THESIS FOR THE DEGREE OF LICENTIATE IN COMPLEX SYSTEMS

Dynamics and Performance of a Linear Genetic Programming System

FRANK D. FRANCONE

Department of Energy and Environment

Division of Physical Resource Theory

CHALMERS UNIVERSITY OF TECHNOLOGY

Göteborg, Sweden 2009

Dynamics and Performance of a Linear Genetic Programming System
FRANK D. FRANCONE

Department of Energy and Environment
Division of Physical Resource Theory
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone +46 (0) 31-772 1000

Göteborg, Sweden 2009.

Dynamics and Performance of a Linear Genetic Programming System
FRANK D. FRANCONE
Department of Energy and Environment
Chalmers University of Technology

ABSTRACT

Genetic Programming ("GP") is a machine-learning algorithm. Typically, GP is a supervised learning algorithm, which trains on labeled training examples provided by the user. The solution output by GP maps known attributes to the known labels.

GP is distinctive from other machine-learning algorithms in that its output is typically a computer program—hence "Genetic *Programming*." The GP system documented here conducts learning with a series of very simple selection and transformation steps—modeled loosely on biological evolution—repeated over-and-over on a population of evolving computer programs. The selection step attempts to mimic natural selection. The transformation steps—crossover and mutation—loosely mimic biological eucaryotic reproduction. Although the individual steps are simple, the dynamics of a GP run are complex.

This thesis traces key research elements in the design of a widely-used GP system. It also presents empirical comparisons of the GP system that resulted from these design elements to other leading machine-learning algorithms. Each of the issues addressed in this thesis touches on what was, at the time of publication, a key—and not well understood—issue regarding the dynamics and behavior of GP runs. In particular, the design issues addressed here are threefold: (1) The emergence in GP runs of "introns" or "code-bloat." Introns in GP are segments of code that have no effect on the output of the program in which they appear. Introns are an emergent phenomenon in GP. This thesis reports results that support the hypothesis that introns emerge as a means of protecting evolving programs against the destructive effect of the traditional GP crossover transformation operator. (2) Mutation in biological reproduction is rare and usually destructive. However, we present results which establish that, in GP, using the mutation transformation operator with high probability, generates better and more robust evolved programs than using the mutation transformation operator at the low levels found in biological reproduction. (3) Finally, we return to the GP crossover operator and present results that suggest that a "homologous" crossover operator produces better and more robust results than the traditional GP crossover operator.

The GP system that resulted from the above research has been publicly available since 1998. It has been extensively tested and compared to other machine-learning paradigms. This thesis presents results that suggest the resulting GP system produces consistently high-quality and robust solutions when compared to Vapnick statistical regression, decision trees, and neural networks over a wide range of problem domains and problem types.

Keywords: Genetic Programming, Linear Genetic Programming, GP, Introns, Code-Bloat, Mutation, Crossover, Homologous Crossover, Machine-Learning.

Table of Contents

Chapter 1

Introduction

Genetic Programming ("GP") is a form of machine-learning. It comprises the automated, computerized learning of the form and contents of computer programs. In other words, it means computers writing computer programs.[1] GP is rooted in some of the earliest research performed in computerized learning by Arthur Samuel. He defined "machine-learning" as computers programming themselves (Samuel, 1963).

Like Samuel's seminal work, GP involves computers writing computer programs. However, unlike Samuel's work, GP's learning algorithm is inspired by the theory of evolution and our contemporary understanding of biology. Accordingly, instead of operating on a single program, GP operates on a population of programs. Instead of Samuel's simple, deterministic, transformations of a single program, GP uses crossover and mutation—both inspired by natural evolution.

Viewed as a learning process, natural evolution results in very long-term learning resulting from survival and reproduction of generations of populations of organisms (Banzhaf et al., 1998). Information "learned" through biological evolution is stored in DNA base pairs. Sequences of DNA base pairs act like instructions in computer programs, mediating the manufacture of proteins and the sequence of manufacture (Eigen, 1992). This program-like nature of DNA, together with the variable length structure of DNA, explains the appeal of biological evolution as a model for computer program induction (Banzhaf et al., 1998).

However, no claim is made that GP duplicates biological evolution or is even closely modeled from it. Rather, GP learning algorithms have been loosely based on biological models of evolution and eucaryotic sexual reproduction.

Charles Darwin laid out the following requirements for evolution to occur:

> . . . if variations useful to any organic being do occur, assuredly individuals thus characterized will have the best chance of being preserved in the struggle for life; and from the strong principle of inheritance, they will tend to produce offspring similarly characterized. This principle of preservation, I have called for the sake of brevity, Natural Selection (Darwin, 1859).

In other words, there are four preconditions for the occurrence of evolution by natural selection:

1. Reproduction of individuals in the population;
2. Variation that affects the likelihood of survival of the individuals;

1. We note that the term "Genetic Programming" has expanded in meaning over the years to include any learning algorithm that uses an evolutionary algorithm to derive solutions that are tree-structured, regardless whether the evolved structure is a computer program. This thesis uses the term in its original sense—the use of evolutionary algorithms to derive computer programs.

3. Heredity in reproduction (that is, like begets like); and
4. Finite resources causing competition.

GP has analogs to all of these conditions. GP operates on a population of computer programs. Variation and heredity in the population are controlled by the transformation operators—crossover and mutation. Competition amongst the programs is forced by allowing only "fitter" programs to survive and reproduce.

This thesis focuses on the second and third preconditions above—variation and heredity—and the emergent properties of a GP population that result from repeatedly selecting fitter programs over many generations for reproduction, with variation and heredity.

Thus, Chapters 4 and 5 address the tension between variation and heredity in simulated computer evolution. Both are necessary but more variation between parent and child means they are less like their parents, and vice-versa. The GP transformation operators, mutation and crossover, dictate the balance between variation and heredity in computerized evolution. Chapters 3 and 5 focus on the emergent property of "introns" or "code-bloat" and on the possibility of emergent population homologies.

