Introns in Nature and in Simulated Structure Evolution

Peter Nordin[1] Wolfgang Banzhaf[1] and Frank D. Francone[2]

[1] LS11, CS, University of Dortmund, Dortmund, Germany; email nordin,banzhaf@ls11.informatik.uni-dortmund.de
[2] RML inc.,360 Grand Ave., Oakland, CA, USA; email YTNS65A@prodigy.com

Abstract. In this study we *measure* the compression of information in a simulated evolutionary system. We do the investigation taking *introns* in the genome into account. We mainly investigate evolution of linear computer code but also present results from evolution of tree structures as well as messy genetic algorithms. The size of solutions is an important property of any system trying to learn or adapt to its environment. The results show significant compression or constant size of exons during evolution—in contrast to the rapid growth of overall size. Our conclusion is that an *built-in pressure* towards low-complexity solutions is *measurable* in several simulated evolutionary systems which may account for the robust adaptation showed by these systems.

1 Introduction

DNA works principally by coding for the production of proteins, polypeptides or RNA. One of the many curiosities of biology is that *over 70 %* of the DNA base pairs in eucaryotic life forms do not produce proteins, polypeptides or RNA, [11]. While some of these DNA base pairs are undoubtedly control sequences that turn other genes on and off, most of them appear to do nothing at all. Biologists refer to much of this code as *'introns'* or *'junk DNA'*. Excess like this rarely appears in evolved creatures. Is this code merely excess — useless DNA that is somehow left over from other times — or has it evolved as a direct result of the process of evolution?

Biologists are not in agreement on why or how introns have evolved. We know that the normal DNA repair mechanisms are apparently *not* applied to intron segments because the mutation rate in these segments is far higher than in functioning genes. This suggests that the particular base pair sequence in introns is not especially important.

Nevertheless, introns appear to serve some purpose (or at least to be a deliberately included part of the genome) because of the precision with which they are handled in the process of protein production. The beginning and ending of introns are precisely marked by specific base pair sequences. Much protein synthesis begins when a gene produces a strand of mRNA (messenger RNA), which then produces the protein or protein segment (a polypeptide). What happens when there are one or more intron segments in the sequence of a gene? The

answer is fascinating. The genetic machinery is able to prohibit this genetic material from producing polypeptide by curving out and clipping off this genetic information (Figure 1). Thus, intron sequences are never converted into amino acids or proteins. Such a purposeful mechanism hardly appears likely to have occurred over and over in the genome (i.e. for every intron) by accident. Somehow, natural evolution appears to *select* for the existence of introns.

The facinating fact is that introns emerge in simulated evolution as well. Genetic Programming (GP) is a form of simulated structure evolution where computer programs are evolved through simulated natural selection [6]. In GP the analog to biological introns would be evolved code fragments that do not affect the quality of an individual. For example: $y = y + 0$.

The evolution of such code fragments has been repeatedly observed by GP

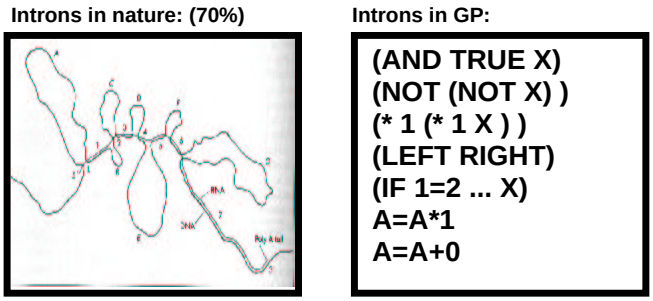

Fig. 1.: Introns in Nature and GP. The leftmost image shows how the introns are curved and cut before transcription.

researchers in tree–based GP–some refer to the phenomenon as *bloat* , [10], [2]. Some researchers argue that introns appear in GP populations because they have an important function in evolution. As a result, the algorithm selects for the existence of GP introns in a wide variety of conditions. Introns can be interpreted as punctuation characters in the genome, directing the crossover point to feasible sites. Introns are, however, a decidedly mixed blessing in GP.

