Complexity Compression and Evolution

Peter Nordin
Universität Dortmund
Fachbereich Informatik
Lehrstuhl für Systemanalyse
D–44221 Dortmund
nordin@ls11.informatik.uni-dortmund.de

Wolfgang Banzhaf
Universität Dortmund
Fachbereich Informatik
Lehrstuhl für Systemanalyse
D–44221 Dortmund
banzhaf@ls11.informatik.uni-dortmund.de

Abstract

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 a genetic programming system and other variable length evolutionary algorithms. This pressure becomes visible if the size or complexity of solutions are measured without non-effective code segments called introns. The built in parsimony pressure effects complex fitness functions, crossover probability, generality, maximum depth or length of solutions, explicit parsimony, granularity of fitness function, initialization depth or length, and modularization. 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. An empirical investigation that supports our hypothesis is also presented.

1 Introduction

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. Some researchers have claimed that this principle could be the basis of many cognitive processes in the brain (Wolff 1993).

In this paper, we argue that one of the foundations of Evolutionary Algorithms in general, and Genetic Programming in particular, is that they have the built in property of favoring short solutions and sub-solutions. This property might be one of the reasons that Evolutionary Algorithms work 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 on a larger set of data than the set of data or fitness cases used during evolution. The other side of the coin is that the built in compression pressure in certain cases is too strong and results in premature convergence and 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.

In this paper, we focus on the problem of symbolic regression of programs with a genetic algorithm. 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 (Nordin 1994). 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 standard tree-representation Genetic Programming and the reasoning is useful for other evolutionary systems.

1.1 Destructive Crossover

We would like to start by defining some important concepts. These are destructive crossover, different kinds of introns and effective complexity.
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 1 we can see a typical distribution of the effect of crossover on fitness in an early generation of the symbolic regression problem from section 3. The x-axis gives the change in fitness $\Delta f_{percent}$ after crossover f_{after}.($f_{best} = 0$, $f_{worst} = \infty$).

$$\Delta f_{percent} = \frac{f_{before} - f_{after}}{f_{before}} \cdot 100 \qquad (1)$$

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, $p_d = P(\Delta f_{percent} < 0)$. The term could be used for complete programs as well as for blocks in programs.

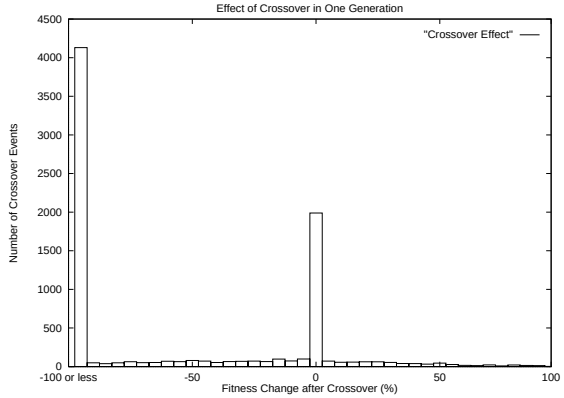

Figure 1: Effects of Crossover in one Generation.

1.2 Introns

As a rule, a solution evolved by Genetic Programming systems contains segments of code that do not seem to perform any useful functions, seem to be unnecessarily lengthy or that are not executed at all within the program. Similar redundant structures are present in nature within DNA and are called "introns" (Watson 1987). The idea of explicitly inserting introns in Genetic Algorithms has previously been investigated (Levenick 1991, Forrest 1992).

In the experiments performed in Section 3 below we use the term *intron segment* for a block that does not affect the behavior of the entire program for any of the fitness cases. The code is thus neutral on the phenotype level. Notice however that an intron segment may affect the output of a program for other inputs than the current ones. By the term *block* we mean any subset of the code regardless of its representation. It could be a sequence of binary digits in a binary string, a structure in a messy genetic algorithm, or a set of nodes within a tree representation. An *active block* is a block of code that is *not* an intron block and *does affect* the properties on the phenotype level.

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 in S-expression notation 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 reality, neither the extent to which a block affects output, nor the sensitivity of crossover is a discrete property of the block. There is a scale or distribution, both on, how sensitive the block is for crossover and how much it affects the program output. To simplify terms, we define an *absolute intron* as a block of code that neither affects the output of the program nor is sensitive to crossover. A crossover inside such a block will not affect the performance of the program. The following is an example of an absolute intron structure that can evolve when using the `if` function:

```
if 2<1 then {Absolute Intron Block ...}
       else {Active Block ...}
```

We call an intron *global* if it is an intron for every valid input to the program, and we call it *local* if it acts as an intron only for the current fitness cases and not necessarily for other valid inputs. This distinction is important for the generalization capabilities of a program, see Section 2.4.