This thesis is organized as follows:

1. Chapter 2 describes salient aspects of the linear genetic programming system with which this research was performed and places the remaining chapters in context.
2. Chapter 3 treats the issue of code-bloat or introns in genetic programming and concludes that introns are an emergent response of a GP population to the destructive effects of the traditional GP crossover operator. Chapter 3, starting at Section 3.3, was originally published as "Explicitly Defined Introns and Destructive Crossover in Genetic Programming," in Angeline and Kinnear (eds.), *Advances in Genetic Programming 2*, Chapter 6, pp. 111 *et seq*. MIT Press, Cambridge, MA (1996).
3. Chapter 4 addresses the effect that extensive use of the mutation operator has on the dynamics of GP runs. Chapter 4, starting at Section 4.2, was originally published as "The Effect of Extensive Use of the Mutation Operator on Generalization in Genetic Programming Using Sparse Data Sets," in *Parallel Problem Solving from Nature IV, Proceedings of the International Conference on Evolutionary Computation,* LNCS, Vol. 1141, pp. 300 *et seq.*, Springer Verlag (1996).
4. Chapter 5 proposes an alternative to the traditional LGP crossover operator called "homologous crossover" and presents results that suggest this alternative operator has emergent properties superior to the traditional crossover operator in GP. Chapter 5, starting at Section 5.2, was originally published as "Homologous Crossover in Genetic Programming," in *Proceedings of the Genetic and Evolutionary Computation Conference,* Vol. 2, pp. 1021 *et seq.*, Morgan Kaufmann Publishers, San Francisco, CA (1999).
5. Chapter 6 presents a series of studies comparing the linear genetic programming system resulting from the research in Chapters 3-5 to other machine-learning algorithms. It concludes that the resulting system provides more consistently good results than the alternative algorithms studied across a wide range of problem domains and problem types. Chapter 6, starting at Section 6.2, was originally published as part of "Extending the Boundaries of Design Optimization by Integrating Fast Optimization Techniques with Machine-Code-Based Linear Genetic Programming," in *Information Sciences Journal—Informatics and Computer Science,* Vol. 161, Numbers 3-4, pp. 99 *et seq.*, Elsevier Press, Amsterdam, The Netherlands (2004).
6. Finally, Chapter 7 summarizes and discusses the results presented in this thesis.

Chapter 2

Architecture and Core Algorithms of Linear Genetic Programming

2.1 Introduction

The purpose of this chapter is to describe, in detail, the architecture and core algorithms used in the following chapters to perform the research reported there.

In Chapter 1, we introduced Genetic Programming or “GP”. Canonical GP holds candidate solutions (programs) in a tree-based genome and the transformation operators act on tree-based genomes. By way of contrast, the GP system utilized in Chapters 3-6 is a *linear* GP system (“LGP”). LGP is distinct from canonical GP systems in that the transformation operators act on a linear—not tree-based—genome (Banzhaf et al., 1998). Our LGP system is not unique in this regard. A number of researchers have reported success using a linear genome in GP. *See, e.g.*, (Spector and Stoffel, 1996) and (Brameier and Banzhaf, 2007).

The type of LGP system we use was first proposed in (Nordin, 1994). That system was referred to there as a “compiling genetic programming system” and was, for many years (and in Chapters 3 and 4 of this thesis) referred to by the acronym, “CGPS.”

CGPS and its successor systems, “AIM-GP”[1] and Discipulus[2] are unique in that the linear genome is comprised of machine-code. That is, the population is held in memory as sequences of machine-code instructions and the transformation operators operate directly on a linear sequence of machine-code instructions. The principal advantage of this approach is an approximately two order of magnitude speedup in learning (Nordin, 1994) and (Fukunaga et al., 1998).

The linear structure is inspired by nature. DNA molecules in nature have a linear structure. DNA may be seen as 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 therefore be seen as a segment of the total genetic information (genome), which can be interpreted, and has a meaning, in itself.

In our LGP system, a *gene* is code in an evolving program representing a single machine-code instruction or a contiguous sequence of machine-code instructions that the crossover operator treats as a single unit. Such an sequence is syntactically closed, in the sense that it is possible to execute it independently of the other genes. Accordingly, this method has some similarity to the gene concept in nature.

1. “AIM-GP” stands for “Automatic Induction of Machine-Code, Genetic Programming.” It is just a different name for “CGPS.”
2. Discipulus™ is a trademark of Register Machine Learning Technologies, Inc. It is a commercial GP package based on CGPS and AIM-GP. It has been available since 1998 and is extensively documented at www.rmltech.com. Most of the work reported in Chapter 5 and Chapter 6 was performed with Discipulus™.

2.2 LGP Machine-Code Programs ("Evolved Programs")

LGP is an inductive, machine-learning algorithm. Learning is conducted on a training data set, consisting of an n-tuple for each training example, comprised of *n*–1 attributes that describe the example and a "label" (or known answer) for that example. The label for a classification-type problem would be the known class represented by the example. The label for a regression-type problem would be the known numeric output for that example.

During training LGP operates on a population of functions comprised of very simple machine-code instructions such as +, *, –, /, sqrt, power and the like. These functions are in the machine-code that is native to the CPU running the LGP system. We shall refer to these functions as "evolved programs" or "evolving programs." The evolved programs map the *n*–1 input attributes to an output that, hopefully, approximates the examples' labels.

Internal computations in the evolving programs operate directly on the CPU's registers. Thus, LGP creates programs that are "register machines." Each instruction acts on the state of the registers as of the time the instruction is executed. The *n*–1 input attributes are stored either in registers or in memory, depending on implementation. The instruction set used by LGP permits the evolving programs to access the attributes. The evolving program then evolves to access the input attributes, place them into CPU registers, operates on the values in registers, and then maps the register states to the labels.

An example may be helpful. Let `r[0]` and `r[1]` represent two CPU registers. They are initialized to zero. Assume further that LGP is configured to permit its evolving programs to operate on these two CPU registers and to use the operators, plus, multiply, and minus. Finally, assume there is only one input attribute for each training example. The one input attribute in this example is represented by x. In this situation, a six instruction machine-code program in a simple evolving LGP program could be represented by the following pseudo-code:

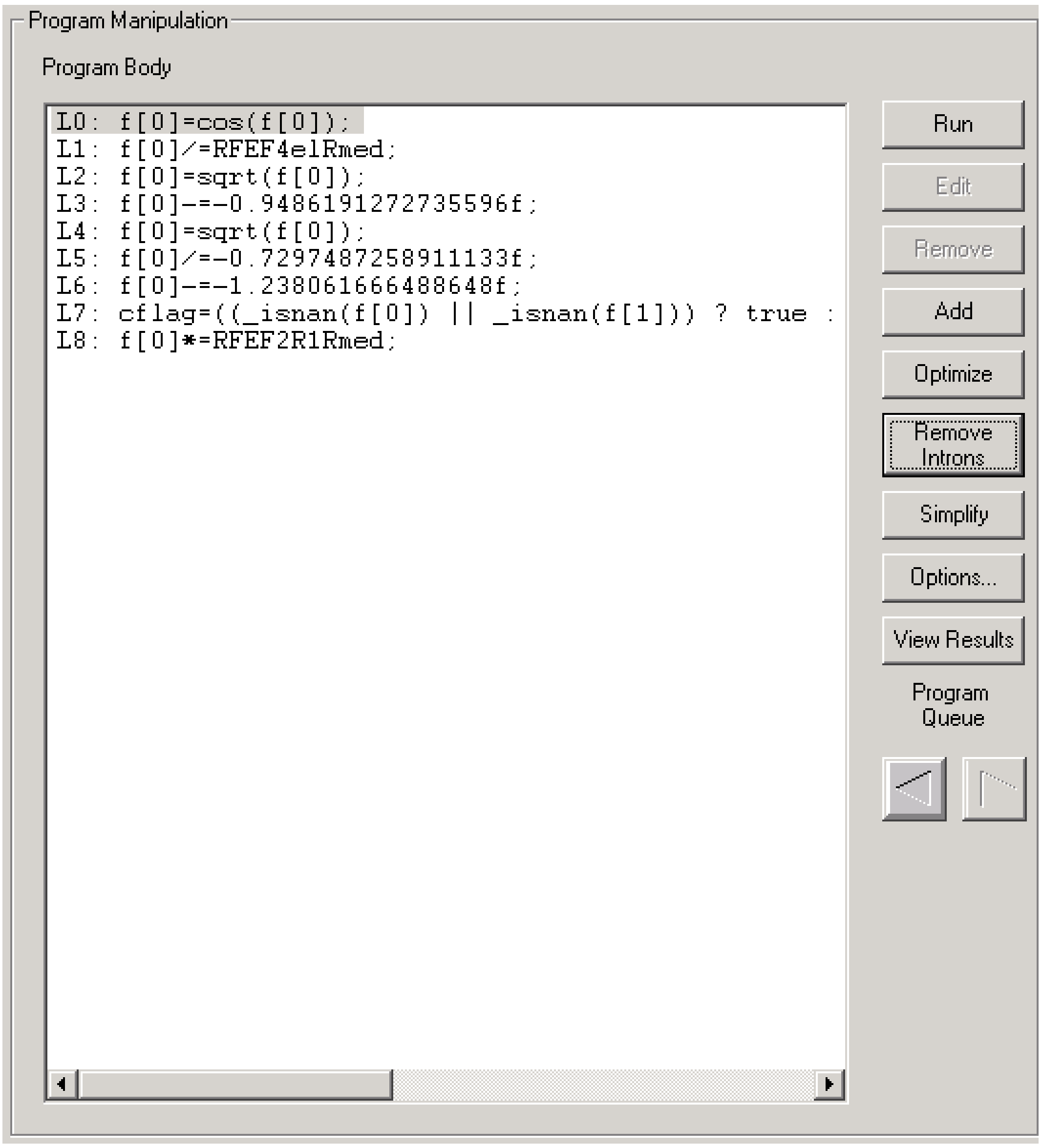
```
r[1]=r[1]+x
r[1]=r[1]-1
r[0]=r[1]*r[1]
r[1]=r[0]*r[1]
r[0]=r[0]+r[1]
Output=r[0]
```

This example program uses two CPU registers to represent a functional mapping of x to an output. The function in this case is the polynomial:

$$Output = (x-1)^2 + (x-1)^3 \qquad (2.1)$$

Note also that the effect of each instruction depends crucially on its position relative to the rest of the instructions in the evolved program. That is, the effect of each instruction depends on the state of the registers at the time the instruction is executed.

As a practical matter, although evolving programs are held in machine-code during an LGP run, the machine-code of the best evolved programs in the run are decompiled into high-level languages, such as C or Java for further use and processing. Figure 2.1 shows an example of an LGP evolved program after the conclusion of a run.



```
L0: f[0]=cos(f[0]);
L1: f[0]/=RFEF4elRmed;
L2: f[0]=sqrt(f[0]);
L3: f[0]-=-0.948619127273596f;
L4: f[0]=sqrt(f[0]);
L5: f[0]/=-0.7297487258911133f;
L6: f[0]-=-1.238061666488648f;
L7: cflag=((_isnan(f[0]) || _isnan(f[1])) ? true :
L8: f[0]*=RFEF2R1Rmed;
```

Figure 2.1
Example of a Decompiled, Evolved LGP Program

2.3 LGP Learning Algorithm

LGP is a steady-state, evolutionary algorithm using fitness-based, tournament selection to continuously improve a population of machine-code functions.[1] A single "run" is comprised of the following simple steps:

1. Randomly initialize a population of machine-code functions that map *n*–1 attributes to a labeled output (Section 2.2).

1. Some of our very early work used fitness proportionate selection (Banzhaf et al., 1998). The work in Chapter 3 was based on that system. Both tournament selection and fitness-proportionate selection have in common that evolving programs that do a better job of predicting the labels on the training data are more likely than other evolving programs to be allowed to produce progeny. Programs that do a worse job are more likely to be replaced by the progeny of better programs.

2. Select two functions from the population randomly. Compare the outputs of the two functions to the label across all n-tuples for the training targets (Section 2.2). This comparison is referred to as a "tournament." Of the two functions in this tournament, designate the function that is more "fit" (that is, the function that has a better fit to the known labels) as winner_1 of the tournament and the other as loser_1. If both perform identically, randomly select one as the winner.
3. Select two more functions from the population randomly. Perform a tournament with them as described in step two. Designate the winner and loser respectively as winner_2 and loser_2.
4. Return winner_1 and winner_2 to the population as is.
5. Apply transformation operators (crossover and mutation) to winner_1 and winner_2 to create two similar, but different evolved programs. The modified winners replace loser_1 and loser_2 in the population.
6. Repeat steps 2-5 until the run termination criterion is reached.

2.4 Structure of an LGP Machine-Code Function

During evolution, each evolved program in an LGP population is represented in a fixed-size space of allocated memory. It is comprised of the four sections, the header, the body, the footer and the buffer, as shown in Figure 2.2.