Introns have also been applied to Evolutionary Algorithms. For instance, when GA researchers had previously worked with introns, such studies had been conducted by explicitly inserting introns into the fixed length GA structure [7]. In GP, by way of contrast, introns emerge spontaneously from the process of evolution as a result of the variable length of GP structures. This emergent property may be important to successful evolution [2].

Evolution in nature as well as in computer simulation is a form of learning. Compression of information is an important concept in the theory of learning. We argue for the hypothesis that there is an inherent compression pressure towards short, elegant and general solutions in variable length evolutionary algorithms

and potentially in natural evolution. This pressure becomes visible if the size or complexity of solutions is measured without introns. The built in "parsimony" selection pressure influences *generality* and *adaptiveness* of the phenotype. Some of these effects are positive and some are negative. In this work we provide a basis for an analysis of these effects and suggestions to overcome the negative implications in order to obtain the balance needed for successful evolution.

The positive effect of compact information representation is often referred to as Occam's Razor. The principle of Occam's Razor, formulated 700 years ago, states that from two possible solutions to a problem we should choose the shorter one. Bertrand Russell claims that the actual phrase used by William of Ockham was: "It is vain to do with more what can be done with fewer". A famous example of Occam's Razor is when the Polish astronomer Copernicus argued in favor of the fact that the earth moves around the sun and not vice versa, because it would make his equations simpler. Many great scientists have formulated their own versions of Occam's Razor. Newton, in his preface to Principia, preferred to put it as; "Natura enim simplex est, et rerum causis superfluis non luxuriat". (Nature is pleased with simplicity and affects not the pomp of superfluous causes.) The essence of Occam's Razor is that a shorter solution is a more generic solution. The process of inferring a general law from a set of data can be viewed as an attempt to compress the observed data. In a similar way could the adaptation of a phenotype be seen as a learning process for shape, behavior and strategy from a set of selection events.

We would like to argue that one of the foundations of evolution in general and Genetic Programming in particular is that they have the built in property of favoring short solutions and sub-solutions. This property may be one of the reasons that evolution works so efficiently and robustly in a diverse set of domains. The compression property could also be responsible for the ability of a solution to be generic and applicable to a larger set of data than the set of seen training data (selection events). The other side of the coin is that the built in compression pressure in certain cases is too strong and results in premature convergence or failure to adapt to complex fitness functions. The Evolutionary Algorithm could choose a short but incomplete solution instead of a long but complete solution. The strength of the pressure is dependent on the different attributes of a particular Evolutionary Algorithm such as, representation, genetic operators and probability parameters.

The bottom line is that it is helpful to be aware of this compression pressure and to try to keep it on a balanced optimum level during evolution. Here we also try to *measure* the compression of the active part of the genome. However, first, we will give a short introduction to evolutionary algorithms and genetic programming.

2 Evolutionary Algorithms and Genetic Programming

Genetic programming uses an evolutionary technique to *breed* programs [6]. First, a goal in the form of a quality criterion is defined. This so-called fitness function

could, for instance, be the error in a symbolic regression function. The population – a set of solution candidates – is initialized with random content, i.e., as random programs. In each "generation" the fittest individual programs are selected for reproduction. These highly fit individuals have offspring through recombination, often called crossover, and mutation. Various methods exist for selection and reproduction but the idea is always that better individuals and their offspring gradually replace the worse performing individuals
The individual solution candidate is represented as a tree which constitutes its genome. This tree can be seen as the parse tree of the program in a programming language. Recombination is normally performed by two parents exchanging subtrees, see Figure 2.