In Section 3, we introduce a method for measuring the size of the intron segments in Evolutionary Algorithms. We look at intron blocks that do not affect

the behavior of the individual for any of the fitness cases. This kind of intron will not be totally immune against crossover but, as long as the population contains some of these segments, swapping two of them by crossover will not affect the performance of the two individuals involved. Introns of this kind arise by a mutual agreement among the individuals to keep these sort of NoOperation code blocks.

1.3 Effective Complexity

By *complexity* of a program or program block we mean the length or size of the program measured with a method that is natural for a particular representation. For a tree representation, this could be the number of nodes in the block. For the binary string representation, in our work it is the number of bits, etc. The absolute length or *absolute complexity* is the total size of the program or block. The effective length or *effective complexity* of a block, or program, is the length of the active parts of the code within the program or block, in contrast to the intron parts.

2 Program Complexity, Effective Fitness and Evolution

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 (Koza 1992). 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.

In this section, we give a reason for which a program has the tendency of increasing its absolute length during the course of evolution and at the same time favoring Parsimony. The crucial point is to measure *effective length* instead of *absolute length*. Observing the effective length will clearly show that the genetic programming 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 GP system with fitness proportional selection and crossover as genetic operators. The crossover operator could be any crossover operator exchanging blocks of code such as the standard tree based subtree exchanging crossover (Koza 1992) or two point bit string crossover (Nordin 1994). 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 1. 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 absolute 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 ability to create and add introns during evolution is another important property of the system and its primitives. This ability could depend on parameters like initial individual size, function set, and fitness function. 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" (Altenberg 1994).

We can formulate an equation with resemblance to the Schema Theorem (Holland 1975) for the relationship between the entities described above. Let C_{ej} be the effective complexity of program j, and C_{aj} its absolute complexity. Let p_c be the standard genetic programming parameter giving 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_{dj}. By definition p_{dj} of an absolute intron is zero. Let f_j be the fitness[1] of the the individual and $\overline{f^t}$ be the average fitness of the population in the current generation. If we use fitness proportionate selection[2] and block exchange crossover, then for any program j, the average proportion P_j^{t+1} of this program in the next generation is:

$$P_j^{t+1} \approx P_j^t \cdot \frac{f_j}{\overline{f^t}} \cdot \left(1 - p_c \cdot \frac{C_{ej}}{C_{aj}} \cdot p_{dj}\right) \qquad (2)$$

In short, Equation (2) 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

[1] Notice that this is not standardized fitness used in GP. Here a better fitness gives a higher fitness value (GA).

[2] The reasoning is analogous for many other selection methods including more elitist strategies

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, etc.[3] Equation (2) could be rewritten as:

$$P_j^{t+1} \approx \left(\frac{f_j - p_c \cdot f_j \cdot C_{ej} \cdot \frac{1}{C_{aj}} \cdot p_{dj}}{\overline{f^t}} \right) \cdot P_j^t \quad (3)$$

Here we see that 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:

$$-p_c \cdot f_j \cdot C_{ej} \cdot \frac{1}{C_{aj}} \cdot p_{dj} \quad (4)$$

This could thus be interpreted as if there where a term (4) in our fitness proportional to program complexity.

We now define "effective fitness" f_{ej} as:

$$f_{ej} = f_j - p_c \cdot f_j \cdot C_{ej} \cdot \frac{1}{C_{aj}} \cdot p_{dj} \quad (5)$$

It is the *effective fitness* that determines the number of individuals of a certain kind in the next generation.

Under these assumptions, it is possible to show that an individual can win a higher proportion of the next generation in spite of the fact that it has worse fitness. By loosing a block with high effective complexity with a moderate contribution to fitness it can increase its chances of survival. The individual is then trading normal fitness for effective fitness by reducing its complexity and a less fit individual can take a higher proportion of the next generation. This is an undesired phenomenon in most training situations.

2.1 Crossover Protection

An individual can do a number of things to protect itself from crossover. As discussed above, it can increase the absolute complexity by adding introns. The possibility of adding introns is limited by a number of factors for any given GP system under evolution. A system cannot add introns with arbitrary complexity instantly.

[3]The event of recreating individuals can be measured to be low except when applying a very high external parsimony pressure which forces the population to collapse into a population of short building blocks.