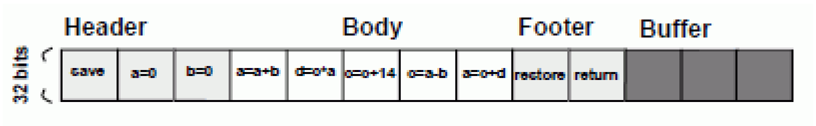


Figure 2.2
Structure of an evolving LGP machine-code program

The buffer is all memory that follows the return instruction. As the return instruction causes the function to stop computing and return control to the LGP algorithm, the only purpose of the buffer is to allow the population to hold evolving programs of different lengths. A longer program has a shorter buffer and a longer body, and vice-versa.

The header is a fixed set of instructions for all evolving programs. It performs tasks such as initializing the registers. The header does not evolve.

The footer is a fixed set of instructions for all evolving programs. It cleans up after the execution of the evolving program and returns the output value to the LGP algorithm. Like the header, the footer does not evolve.

The body of an evolving program is where all evolution occurs and where the transformation operators work.

2.5 Instruction Blocks in the Evolving Program Body

An "instruction block" is a key concept in LGP. An instruction block is one or more instructions that are "glued" together in the LGP algorithm for the purpose of applying the crossover operator. That is, the crossover operator operates at the boundaries between instruction blocks. The crossover operator has no need for any information about the instruction blocks other than the memory location of the boundaries between blocks.

Instruction blocks are rather different in LGP, depending on whether the implementation is on a RISC architecture or a CISC architecture. As the results in this thesis use LGP written for both RICS and CISC architectures, we will briefly describe both below.

Figure 2.2 is an example of an evolving program in a RISC (Reduced Instruction Set Computing) environment, such as the SUN SPARC architecture. RISC CPU's are characterized by a machine-code instruction set in which ALL instructions are exactly the same length. So each instruction block in the body of the evolving program is comprised of a single 32 bit instruction. Accordingly, each instruction block in Figure 2.2 is shown as a single square in the program body. The work reported in Chapter 3 and Chapter 4 was performed on evolving programs configured as shown in Figure 2.2.

The work reported in Chapter 5 and Chapter 6 was performed after we converted the LGP algorithm to CISC architectures (Complex Instruction Set Computing), such as the Intel Pentium. CISC machine-code is somewhat different than RISC machine-code in that CISC machine-code instructions are variable in length. To apply the transformation operators, it is important to know where the boundaries of instructions are. Accordingly, the program body in our CISC architecture implementation is slightly different than shown in Figure 2.2. An example of the LGP program body in a CISC architecture is shown in Figure 2.3.

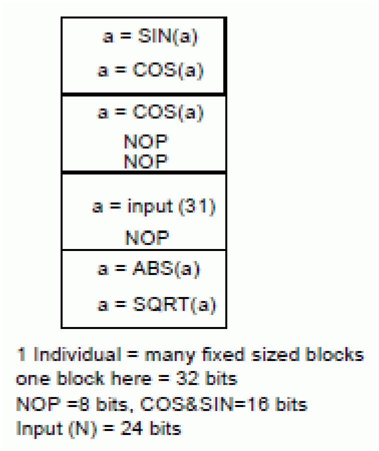


Figure 2.3

Fixed-Size Instruction Blocks in a CISC Architecture Evolving Program.

Each boxed set of instructions in Figure 2.3 comprises a 32 bit single instruction block in the CISC implementation of LGP. Because CISC instructions are variable in length, the number of instructions in each instruction block may vary, depending on the number of bits in each instruction. So, for example, a single instruction block may be composed of one cosine instruction (sixteen bits) and two NOP (No Operation) instructions (eight bits each).

As will be discussed in more detail later, understanding the structure of the instruction blocks in the program body of evolving programs is important because of the operation of the transformation operators. The mutation operator works inside the instruction blocks. The crossover operator works only at the boundaries between instruction blocks.

2.6 LGP Transformation Operators

LGP uses two types of transformation operators: crossover and mutation. A high-level description of the algorithm used for applying both operators is as follows: After a tournament has been conducted, winner_1 and winner_2 of the tournament are copied to the locations of loser_1 and loser_2. The copied winners are labeled as parent_1 and parent_2. The transformation operators are then applied to parent_1 and parent_2 of each tournament in the following steps:

1. Let *P(Crossover)* be the probability of crossover parameter for the LGP run. With *P(Crossover)*, apply the crossover operator to parent_1 and parent_2 of the tournament. If crossover occurs, then the two programs created by crossover are copied over the existing parent_1

and parent_2 and relabeled child_1 and child_2. If crossover does not occur, parent_1 and parent_2 are relabeled child_1 and child_2;

2. Let *P(Mutation)* be the probability of mutation parameter for the LGP run. With *P(Mutation)*, apply the mutation operator to child_1 of the tournament;
3. With *P(Mutation)*, apply the mutation operator to child_2 of the tournament.

Crossover and mutation are, therefore, applied independently.

The traditional LGP crossover and mutation operators used in the research reported in this thesis are described in the immediately following sections. Later in the thesis, we introduce a new, homologous crossover operator. That operator is described in detail in Chapter 5.

2.6.1 The LGP Mutation Operator

As noted above, the LGP mutation operator operates inside the boundaries of a single instruction block. In RISC architectures, it therefore operates on a single instruction. In CISC architectures, it operates on a 32 bit instruction block. In both architectures, the mutation operator results in a syntactically correct random change in the instruction block. The types of changes that may occur range from a complete random regeneration of the entire instruction block to the modification of a single parameter of an existing instruction within the instruction block (for example a change from register 1 to register 2 for one argument of a single instruction).[1]

Chapter 4 of this thesis describes our research to determine an appropriate range for the *P(Mutation)* parameter. The result was a surprising departure from the pre-existing conventional wisdom in the GP community.

2.6.2 The Traditional LGP Crossover Operator

The crossover operator is somewhat more complex to describe than is the mutation operator. The reason for this is that the research described in Chapter 5 completely changed our approach to crossover in our LGP system. Accordingly, this section will describe what we will refer to as the traditional LGP crossover operator.

In the research reported in Chapter 3 and Chapter 4, this “traditional” operator is the only crossover operator used. In Chapter 5, we introduce the homologous crossover operator and that operator will be described and evaluated in detail there. The comparison studies reported in Chapter 6 were performed with the use of both traditional and homologous crossover operators.

Traditional LGP crossover is very much like canonical GP crossover. It operates as follows: portions of the two parent programs are selected randomly, one portion from each parent program, and exchanged. In canonical GP, a random subtree is selected out of each parent program. The tree from parent_1 is then placed into parent_2 and vice-versa (Koza and Rice, 1992). Similarly, in LGP, we randomly select a sequence of one or more instruction blocks from parent_1 and a different sequence of one or more instruction blocks from parent_2. The sequences from parent_1 and parent_2 are then swapped.