A typical application of GP is symbolic regression. Symbolic regression is the

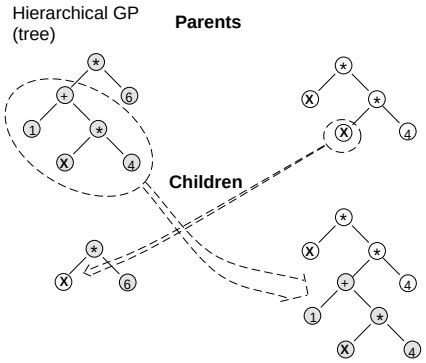

Fig. 2.: hierarchical crossover in genetic programming.

procedure of inducing a symbolic equation, function or program which fits given numerical data. Genetic programming is ideal for symbolic regression and most GP applications could be reformulated as variants of symbolic regression. A GP system performing symbolic regression takes a number of numerical input/output relations, called fitness cases, and produces a function or program that is consistent with these fitness cases. Consider, for example, the following fitness cases:

$$f(2) = 6, f(4) = 20, f(5) = 30, f(7) = 56, \tag{1}$$

One of the infinite number of perfect solutions would be $f(x) = x^2 + x$. The fitness function would, for instance, be the sum of difference between an individual's (function's) actual output and the output specified by the fitness cases. The *function set*, or the function primitives, could in this case be the arithmetic primitives $+, -, \cdot, /$, as shown in Figure 2.

We have used a variant of a variable length Genetic Algorithm operating on a string of bits to evolve an algorithm or program for a register machine [8]. Using a register machine makes the analysis of introns more straight forward, and using a bit string representation will simplify the complexity reasoning. The argumentation, however, is analogous for other Evolutionary Systems. Section 5.1 presents results from other paradigms of simulated evolution.

The structures that undergo evolution are variable length strings of 32 bit instructions for a register machine (CPU). The register machine performs arithmetic operations on a small set of registers. The 32 bits in the instruction represent simple arithmetic operations such as "a=b+c" or "c=b*5". The actual format of the 32 bits corresponds to the machine code format of a SUN-4, which enables the genetic operators to manipulate binary code directly.
Most other genetic programming approaches use a technique where a problem–specific language is executed by an interpreter. The individuals in the population are decoded at run time by a virtual machine. The data structures in those programs often have the form of a tree.

We have implemented the idea of using the lowest level binary machine code as the "programs" in the population. Every individual is a piece of machine code that is called and manipulated by the genetic operators. There is no intermediate language or interpreting part of the program. The structure of an individual can be seen in Figure 3.

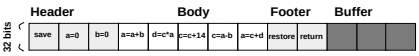

Fig. 3.: Structure of a program individual.

The genetic operators The evolutionary algorithm has the following two operators:

- A *mutation* operator changes the content of an instruction by mutating constants and register references.
- The *crossover* method operates on variable length individuals (Figure 4). Crossover is only allowed between instructions at 32–bit intervals in the binary string.

All operators ensure syntactic closure during evolution.
The DNA genome in nature has a linear structure. It is a string of letters in a four letter alphabet. It is furthermore subdivided into genes. Each gene codes for a specific protein. A gene could thus be seen as a segment of the genome which can be interpreted – and has a meaning – in itself. In our case a gene is a line of code representing an instruction or command. Such a command is also

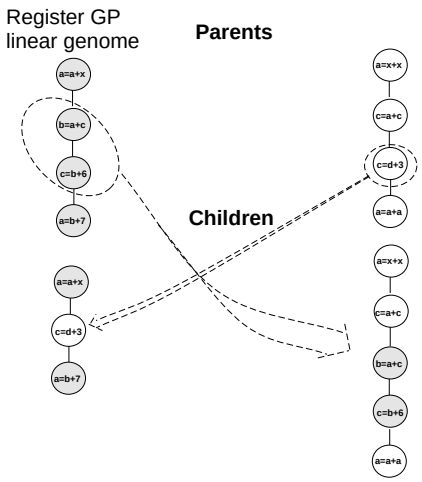

Fig. 4.: Crossover of a machine code program with a linear genome.

syntactically closed in the sense that it is possible to execute it independently of the other "genes". This method thus has some analogy to the gene concept in nature – it consists of a syntactically closed independent structure which has a defined starting and ending point. It is in both cases treated as a separate structure from the whole genome structure.
This variant of genetic programming that uses crossover for manipulations of whole genes/instructions and mutation for manipulation of the inside of the gene/instructions can be used with other computer languages than machine code see for instance.