One other possibility for the individual to protect itself is to reduce the effective length by finding a more parsimonious solution, but this ability is limited by the fitness function and the dynamics of the system. It is in the balance between these two strategies that the parsimony pressure of a GP system appears.

If the effective length cannot be further reduced then the system will try to add more introns. This is on condition that the maximal length or depth of this system is not exceeded. If sufficient intron adding is allowed by the dynamics of the system, then it can gradually balance out differences in the effective fitness' by a short and a long program. All genetic programming systems have some defined maximal size of the program structure. This size is important because chosen too small it may limit the additions of intron, thus preventing the system from finding a perfect solution. In certain cases it is necessary for the intron blocks to be many times the size of useful blocks before an exact solution can be found. This means that it is important to adjust the allowed maximum size of individuals according to the composition of the fitness function.

A third possible way to reduce the probability for destructive crossover is to allow crossover at certain points only. Where crossover should be allowed could also be evolved through a suitable representation in the individual of allowed crossover points. This is common in nature, where there are numerous sophisticated systems for protection from changes in genetic material. Sexual recombination in higher species in particular is only allowed at very well defined points (Watson 1987).

The composition of the fitness function influences the sensitivity of the system too. For example, by altering the size of constants and scaling of different contributing part of the fitness function, it could be possible to reduce the crossover sensibility below the threshold of interference with the perfect solution.

2.2 Balanced Evolution

Adding an external parsimony pressure is one way to balance up the effective fitness. This pressure P is in its simplest form proportional to the absolute length of the program and subtracted from the fitness expression. This results in the following equation for the effective fitness:

$$f_{ej} = f_j - P \cdot C_{aj} - p_c \cdot f_j \cdot C_{ej} \cdot \frac{1}{C_{aj}} \cdot p_{dj} \quad (6)$$

The external parsimony pressure could be made variable during evolution and could also be allowed to have a negative value if necessary.

When evolving functions with non-continuous fitness functions for instance where the results is an integer, it is important to scale the output from the fitness function so that the smallest change in fitness, the granularity, is balanced against the biggest change in complexity. Otherwise, the change in fitness can drown in the change of complexity, see equation 5.

When heterogeneous complex fitness functions are used, it is also important to balance the contribution from the different parts in the fitness function.

The average initial size or complexity will affect the average complexity pressure in the beginning of a genetic programming session. The complexity pressure is proportional to the relation between the effective and absolute complexity. A high complexity at the initial state will give a low pressure in the beginning which is of importance for the dynamics of the whole training session.

Other strategies that could help evolving programs to obtain a shorter effective length are the introduction of control structures like loops or the use of subfunctions and other modularization techniques. In the next section we give a brief motivation for spontaneously emerging subfunctions.

2.3 Spontaneously Emerging Subfunctions

A natural tool for humans when defining an algorithm or computer program is to use modularization and divide the solution into smaller blocks of code. Different modularization techniques have been suggested for use with genetic programming, where the most thoroughly evaluated are automatically defined functions (ADF) (Koza 1994). Other examples of modularization techniques are Module Acquisition (Angeline 1993) and The Encapsulation Operation (Koza 1992). All modularization techniques are ways of encapsulating blocks of code. ADFs encapsulate blocks that becomes subroutines which could be called from the main program or from another subroutine. A subroutine can be called more than once from the same program. This means that a program can reduce its effective length by putting frequently used identical blocks into a subroutine. As we have seen, a small effective length increases the chances of survival of this offspring. This is in our mind one of the main reasons for which ADFs and other encapsulation techniques work spontaneously.

If a block of code with effective complexity δC_{block} within an individual is present n times, then the program can decrease its effective complexity and thus increase its effective fitness by using ADFs.

Let C_{call} be the size or complexity of a call to an ADF, and C_{adf} the complexity of the overhead for an ADF definition. Let the original effective length of the complete program be C_{ej}, and the original absolute length be C_{aj}. A subfunction can then change the individual's effective fitness:

$$f_{ej} = f_j - p_c \cdot f_j \cdot \frac{C_{ej} - \Delta C_{ej}}{C_{aj} - \Delta C_{ej}} \cdot p_{dj} \qquad (7)$$

$$\Delta C_{ej} = \delta C_{block} \cdot (n-1) - C_{call} \cdot n - C_{adf2} \qquad (8)$$

As long as the change in effective fitness is positive, the individual will have an advantage in survival by using an ADF. Equation 7 motivates why a program sometimes can gain a reproduction advantage by spontaneously using modularization, and also explains why ADFs do not appear in simple problem spaces. Initial empirical investigations support this hypothesis, but a more rigid evaluation is planned in our future work.