Figure 2.4 is an example of traditional LGP crossover, showing the parent programs, the selected instruction block sequences, and the resulting progeny. We show a swap of the second and third instructions from parent_1 and the second instruction from parent_2 (shown at the top of the

1. Although there are differences in detail between the RISC and CISC architecture mutation operators in their internal operation, we do not regard those as salient to this thesis as the research reported deals with the overall probability of mutation, not the details of how mutation is accomplished in the RISC and CISC architectures once it is determined that mutation should occur. The reader interested in more details may consult (Nordin et al., 1998) and (Francone, 1998).

Figure 2.4). When the instructions are swapped, the resulting child programs are shown at the bottom of Figure 2.4.

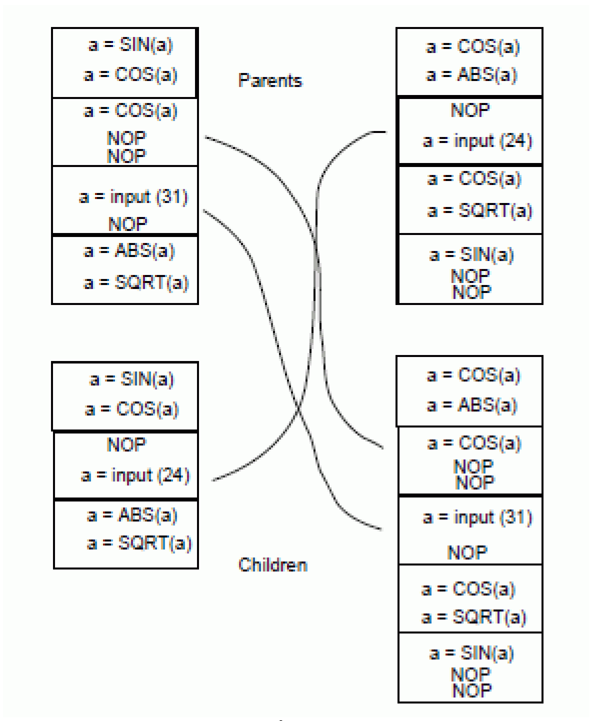


Figure 2.4

Note that this traditional crossover operator pays no heed to the relative position of the selected sequences in the two parent programs.

2.7 Conclusion

This concludes our overview description of the LGP system used in this research. With this background in mind, we now turn to the research that was performed with that system.

Chapter 3

Explicitly Defined Introns and Destructive Crossover in Genetic Programming

3.1 Chapter Preface

The materials in this chapter starting at Section 3.3, originally appeared in (Nordin et al., 1996). Some preliminary comments to tie terminology into the remainder of this thesis are in order. In this chapter, we use the terms, "CGPS" and "Compiling Genetic Programming System" to refer to an early version of our LGP system. That version is written for the SUN Sparc architecture and, for crossover, utilizes only what we refer to in Chapter 2 as the traditional LGP crossover operator. We also use the term "node" in this chapter interchangeably with the term "instruction block" introduced earlier in this thesis. Finally, we use the term "individual" interchangeably with the terms, "evolved program" or "evolving program" that were introduced earlier in this thesis.

The principal dynamic studied here is the emergence of "introns" or "code-bloat" as a response to traditional LGP crossover. The results of this current chapter lead to our work in Chapter 4 and Chapter 5.

3.2 An Example of Introns

As this chapter focuses on introns in LGP, it would be useful to begin with a definition and a concrete example of LGP introns. Simply put, introns in LGP are defined as code that is contained in an evolved program but that has no effect on the output of the evolved program. To generate the example shown below, we used a feature of our current LGP system, which contains an automated intron removal feature. It identifies introns empirically by removing one, two, and three instruction sequences from the evolved program and then determining if the removal changes the output of the program on the training data. If it does not change the output of the evolved program, that instruction or instruction sequence is labeled an intron and is removed from the evolved program. This algorithm is then applied again to the evolved program sans the removed introns. It is applied repeatedly until the removal of every instruction and instruction sequence in the evolved program results in no change in the evolved program output.

The following two C code functions are actual evolved LGP programs created by LGP on a regression data set. The first C function is shown before intron removal and the introns are marked in bold. The second C function is shown after intron removal.

To read this code, the following conventions are helpful:

1. `v` represents a vector of inputs to the function.
2. `f[n]` represents the CPU registers in which values are stored by this evolved program.

3. In1, In2, In3, and In4 are convenient labels for the values in the input vector.
4. cflag is a variable representing the state of the CPU conditional flag.
5. Bolded lines of code are introns.

3.2.1 An Evolved Program before Intron Removal

```
float DiscipulusCFunction(float v[])
{
 long double f[8];
 long double tmp = 0;
 int cflag = 0;

 f[0]=f[1]=f[2]=f[3]=f[4]=f[5]=f[6]=f[7]=0;

 double In0=v[0] ;
 double In1=v[1] ;
 double In2=v[2] ;
 double In3=v[3] ;

 L0:  cflag=((_isnan(f[0]) || _isnan(f[2])) ? true : (f[0] < f[2]));
 L1:  f[2]-=f[0];
 L2:  f[0]+=In1;
 L3:  if (cflag) f[0] = f[0];
 L4:  f[2]*=f[0];
 L5:  f[0]+=In2;
 L6:  f[0]+=In3;
 L7:  f[0]=sqrt(f[0]);
 L8:  f[0]=cos(f[0]);
 L9:  cflag=((_isnan(f[0]) || _isnan(f[0])) ? true : (f[0] < f[0]));
 L10:  f[3]+=f[0];
 L11:  cflag=((_isnan(f[0]) || _isnan(f[2])) ? true : (f[0] < f[2]));
 L12:  f[2]/=f[0];
 L13:  f[1]/=f[0];
 L14:  f[2]-=f[0];
 L15:  f[0]*=In2;
 L16:  f[0]-=In3;
 L17:  f[0]+=-1.364777803421021f;
 L18:  f[0]-=In1;
 L19:  f[0]-=In2;
 L20:  f[0]=fabs(f[0]);
 L21:  f[0]=sqrt(f[0]);
 L22:  f[0]-=-1.259177207946777f;
 L23:  f[3]-=f[0];
 L24:  f[1]+=f[0];

 if (!_finite(f[0])) f[0]=0;

 return f[0];
}
```

The program body in the above evolved program is 24 lines long. One can easily see that the bolded lines of code have no effect on program output. For example the lines of code, L9 and L13, set the value of the cflag variable. However, that value is never used in the program in a computation after `L9` or `L13`. Furthermore, `L23` sets the value of the `f[3]` register. But that register is not used in any computations that follow line 23. Finally, line 1 of the above code is an intron because the `f[1]` register is initialized to zero. Subtracting zero from the value in `f[0]` obviously makes no change in the register represented by the `f[0]` variable.