2.1 Crossover and Fitness

A crossover acting on one block or segment of the code in an individual, might have different results. In one extreme case the two blocks that are exchanged in crossover are identical, therefore, the performance of the program is not affected at all. Normally, however, there is a high probability that the function of the program is severely damaged, resulting in a fitness decrease for the individual. In Figure 5 we can see a typical distribution of the effect of crossover on fitness in an early generation of a symbolic regression problem. The x-axis gives the change in fitness $\Delta f_{percent}$ after crossover f_{after}.($f_{best} = 0$, $f_{worst} = \infty$).

$$\Delta f_{percent} = \frac{f_{before} - f_{after}}{f_{before}} \cdot 100 \tag{2}$$

Individuals with a fitness decrease of more than 100 percent are accumulated at the left side of the diagram. This diagram shows that the most common effect

of crossover is a much worsened fitness (the spike at the left). The second most common effect is that nothing happens (the spike of zero). Below we use the term "probability of destructive crossover" for the probability that a crossover in the program or block will lead to a deteriorated fitness value, comprising the area left of zero in Figure 1 $p_d = P(\Delta f_{percent} < 0)$.

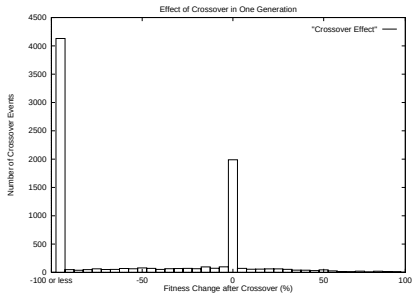

Fig. 5.: Effects of Crossover in one generation

3 Introns in GP

Genetic programs do not seem to favor parsimony in the sense that the evolved program structures become short and elegant measured with the absolute size of an individual. Instead, evolved programs seem to contain a lot of garbage and the solutions do not give an elegant impression when first examined.

On the contrary, solutions look unnecessarily long and complex.

Examples of introns can be found in most unedited individuals from a genetic programming run. The system can be very creative in finding such blocks. Some typical examples are:

(NOT (NOT X)), **(AND** ... **(OR X X))**, **(+** ... **(- X X))**, **(*** ... **(DIV X X))**, **(MOVE-LEFT MOVE-RIGHT)**, **(IF (2=1)** ... **X)**, **(SET A A)**

In this paper we give an explanation for why a program has a tendency to increase its length during the course of evolution (by adding introns) and at the same time favor Parsimony, and we try to measure the effect empirically. The crucial point is to measure the size of the code that are exons (not introns) called *effective length* instead of *absolute length*. Observing the effective length will clearly show that the evolutionary system not only favors parsimonious solutions for the final result, but constantly, for sub-solutions during the evolution of the population.

Let us say that we have a simple evolutionary system with fitness proportional selection and crossover as genetic operators. If we have an individual program with a high relative fitness in the population, it will be reproduced according to its fitness by the selection operator. Some of these new copies will undergo crossover and will loose one block and gain another. If the crossover interferes with a block that is doing something useful in the program, then there is a probability that this new segment will damage the function of the block, see figure 5. In most cases, the probability of damaging the program is much greater than the probability of improving the function of the block. If, on the other hand, the crossover takes place at a position within an intron block, then by definition there will be no harm done to this block or to the program.

A program with a low ratio of effective complexity to absolute complexity has a small "target area" for destructive crossover and a higher probability to constitute a greater proportion of the next population.

The additions of introns could be viewed as a way for the program to self-regulate the crossover probability parameter or as a "Defense against crossover" [?].

We can formulate an equation with resemblance to the Schema Theorem [5] for the relationship between the entities described above.

