing generation numbers.

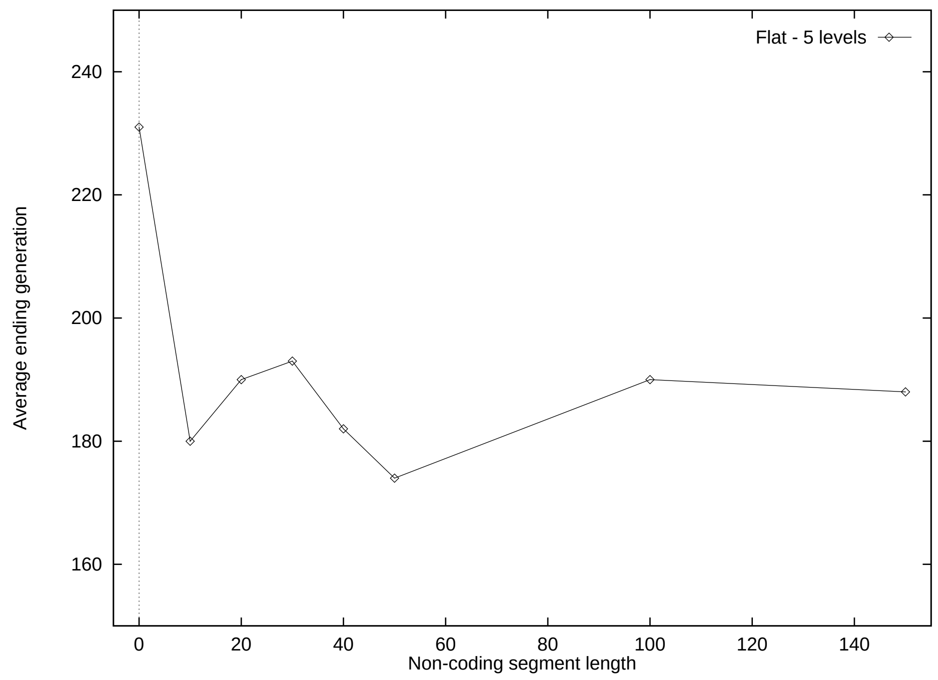

Figure 17: Data from the five level flat coefficient runs: the average number of generations needed to find the optimum versus non-coding segment length.

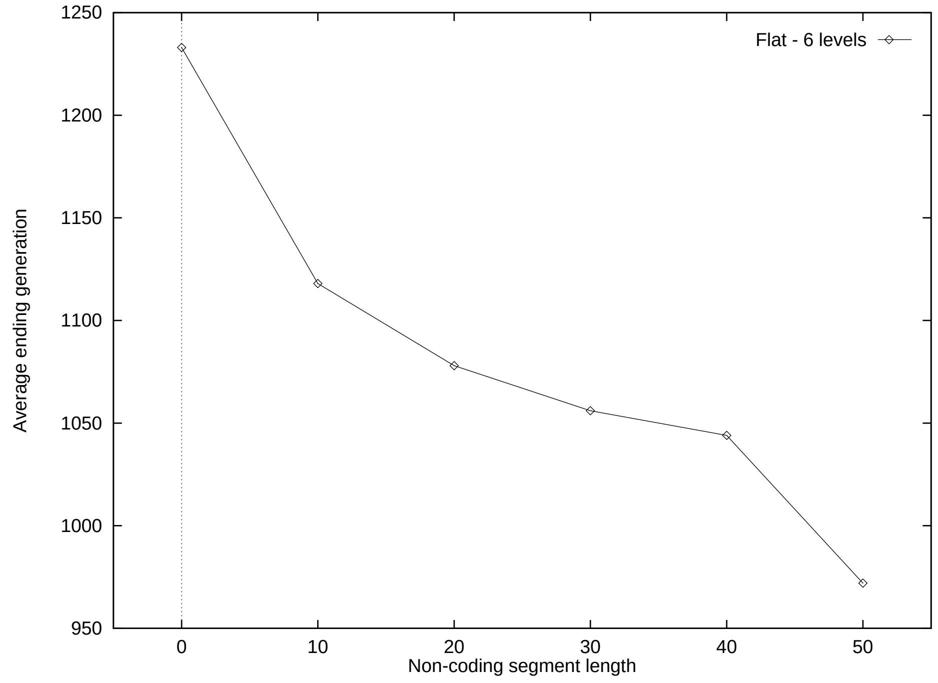

Figure 18: Data from the six level flat coefficient runs: the average number of generations needed to find the optimum versus non-coding segment length.