2.4 Benefits from parsimony pressure

We have concluded, in the previous sections that a Genetic Programming system has an inherent tendency to promote solutions that have a short effective complexity. We have also seen that this could sometimes conflict with our goal of adapting to a specific fitness function.

It has previously been noted that a shorter overall length of an evolved program seems to results in a program with more generic behavior (Kinnear 1993, Tackett 1993). This could be made intuitively reasonable by many different examples. For instance, let us say that we want to perform symbolic regression of a function with the following fitness cases.
fn(3)=9, fn(0)=0, fn(2)=4, fn(1)=1
Two possible functions with maximal fitness' are:
First solution:
if x=0 then fn= 0 else, if x=1 then fn=1, else, if x=2 then fn= 4, else, if x=3 then fn=9, else, fn=0
Second solution:
fn=x*x
The second solution is shorter than the first and it behaves more uniformly for a larger set of input/output pairs, than the first solution. It could be argued that the second solution is more generic. This is the so called principle of Occam's Razor. In principle a solution has a greater probability of being general if it is shorter, provided that the functional and terminal set is not biased in an *unwanted way*. If the first solution in the example above was included as one of the

functions in the function set, then the function set could be regarded as biased in an unwanted way. The Principle of Occam's Razor can be formalized and put into a mathematical framework by algorithmic information theory and the Solomonoff-Levin distribution. For an excellent introduction to the relation between complexity and machine learning, see (Li 1990). Normally when we evolve an algorithm with a genetic programming system, or when optimizing parameters from data, we want to be able to apply the solution to a much wider set of inputs than the ones given by the fitness cases. We thus want a solution that is as generic as possible and - in analogy with above - we could say that we want a solution that is as short as possible or has the lowest complexity.

The pressure towards low effective complexity does not only work on the global program level, but also on the block level where short blocks have a higher probability of proliferation, c.f. the Schema Theorem. A genetic programming system can be regarded as employing a divide-and-conquer strategy towards the goal of finding a good solution with low complexity. This general problem solving strategy of "divide-and-conquer" with a continuous pressure toward elegant and generic sub-solutions could be one of the reasons for which a genetic programming system succeeds in reasonable time in such a broad set of domains.

The effects of this pressure should be balanced to avoid unwanted effects such as the inability to obtain a perfect fitness value, where one balancing factor could be an external pressure applied on the absolute size of the individual.

Notice that there is an important difference between a program with a short effective complexity and one with a short absolute complexity. The generalization properties of the program with short effective length could be decreased by introns that are not global introns. There might be blocks of code that only act as introns with the current fitness cases.
Example: (IF (< X 4) (X*X) (+ 0 0))
This program has a perfect score for the fitness cases above but will not give the desired result for input values above four. This example provides a motivation for applying external parsimony pressure, because it could remove the local intron. On the other hand, an external parsimony pressure could further increase the probability of unwanted effects. We propose to have a parsimony pressure that is balanced and variable during the evolution of the population. For instance, it could increase towards the end of the training session.

3 Empirical Results

In this section, we present a method for empirically measuring absolute complexity and effective complexity. We briefly present an example of these measurements and show how they support the hypothesis that there exists a compression pressure in the system.

The example is symbolic regression of a function using a polynomial with large constants. We have a set of ten fitness cases with input/output pairs taken from this polynomial function and we would like to evolve the function in the language of the register machine. For a more complete description of this experiment see (Nordin 1995b).

The register machine used performs arithmetic operations between a small set of registers. All instructions are coded as 32 bits. An instruction defines the destination registers, the two operands and the arithmetic operation to be used. One of the two operands can be a small constant, the other has to be another register. The operators used in this example are addition, subtraction and multiplication. All this information is stored in the 32 bits of the register. An individual consists of a continuous string of bits. Crossover is only allowed between the instructions at a locus that is a multiple of 32. The crossover operator selects two such points in two individuals and then swaps the instructions between them.