- Let C_j^e be the effective complexity of program j, and C_j^a its absolute complexity.
- Let p_c be the standard GP parameter for the probability of crossover at the individual level.
- The probability that a crossover in an *active block* of program j will lead to a worse fitness for the individual is the probability of destructive crossover, p_j^d. By definition p_j^d of an absolute intron is zero.
- Let f_j be the fitness of the individual and $\overline{f}^t$ be the average fitness of the population in the current generation.

Using fitness proportionate selection[3] and block exchange crossover, for any program j, the average proportion P_j^{t+1} of this program in the next generation is:

$$P_j^{t+1} \approx P_j^t \cdot \frac{f_j}{\overline{f}^t} \cdot \left(1 - p_c \cdot \frac{C_j^e}{C_j^a} \cdot p_j^d\right) \tag{3}$$

In short, Equation (3) states that the proportion of copies of a program in the next generation is the proportion produced by the selection operator less the proportion of programs destroyed by crossover. Some of the individuals counted in P_j^{t+1} might be modified by a crossover in the absolute intron part, but they are included because they still show the same behavior at the phenotype level. The proportion P_j^{t+1} is a conservative measure because the individual j might be recreated by crossover with other individuals.

Equation (3) could be rewritten as:

[3] The reasoning is analogous for many other selection methods.

$$P_j^{t+1} \approx \left(\frac{f_j - p_c \cdot f_j \cdot p_j^d \cdot C_j^e / C_j^a}{\overline{f}^t} \right) \cdot P_j^t \tag{4}$$

We can interpret the crossover related term as a direct subtraction from the fitness in an expression for reproduction through selection. In other words, reproduction by selection *and* crossover acts as reproduction by selection *only*, if the fitness is adjusted by the term:

$$p_c \cdot f_j \cdot \frac{C_j^e}{C_j^a} \cdot p_j^d \tag{5}$$

This could thus be interpreted as if there where a term (5) in our fitness which is proportional to program complexity.

We now define "effective fitness" f_j^e as:

$$f_j^e = f_j - p_c \cdot f_j \cdot \frac{C_j^e}{C_j^a} \cdot p_j^d \tag{6}$$

The effective fitness of a parent individual, therefore, measures how many children of that parent are likely to be chosen for reproduction in the next generation.[4] A parent can increase its effective fitness by lowering its effective complexity (that is, having its functional code become more parsimonious) or by increasing its absolute complexity or both. Either reduces the relative target area of functioning code that may be damaged by crossover. Either has the effect of increasing the probability that the children of that parent will inherit the good genes of the parent intact. In other words, the difference between effective fitness and actual fitness measures the extent to which the destructive effect of genetic operators is warping the real fitness function away from the fitness function desired by the researcher. In the next part of this text we will try the compression hypthesis on a large number of simulations in three different domains.

4 Problem set-up

We decided to study the compression hypothesis on three standard problems. The problem domains are collected from the machine learning literature [3]. Two of these are real world problems while one is a difficult artificial data set. The classification problems were cast into symbolic regression problems. Each class was given a range. We performed an extensive batch of test runs consisting of 420 complete CGPS runs with a population of 3000 each, evaluating several hundred billion test cases.

[4] This assumes $\overline{f}^t \approx \overline{f}^{t+1}$.

Phoneme Recognition Data Set The phone recognition problem is the same problem used for comparison between a register machine and a tree–based GP system mentioned in the previous chapter. The Phoneme recognition database contains two classes of vowels from isolated syllables spoken by different speakers. Each input training vector is five dimensional and describes different frequency properties of a vowel.

The Gaussian 3D Data Set Gaussian 3D is an artificially generated Classification problem. It has a three dimensional input and two different output classes. Class 0 is a set of points with a normal distribution across the three input axes with zero mean and standard deviation of 1. Class 1 is a similar series of points except that the standard deviation is 2 [3].

The Iris Data Set The IRIS data set is a well know classification benchmark from the machine learning community. The data set contains 150 examples of properties of iris plants which shall be classified into three different plant types.

5 Empirical Results

The average fitness (godness) during evolution of the function during a typical run is plotted in Figures 6 while 7, shows the evolution of absolute length and effective length in the same experiment.