Segment length	Final generation	S_basic	S_all	Level 0 (1st gen.)	Level 1 (1st gen.)	Level 2 (1st gen.)	Level 3 (1st gen.)
0	25	9	17	0	0	7	25
100	26	9	17	0	0	6	26
200	29	9	17	0	0	5	29
300	26	9	17	0	0	6	26

Table 5: Average values from the four level RR3 runs.

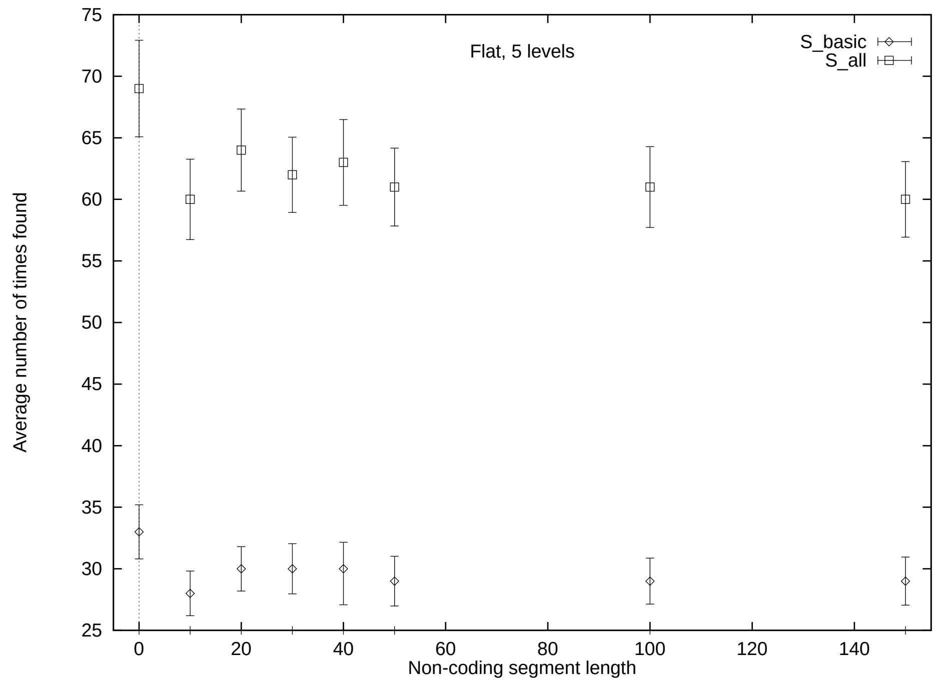

Figure 19: Data from the five level flat coefficient runs: the average number times building blocks are found before the optimal solution is discovered.

Figures 19 and 20 show the building block stability data from the five and six level RR3 runs. Again, S_{basic} refers to the sum of the number of times each basic building block is found, S_{all} refers to the sum of the number of times all of the building blocks are found, and the vertical bars indicate the 95% confidence intervals for the average values. Both the five and six level runs show a significant improvement when non-coding segments are added. The improvement seems to be more apparent in S_{all} than in S_{basic}.

Figures 21 and 22 show the average generation at which each level is discovered in the five level and six level RR3 runs. In both sets of runs, including non-coding