This approach enables us to cut and splice blocks of instructions into the individual without the risk of generating programs with invalid syntax. It also enables us to make a good estimation of the effective length of individuals. We do this by systematically replacing each instruction in an individual with a NOP (NoOperation) instruction that, by definition, has no effect on the state or output of the machine. If the individual still gives the same output for all fitness cases, then we know that the particular instruction substituted acted as an intron. The number of these instructions is added, and then subtracted from the absolute length to give a lower bound for the effective length of the program. This method gives a lower bound on the number of introns because it does not find higher order introns consisting of cooperating instructions such as the two instructions $a = a + 1, a = a - 1$. But higher order introns are themselves sensitive to crossover, and experiments show that the small proportion present in the first generations is rapidly substituted by the same number of first order introns. The estimation of effective length is thus close to the actual figure. In addition to crossover a mutation operator toggles bits in the individual with a certain probability. The

selection operator in the examples below is fitness proportional selection. We have tried different selection operators resulting in different strength and dynamics of the system but with the same general results. The average standardized fitness during evolution of the function, plotted in Figures 2 and 3, shows the evolution of absolute length and effective length in the same experiment.

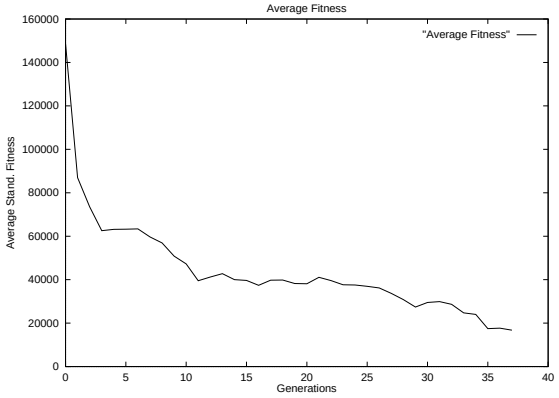

Figure 2: Average fitness, population size 2000.

Evolution of lengths starts from a point defined by the average initialization length. From generation 2 to 7, the rapidly decreasing fitness is the dominating term in the expression for effective fitness and the change in complexity is not dramatic. When the change in fitness becomes less important, the compression pressure increases and the effective lengths decrease. Finally, the absolute length starts to grow exponentially. Note that this happens while the effective length remains small and the average fitness continues to improve. Evolution over longer time has shown that the absolute complexity continues to grow exponentially without limit.

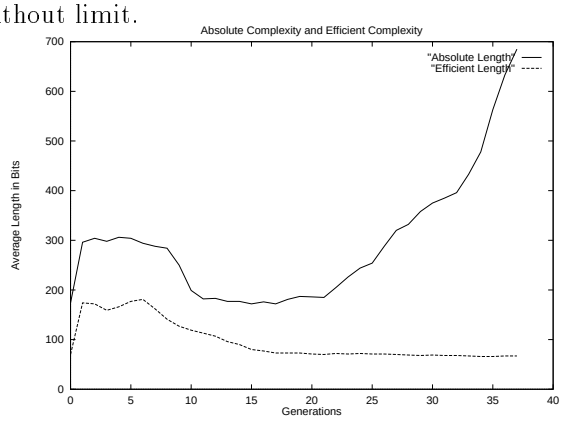

Figure 3: The evolution of the absolute and effective length.

To support the hypothesis that compression achieves its goal of protecting the individual, we have plotted the effect of crossover in different generations. Figure 4 shows the change of effects of crossover during evolution. This diagram consists of many diagrams of the same type as Figure 1 placed in sequence after each other. We can see that the absolutely dominating effects of cross-over are that either nothing happens to the fitness, or the fitness is worsened by more than 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.

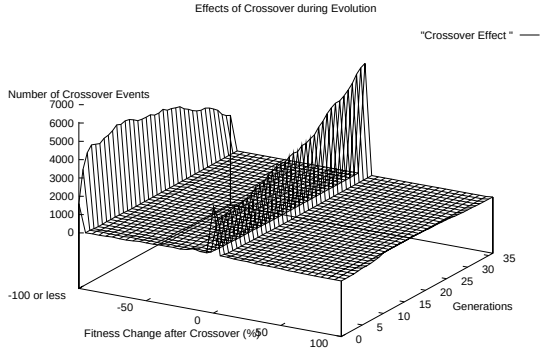

Figure 4: Distribution of crossover effect during evolution.

Figure 5 shows how a moderate constant external parsimony pressure can remove the introns completely after some generations. In this case, however, the negative effect of parsimony pressure is too strong and the system does not converge to an optimal solution. A smaller parsimony pressure would stop the exponential growth of the absolute length and force the two curves to follow each other more closely.

A tightly set maximum individual size can also be seen to increase the compression pressure, because the only way to increase the crossover target area when the maximum length is reached is to decrease the effective length. This manipulation sometimes leads to faster convergence but often the pressure becomes too high and the system fails to converge to an optimal solution.