3.2.2 An Evolved Program after Intron Removal

The following C function is the same program as the evolved program shown above; but with introns (the bolded lines of code) removed by our automated intron removal algorithm.

```
float DiscipulusCFunction(float v[])

{
 long double f[8];
 long double tmp = 0;
 int cflag = 0;

 f[0]=f[1]=f[2]=f[3]=f[4]=f[5]=f[6]=f[7]=0;

 double In0=v[0] ;
 double In1=v[1] ;
 double In2=v[2] ;
 double In3=v[3] ;

 L0:  f[0]+=In1;
 L1:  f[0]+=In2;
 L2:  f[0]+=In3;
 L3:  f[0]=sqrt(f[0]);
 L4:  f[0]=cos(f[0]);
 L5:  f[0]*=In2;
 L6:  f[0]-=In3;
 L7:  f[0]+=-1.364777803421021f;
 L8:  f[0]-=In1;
 L9:  f[0]-=In2;
 L10:  f[0]=fabs(f[0]);
 L11:  f[0]=sqrt(f[0]);
 L12:  f[0]-=-1.259177207946777f;

 if (!_finite(f[0])) f[0]=0;

 return f[0];
}
```

Although the second example contains only half as many instructions as the first, the output of these two evolved programs is identical despite the removal of the twelve lines of code representing the introns.

Almost every LGP run produces introns. In the above example, fully half of the evolved instructions were introns—and this is not an extreme example. It is to that problem that the paper contained in the remainder of this chapter addresses itself.

3.3 Introduction

Biological introns are portions of the genotype (the DNA) that are not expressed in the phenotype (the organism) (Watson et al., 1987). In some eucaryotic cells, up to 70% of the genetic information is snipped out of the genome chemically before the creation of amino acids. Researchers have suggested that biological introns play some role in providing genetically safe areas for mutation and crossover (*Id*).

The operators in GP – mutation and crossover – were designed by analogy to biology (Koza and Rice, 1992). Furthermore, like DNA, GP is variable in length. One might, therefore, expect introns to evolve in GP.

But GP introns should look different than biological introns. In most GP implementations, the genotype and the phenotype are one and the same. Unlike the biological model, non-essential parts of the GP genome cannot be “snipped out”. In GP, therefore, the analog to biological introns would be evolved code fragments that do not affect the fitness of the individual. For example: `y=y+0`.

The evolution of such code fragments has been repeatedly observed by GP researchers in tree based GP; some refer to the phenomenon as “bloating” (Tackett, 1995), (Angeline, 1994). “Bloating” is the accumulation of apparently useless code in a GP population–that is, code that does not affect the fitness of the individual. Our research here confirms existence of the bloating phenomenon in linear GP structures (Figure 3.1).

In this chapter, we argue that introns appear in GP populations because they have an important function in making evolving programs more “fit” during simulated evolution. We argue that their fitness, measured by their ability to produce “fit” progeny, is increased by introns. As a result, the GP selects for the existence of GP introns in a wide variety of conditions.

Introns are, however, a decidedly mixed blessing in GP. Because GP introns are stored in evolved program structures instead of in a separate genome, a large amount of CPU time is spent calculating intron values.

In (Nordin, 1994), we began investigating introns. We devised a way to measure the intron content of genetically evolved programs using the linear structure of CGPS. In this paper, we continue that research by introducing explicitly defined introns (“EDI’s”) into CGPS.

An EDI is a structure in the CGPS system that plays no part in the fitness calculation. It does, however, affect the probability of crossover between the two nodes on either side of the EDI (Section 3.4 and Section 3.5.5, below).

By way of contrast, we will refer to introns that emerge from the code itself as “Implicit Introns” (II’s). The situation is depicted in Figure 3.2. The circular nodes affect the fitness calculation while the squares affect the crossover points. For instance the square with (14) inside acts as a string of 14 nodes during crossover but does not interfere with the fitness evaluation.

Previously, researchers have studied structures similar to our EDI’s in fixed length GA representations, (Levenick, 1991), (Forrest and Mitchell, 1992). This chapter is, apparently, the first application of EDI’s to variable length evolutionary algorithm structures or to GP in particular.

Our results suggest that EDI’s have the following effects in variable length representations:

1. Generalization may improve with the introduction of EDI’s.
2. II’s and EDI’s frequently work together, with II’s probably serving to chain EDI’s together.
3. Under some circumstances, EDI’s replace II’s in the population almost completely.
4. Like II’s, EDI’s can, and frequently do, protect an entire individual or code block against the destructive effects of crossover.
5. A combination of parsimony pressure and EDI’s allow a population to keep the structural advantages of II’s without carrying some of the computational overhead of II’s.

3.4 Definitions

We have defined EDI's and II's above. The following additional terms are needed to clarify the following discussion:

Node: The atomic crossover unit in the GP structure. Crossover can occur on either or both sides of a node but not within a node. Because our particular implementation of GP works with 32 bit machine-code instructions (see below), a node is a 32-bit instruction. A node can be comprised of either working code (see definition below) or an II. An EDI is not a node because it plays no role in the fitness calculation and because crossover occurs, effectively, within the EDI, not on either side of the EDI (see Section 3.5.5, below).

Working Code or Exon: A GP node that is not an II. Working code affects the fitness calculation of the individual for at least one fitness case.