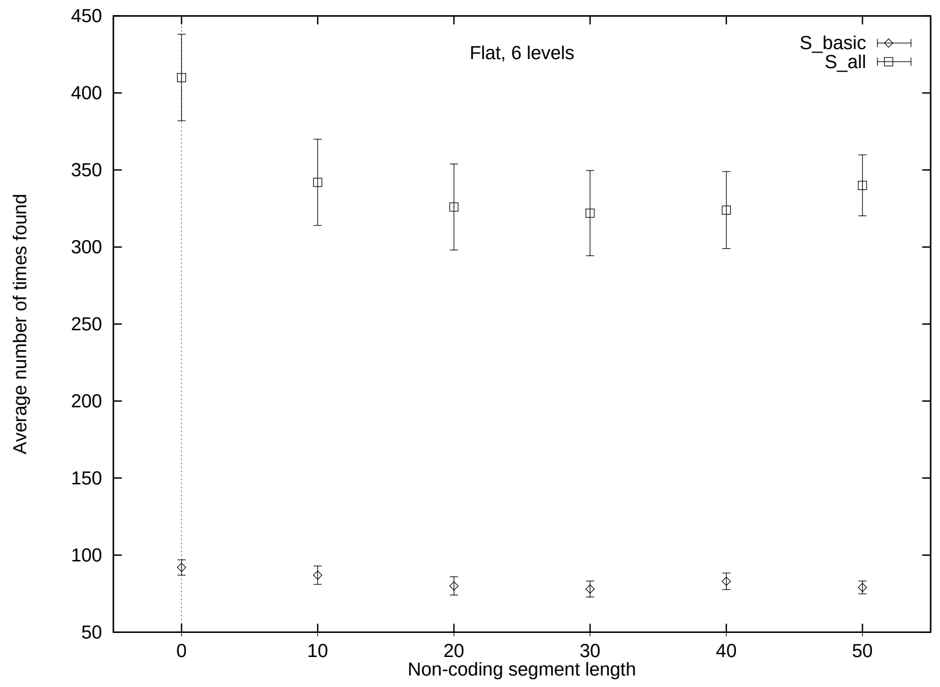

Figure 20: Data from the six level flat coefficient runs: the average number times building blocks are found before the optimal solution is discovered.

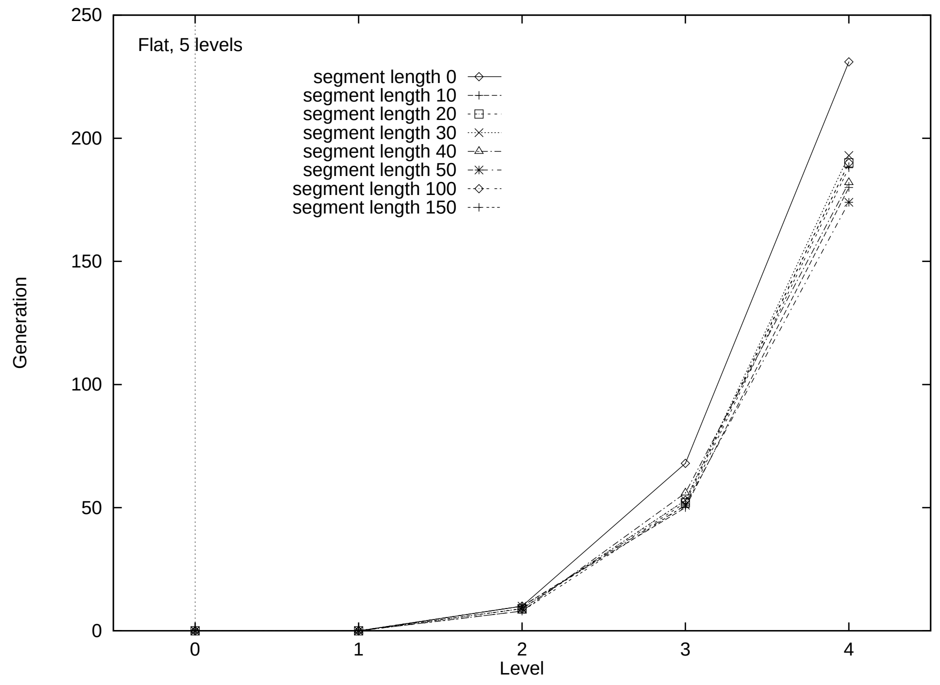

Figure 21: Data from the five level flat coefficient runs: the average generation at which each level is discovered.