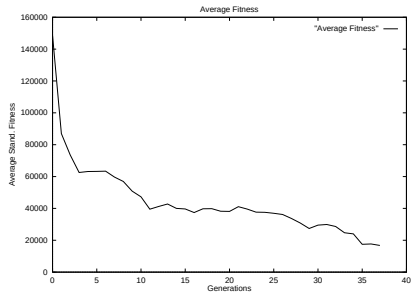

Fig. 6.: Average fitness

To support the hypothesis that compression achieves its goal of protecting the individual, we have plotted the effect of crossover in different generations. Figure 8 shows the change of effects of crossover during evolution. This diagram consists of many diagrams of the same type as Figure 5 placed in sequence after each other. We can see that the absolutely dominating effects of crossover are that either nothing happens to the fitness, or the fitness is worsened by more than

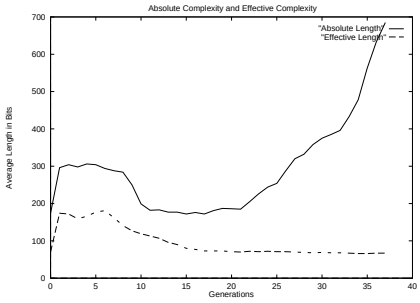

Fig. 7.: Average fitness

100 percent. The peak over the zero line increases which indicates a growingly unaffected fitness. The accumulated destructive effect of crossover to the left decreases after generation 15 as the ratio between absolute and effective length increases and the individual becomes more and more protected.

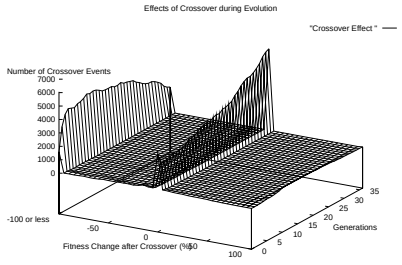

Fig. 8.: Distribution of crossover effect during evolution

All runs observed end with exponential intron growth, if given enough time(30-800 generation) to proceed.

We tried the compression hypothesis on the 480 runs and 3 problems. Compression was measured from the point in evolution where the fitness value stabilised and the evolution of the effective size from this point was taken as the compression measure. 64 runs ended with increased effective size (most of them a few percent), 100 runs decreased, 316 runs remained constant. Most runs where effective size increased show some problem with early termination or bad fitness. A majority of runs showed stable effective length or *compression*. Runs which increased terminated 9 generations earlier on average than the other runs. The constant or decreasing runs usually displayed this behaviour for several tens of

generations before the run was terminated by an exponential growth in absolute size (bloat).

The average compression was 4%.

5.1 Other Evolutionary Techniques

The results may be extended to a more canonical GP system with hierarchical crossover and tree representation. We have so far done initial experiments with a tree–based GP system doing symbolic regression. The results indicate a distribution of destructive crossover similar to that in the *linear representation GP system* distribution. Figure 10 shows the distribution of crossover effect of tree based GP system doing symbolic regression over 60 generations. Figure 9 shows the evolution of absolute and effective length in the same experiments. Few issues in our assumption are specific to *linear representation GP system* which together with results such as those in Figure 10 suggests that our results may apply to a wider domain of systems, see also [9]. Another evolutionary al-

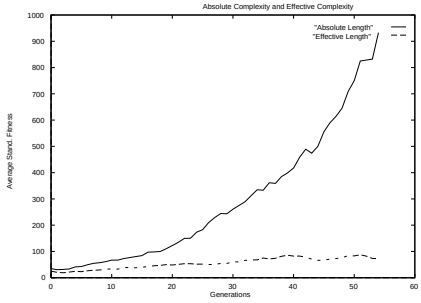

Fig. 9.: tree–based GP

gorithm with varying length binary genome is the *messy genetic algorithms* [4]. Initial experiments with this representation method for optimisation problems show similar results and a rapid growth of introns, see Figure 11.