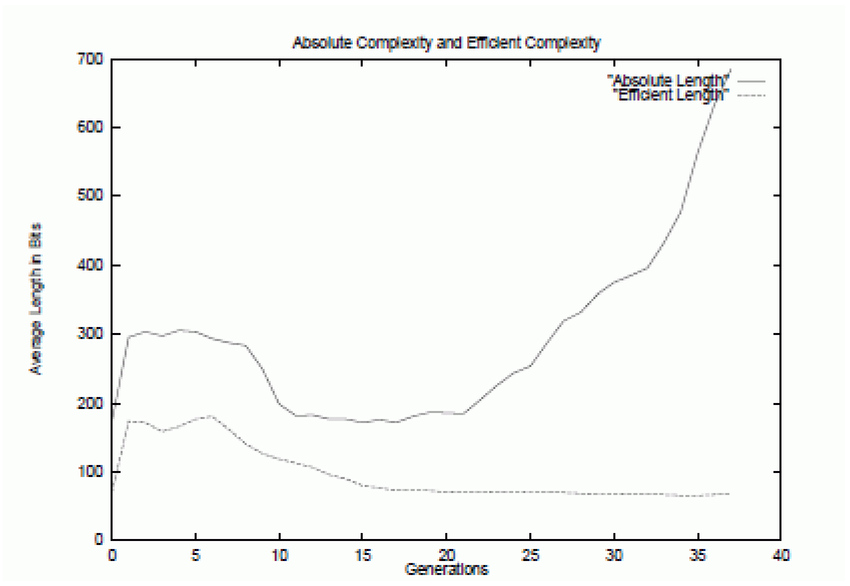


Figure 3.1
Growth of Genome Size during Evolution. Absolute and Effective Size.
(Reproduced from (Nordin and Banzhaf, 1995b).)

Absolute Size: The number of nodes in a GP individual.

Effective Size: The number of nodes in a GP individual that constitute working code–that is the number of nodes in a GP individual that make a difference in the result of the individual's fitness calculation for at least one of the fitness cases. Figure 3.1 shows the evolution of effective and absolute size during training.

Intron Equivalent Unit: (IEU). An II or an EDI with a probability of crossover that is equal to an II comprised of a single node. We will designate an II with an IEU value of 1 as II. We will designate an EDI with an IEU value of 1 as EDI(1). The purpose of defining this unit is to allow us to deal with both II's and EDI's in one designation that consistently reflects their effect on the probability of crossover.

Explicitly Defined Intron Value: Each EDI stores an integer value which is initialized randomly through three different ranges. That EDI integer value shall be referred to as an "EDIV." This value affects the probability of crossover at the EDI, as discussed below in Section 3.5.5.

See Figure 3.2 for an illustration of EDI's, II's, and working code. Section 3.7.2.1 gives further details on the behavior of EDI's and II's during evolution.

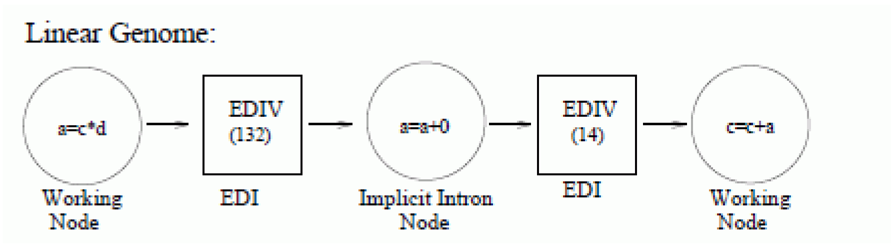


Figure 3.2
Explicitly Defined Introns, Implicit Introns, and Working Nodes in a linear Genome.

3.5 The Experimental Setup

3.5.1 The Problem

We chose a straightforward problem of symbolic regression on a second order polynomial. Large constants for the polynomial and small terminal set ranges were deliberately chosen to prevent trivial solutions.

3.5.2 Runs

We chose 10 fitness cases and tested the best individuals for generalization on 10 data elements that were not included in the training set. Each run was performed on a population of 3000 individuals. We completed 10 runs each with and without parsimony (values: 0, 1), with and without EDI's enabled, and over three ranges of initialization for EDI values (values: high, medium, low). The total number of runs was 200 comprised of 240,000 individuals. Some additional runs were performed to investigate specific issues and will be described below.

3.5.3 Implementation of GP For This Problem

The evolutionary algorithm we use in this paper is an advanced version of the CGPS system described in (Nordin, 1994), composed of variable length strings of 32 bit instructions for a register machine. The register machine performs arithmetic operations on a small set of registers. Each instruction can also include a small integer constant of maximum 13 bits. The 32 bits in the instruction thus represents 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, (The SPARC Architecture Manual, 1991), which enables the genetic operators directly to manipulate binary code. For a more thorough description of the system and its implementation, see (Nordin and Banzhaf, 1995a).

This implementation of GP makes it easier to define and measure intron sizes in code for register machines than in, for instance, functional S-expressions. The setup is also motivated by fast execution, low memory requirement and a linear genome which makes reasoning about information content less complex.

3.5.4 Intron Measurements

Many classes of code segments with varying degree of intron behavior can be identified (Nordin and Banzhaf, 1995a). For instance:

1. Code segments where crossover never changes the behavior of the program individual for any input from the *problem domain*;
2. Code segments where crossover never changes the behavior of the program individual for any of the *fitness cases*;

3. Code segments which cannot contribute to the fitness and where each node can be replaced by a NoOperation instruction without affecting the output for any input in the *problem domain*;
4. Code segments which do not contribute to the fitness and where each node can be replaced by a NoOperation instruction without affecting the output for *any of the fitness cases*; or
5. More continuously defined intron behavior where nodes are given a numerical value of their sensitivity to crossover.

The introns that we measure in this paper are of the fourth type.

We determine whether a node is an II by replacing the node with a NoOperation instruction. A NoOperation instruction is a neutral instruction that does not change the state of the register machine or its registers. If that replacement does not affect the fitness calculation of the individual for any of the fitness cases, the node is classified as an II.[1]

When this procedure is completed the number of first order introns is summed together as the intron length of that individual. Effective length is computed as absolute length less the intron length. The intron checking facility is computationally expensive but it operates in linear time in relation to the size of individuals.

3.5.5 Genetic Operators

This section gives a brief description of the evolutionary operators used, For more details on the operators and the system, see (Nordin and Banzhaf, 1995a).

Selection: Fitness proportionate.

Crossover: Two arbitrary subsegments of nodes are selected from a copy of each parent and then swapped to form the two children. If the two chosen segments are of different length, the length of the children will vary.

Crossover with EDI's: In runs using EDI's, the crossover point is selected just as if there were a chain of *N* nodes instead of the EDI with EDI-value equal to *N*. The crossover point is thus selected by examining the integer values (the EDIV) stored in the EDI's between nodes in an individual. The probability of crossover between two nodes is proportional to the EDIV of the EDI separating the nodes. The EDIV values from two parents (*k* and *n*) are transmitted to the children as follows. EDIV(*k*) and EDIV(*n*) are summed. Then the sum is divided randomly between the EDI's that appear at the crossover point in the two children. This crossover operator performs equivalent to crossing over *two individuals in the middle of two chains of II's*. We felt it was important to duplicate this phenomenon because of the frequency with which we have observed long chains of II's in our prior work. In other words, crossover acts just as if every EDI was substituted by a string of normal introns with a length defined by the EDIV of the EDI. The values transmitted to the children corresponds to the values that would have been transmitted during a crossover with a normal intron (II) segments of this length.