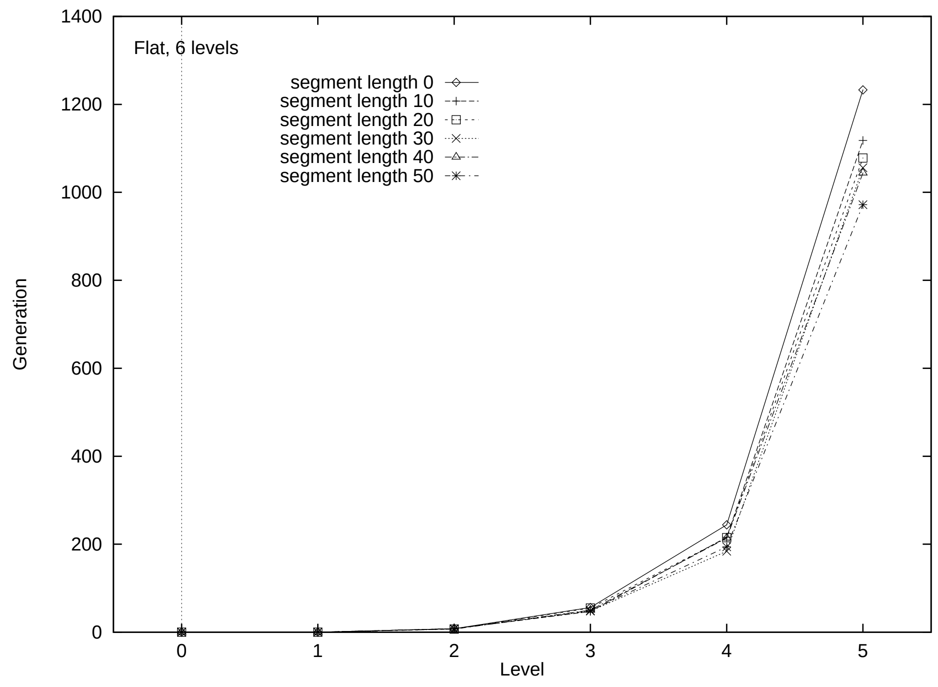

Figure 22: Data from the six level flat coefficient runs: the average generation at which each level is discovered.

segments shows a distinct improvement over not including the segments. Like our experiments with RR2, the distinction seems to depend on whether non-coding segments are used, and not on the size of the segments. Once these segments are added, there does not seem to be any correlation between segment length and the average level discovery generations.

6 Conclusions

We can draw a number of conclusions from the results described in this paper.

First, we were able to successfully reproduce portions of Levenick's research and Forrest and Mitchell's research. Of particular significance to us is that we were able to reproduce Levenick's results showing that non-coding segments increase the likelihood of finding a particular solution within a limited time frame. The random nature of the GA makes exact replication of experiments difficult. Under similar run-time conditions, however, we have been able to obtain statistically similar results.

Second, the effect of non-coding segments varies depending on the fitness landscape of the problem. The exponential and flat RR2 runs performed worse when non-coding segments are added (see Figures 4 and 5). Both the power RR2 runs (Figures 6 and 7) and the RR3 runs (Figures 17 and 18) from the second experiments showed improvement (shorter run times) with the addition of non-coding segments. Problems in which there is a large increase in fitness when the GA discovers a new level seem to benefit more from the inclusion of non-coding segments than problems in which the fitness increase from level to level is relatively small. Since the larger the fitness increase between levels, the more pronounced the hitchhiking effect will be, non-coding segments must have some effect on reducing the hitchhiking effect.

Among the arguments given in Section 3 for using non-coding segments is the hypothesis that non-coding segments may reduce the chance of crossover breaking up existing building blocks by reducing crossover activity within building blocks. While lowering the activity level within building blocks makes it harder for the GA to destroy existing building blocks, it also decreases the chance that the GA will put together a building block using crossover. As a result, the discovery of building blocks is partially inhibited and depends more on mutation alone. Thus, longer run times are not an unexpected side effect of adding non-coding segments.

Third, all of our experiments indicate that including non-coding segments in a GA reduces the number of times that building blocks are found (see Figures 8 to 11, 19, and 20). This, in turn, means that the GA is more stable with non-coding segments — the GA is less likely to lose building blocks after they have been found. Theoretical analysis of this situation supports our experimental results. Taking RR2 as an example, we have 64 significant bits grouped into eight building blocks. Any location is equally likely to be chosen as a crossover location. With no non-coding segments, there are eight crossover locations (counting both ends as the same location) that would not break up a basic building block. Thus, $8/64 = 1/8 = 0.125$ of the crossover locations will never destroy any basic building blocks and $7/8 = 0.875$ of the locations could potentially destroy basic building blocks. If non-coding segments of length n are added in between each basic building block, the individual becomes $64 + 8n$ long. Now there are $8(n + 1)$ crossover locations that will not break up a basic building block. This means that $\frac{8n+8}{64+8n} = \frac{n+1}{n+8}$ of all crossovers will never destroy a basic building block and $\frac{(64+8n)-(8n+8)}{64+8n} = \frac{7}{8+n}$ of all crossovers could potentially destroy a basic building block. Given that $n > 0$, the following two inequalities always hold:

$$\frac{n+1}{n+8} > \frac{1}{8}$$

and

$$\frac{7}{8+n} < \frac{7}{8}$$

. The first inequality shows that including non-coding segments increases the percentage of crossover locations that do not break up a basic building block. The second inequality shows that including non-coding segments decreases the percentage of crossover locations that could destroy a basic building block.