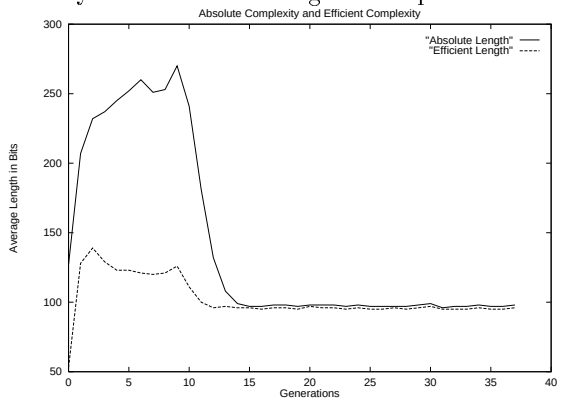

Figure 5: Effects of parsimony pressure.

4 Summary and Conclusions

We have argued for the existence of a compression pressure in evolutionary algorithms with variable length individuals. This pressure promotes short solutions with low complexity if the complexity is measured as effective complexity with introns removed. This phenomenon has both positive and negative effects, and is supported by data from evolution of computer programs. The positive effects are a more efficient search and more general behavior of the solution when used with unseen data. The negative effect is premature convergence towards a non-optimal solution. The key to the positive effects is to balance the complexity pressure according to the equation for effective fitness. This analysis should take into account a number of relevant properties, for instance representation, genetic operators, composition of fitness functions, crossover probability, generality of solutions, maximum depth or length of solutions, explicit parsimony, granularity of fitness function, initialization length and modularization.

Acknowledgement

The authors would like to thank Thomas Bäck and Sami Khuri for valuable comments to this paper. This research has been supported by the Ministry for Wissenschaft und Forschung (MWF) of Nordrhein-Westfalen, under grant I-A-4-6037.I .

References

Wolff, J.G. (1993) Computing, cognition and information compression. *AI Communications* 6(2):pp 107-127

J.P. Nordin,W. Banzhaf (1995) Evolving Turing Complete Programs for a Register Machine with Self-Modifying Code. In *Proceedings of the Sixth International Conference on Genetic Algortihms*,San Mateo, CA: Morgan Kaufmann Publishers

J. Levenick (1991) Inserting introns improves genetic algortithm success rate: Taking a cue from biology. In *Proceedings of the Fourth International Conference on Genetic Algortihms*, R.K. Belew and L.B. Booker (eds.) San Mateo, CA: Morgan Kaufmann Publishers Inc., pp 123-127

S. Forrest,M. Mitchell (1992) Relative building block fitness and the building block hypothesis In *Foundations of Genetic Algorithms 2*, D. Whitley (ed.). San Mateo, CA: Morgan Kaufmann Publishers Inc., pp 109-126.

J.D Watson,N.H. Hopkins,J.W Roberts,A.M Wiener (1987) *Molecular Biology of the Gene*, Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc.

J. Koza (1994) *Genetic Programming II*, Cambridge, MA: MIT Press.

J. Koza (1992) *Genetic Programming*, Cambridge, MA: MIT Press.

L. Altenberg (1994) The Evolution of Evolvability in Genetic Programming. In *Advances in Genetic Programming*, K. Kinnear, Jr. (ed.), Cambridge, MA: MIT Press. pp47-74.

J. Holland (1975) *Adaption in Natural and Artificial Systems*, Ann Arbor, MI: The University of Michigan Press.

P.J. Angeline, J.B Pollack (1993) Evolutionary Module Acqusition, In *Proceedings of the Second Annual Conference On Evolutionary Programming*, La Jolla, CA: Evolutionary Programming Society.

K. Kinnear (1993). Generality and Difficulty in Genetic Programming: Evolving a Sort. In *Proceeding of the fifth International Conference on Genetic Algorithms*, San Mateo, CA, Morgan Kaufmann.

W.A. Tackett (1993). Genetic Programming for Feature Discovery and Image Discrimination. In *Proceeding of the fifth International Conference on Genetic Algorithms*, San Mateo, CA, Morgan Kaufmann.

M. Li, P. Vitani (1990) Inductive Reasoning and Kolmorogov Complexity. In *Journal of Computer and System Sciences*, pp343-384

J.P. Nordin,F. Francone W. Banzhaf (1995b) Explicitly Defined Introns in Genetic Programming. Submitted to the GP workshop at *Machine Learning 1995*, Tahoe City, CA.