6 Conclusions

In this paper we have measured *compression* in simulated evolution of structures. We have investigated the compression factor over a very large number of simulations (480) and three different domains. We have also showed similar compression behavior for other types of simulated evolution—messy-genetic algorithms and tree-structures. All results show a similar pattern where the absolute length of an individual grows rapidely while the *used* part of the genome

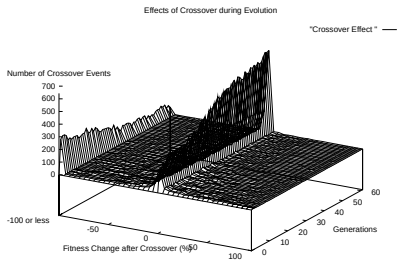

Fig. 10.: Crossover Effects In S-Expression Style GP

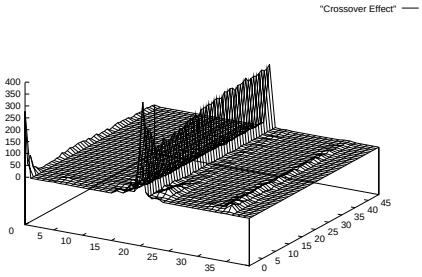

Fig. 11.: Messy-GA

stays constant (after some time), decreases or grows only very slowly. A possible explanation of this phenomenon is the need to protect the genome from the destructive effects of the genetic operators. The advantage of this compression phenomenon is that shorter solutions in often generalize better this could be one explanation of the power of evolutionary search, possibly extendable *beyond simulated evolution*—to natural evolution.

References

1. Altenberg, L. (1994) The Evolution of Evolvability in Genetic Programming. In *Advances in Genetic Programming*, K. Kinnear, Jr. (ed.), Cambridge, MA: MIT Press. pp47-74.
2. Angeline P.J. (1994) Genetic Programming and Emergent Intelligence In *Advances in Genetic Programming*, K. Kinnear, Jr. (ed.), Cambridge, MA: MIT Press.
3. ELENA partners, Jutten, C., Project Coordinator 1995.Esprit Basic Research Project Number 6891, Document Number R3-B1-P.
4. Goldberg D E, Korb B and Deb K (1989) Messy genetic algorithms: Motivation analysis and first results *Complex Systems* **3** pp 493-530.

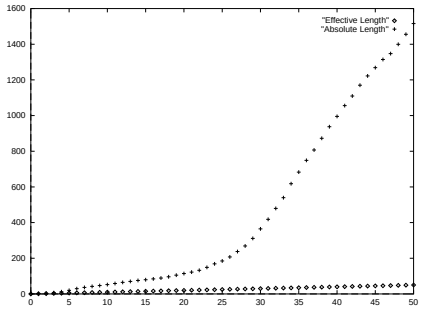

Fig. 12.: Messy-GA effective and absolute length

5. Holland J. (1975) *Adaption in Natural and Artificial Systems*, Ann Arbor, MI: The University of Michigan Press.
6. Koza J. (1992) *Genetic Programming*, MIT Press, Cambridge, MA.
7. 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*,Belew, R.K. and Booker, L.B., editors, Morgan Kauffman. San Mateo, California, pp. 123-7.
8. Nordin .J.P. (1994) A Compiling Genetic Programming System that Directly Manipulates the Machine-Code, in: *Advances in Genetic Programming*, K. Kinnear, Jr.(ed.), MIT Press, Cambridge, MA
9. Rosca J.P. (1995) Entropy-Driven Adaptive Representation. In *Proceeding of the GP workshop at Machine Learning 95, Tahoe City, CA*, J. Rosca (ed.),University of Rochester Technical Report 95.2, Rochester, NY, p. 23-32.
10. Tackett W. (1994) Recombination Selection and the Genetic Construction of Computer Programs. *Dissertation: Department of Electrical Engineering Systems*, University of Southern California, Los Angeles, CA.
11. Watson J.D, Hopkins, N.H, Roberts J.W, Wiener A.M, (1987) *Molecular Biology of the Gene*, Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc.

This article was processed using the LaTeX macro package with LLNCS style