Fourth, the differences observed from different lengths of non-coding segments seem to be more pronounced at higher levels (see Figures 21 and 22). The more levels a Royal Road function has, the more building blocks there are to find, and the more difficult the problem is to solve. In our second set of experiments, we can see a definite advantage in all performance criteria for runs that use non-coding segments over runs that do not use non-coding segments. In both the five level and six level runs, the difference between having non-coding segments and not having non-coding segments is very clear. In the six level runs, performance improves steadily as segment length increases. In the five level runs, however, there does not seem to be any correlation between segment length and performance. These data suggest that large problems with many levels may be the most useful candidates for further studies on the effects of non-coding segments.

The purpose of this work is not to optimize GA performance on the Royal Road functions using non-coding segments. Rather, we use the Royal Road functions as a tool for studying the effects of non-coding segments on the GA. The hierarchical structure of this artificial function allows us to monitor the progress of the GA as it solves a problem. The pre-defined nature of this function allows us to rigorously test specific characteristics of the function while maintaining a constant value for all other characteristics. Our results suggest that non-coding segments may be beneficial in more complex problems, but not very influential in simpler problems. Though non-coding segments may slow down the discovery of new building blocks, they also stabilize the GA and reduce the chance of losing good building blocks. This type of information could be useful in helping others set up their problems in a GA. The better we understand the GA's problem solving process, the more effectively we can use the GA.

There are a number of extensions to this work that would be interesting to investigate. Our next area of study will involve location independent building blocks and non-coding segments. In these experiments, the placement (location and ordering) of the basic building blocks will be decided dynamically during a run by the GA. Other topics of interest that stem from the research described here include testing random length non-coding segments, comparing non-coding segment runs to runs with variable crossover probabilities, and testing the effects of non-coding segments on a real problem.

Acknowledgements

This research was sponsored by NASA/JSC under grant NGT-51057. The authors would like to thank John Holland and Michael Smith for many interesting discussions and helpful suggestions relating to this work.

References

Bäck, T. (1991). Self-adaptation in genetic algorithms. In *Toward a practice of autonomous systems: Proceedings of the First European Conference on Artificial Life*, (pp. 263–271).

Curtis, H. (1983). *Biology.* Worth Publishers.

De Jong, K. A. (1975). *Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, University of Michigan.

Eshelman, L. J., Caruana, R. A., & Schaffer, J. D. (1989). Biases in the crossover landscape. In *Proceedings of the Third International Conference on Genetic Algorithms*, (pp. 10–19).

Eshelman, L. J. & Schaffer, J. D. (1993). Crossover's niche. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, (pp. 9–14).

Fogarty, T. C. (1989). Varying the probability of mutation in the genetic algorithm. In *Proceedings of the Third International Conference on Genetic Algorithms*, (pp. 104–109).

Forrest, S. & Mitchell, M. (1992). Relative building-block fitness and the building-block hypothesis. In *Proceedings of the Foundations of Genetic Algorithms Workshop.*

Forrest, S. & Mitchell, M. (1993). What makes a problem hard for a genetic algorithm? Some anomalous results and their explanation. *Machine Learning*, *13*, 285–319.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley.

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

Goldberg, D. E., Korb, B., & Deb, K. (1990). Messy genetic algorithms revisited: Studies in mixed size and scale. *Complex Systems*, *4*, 415–444.

Grefenstette, J. J. (1986). Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, *SMC-16*(1), 122–128.

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

Lee, M. A. & Takagi, H. (1993). Dynamic control of genetic algorithms using fuzzy logic techniques. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, (pp. 76–83).

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. & Holland, J. H. (1993). When will a genetic algorithm outperform hillclimbing? In *Proceedings of the Fifth International Conference on Genetic Algorithms.*

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

Oliver, I. M., Smith, D. J., & Holland, J. R. C. (1987). A study of permutation crossover operations on the traveling salesman problem. In *Proceedings of the Second International Conference on Genetic Algorithms*, (pp. 224–230).

Patrusky, B. (1992). The intron story. *MOSAIC*, *23*(3), 22–33.

Schaffer, J. D. & Eshelman, L. J. (1991). On crossover as an evolutionarily viable strategy. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, (pp. 61–68).

Schaffer, J. D. & Morishima, A. (1987). An adaptive crossover distribution mechanism for genetic algorithms. In *Proceedings of the Second International Conference on Genetic Algorithms*, (pp. 36–40).

Syswerda, G. (1989). Uniform crossover in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, (pp. 2–9).