Mutation: Changes bits inside the 32 bits of the instruction (node), which can change the operator, the source and destination registers, and the value of any constants.

1. Note that this technique measures the presence of only first order introns. Examples of such intron segments with length one, called first order introns, are "*a=a+0*", "*b=b*1*" etc. Higher order introns can also appear, such as the second order "*a=a–1; a=a+1*". In this case, the intron segment only acts as an intron as long as the two instructions are kept together. We chose to limit our measurement in this manner because observations and theoretical argumentation support the claim that higher order introns are a small proportion of the total intron length (Nordin and Banzhaf, 1995b).

The EDIV in each EDI is initialized as a uniform random distribution between a minimum and maximum value.

3.5.6 Parsimony Pressure

We used external parsimony pressure in some of our experiments. This feature of the system punishes absolute size in an individual by adding a parsimony factor times the absolute size of the individual to the fitness expression. A parsimony factor of one means that the absolute size of the individual is added to the computed fitness. Parsimony was never applied so as to penalize EDI's, and they could thus grow unaffected by the parsimony pressure.

Table 3.1 summarizes the parameters used during training in the 200 different training runs that constitute the basis for our analysis.

Table 3.1
Summary of parameters used during training.

Parameter name:	
Objective:	Symbolic regression of a polynomial with large constants
Terminal set:	Integers in the range 0-10
Function set:	Addition Subtraction Multiplication
Raw and stand. fitness:	The sum taken over the 10 fitness cases, of the absolute value of the difference between actual and desired value
Wrapper:	None
Maximum population size:	300, 3000
Crossover Prob:	90%
Mutation Prob:	5%
Selection:	Fitness proportional selection
Termination criteria:	Maximum number of Generations exceeded
Maximum number of generations:	150,1500
Parsimony Pressure:	0, 1, 5
EIDV init value:	10-20, 10-100, 10-1000
Maximum number of nodes:	512
Total number of experiments:	200

3.6 Protection Against Destructive Crossover

3.6.1 Definitions

The following terms have the following meanings:

Destructive Crossover. A crossover operation that results in fitness for the offspring that is less than the fitness of the parents.

Constructive Crossover. A crossover operation that results in fitness for the offspring that is more than the fitness of the parents.

Neutral Crossover. A crossover operation that results in a combined fitness for the offspring that is within 2.5% of the fitness of the parents[1].

Figure 3.3 is a histogram that demonstrates the relative proportions of these three different types of crossover in a typical early generation in a typical run.

1. At least 2.5% less fitness than the parents.

The x-axis gives the change in fitness $\Delta f_{percent}$ after crossover f_{after}, where $f_{best} = 0$, $f_{worst} = Infinity$.

$$\Delta f_{percent} = \frac{f_{before} - f_{after}}{f_{before}} \tag{3.1}$$

The area over zero represents neutral crossover, the area to the left of zero represents destructive crossover and the area to the right of zero represents constructive crossover.

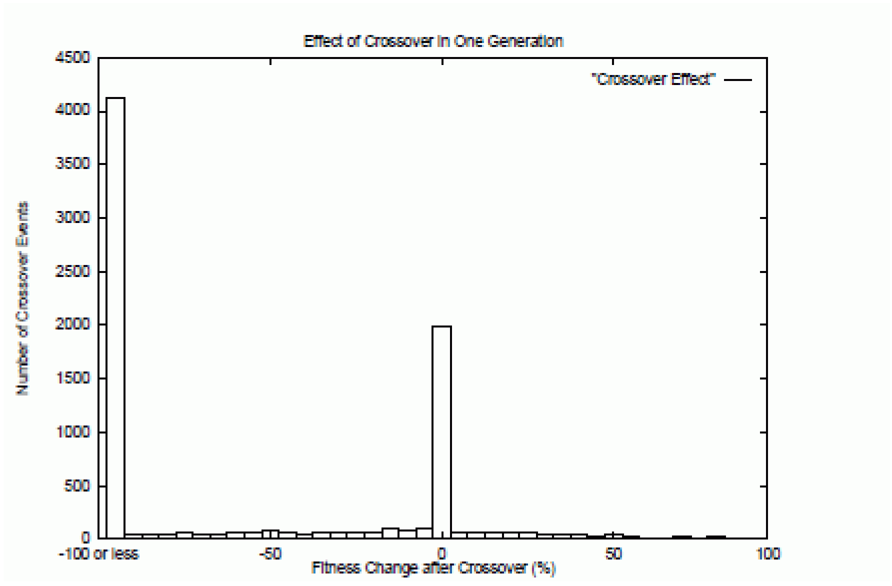


Figure 3.3
Typical Proportion of Destructive, Neutral and Constructive Crossover in an Early Generation. (Reproduced from (Nordin and Banzhaf, 1995b).)

A three-dimensional extension of Figure 3.3 is an important analysis tool for finding out what takes place during evolution. Figure 3.4 is constructed by compiling together, one figure of the same type as Figure 3.3 for each generation, thus enabling the study of distribution of crossover effect during a complete training session. In the example we see how the destructive crossover at the left decreases as the neutral crossover, in the middle, increases. The amount of constructive crossover is some magnitudes lower in this figure and is barely visible.

3.6.2 Effective Fitness and Protection Against Destructive Crossover

Using the concept of destructive crossover, we can formulate an equation describing the proliferation of individuals from one generation to the next, *c.f.* the Schema Theorem (Holland, 1975a). For more details, see also (Nordin and Banzhaf, 1995b).

Let C_{ej} be the effective size of program j, and C_{aj} its absolute size. Let p_c be the standard GP parameter giving the probability of crossover at the individual level. The probability that a crossover in a segment of working code of program j, will lead to a worse fitness for the individual is the probability of destructive crossover, p_{dj}. Let f_j be the fitness of the individual and $Av(f^t)$ be the average fitness of the population in the current generation. If we use fitness proportionate selection and block exchange crossover, then for any program j, the average proportion P_j^{t+1} of this program in the next generation is shown in Equation (3.2).

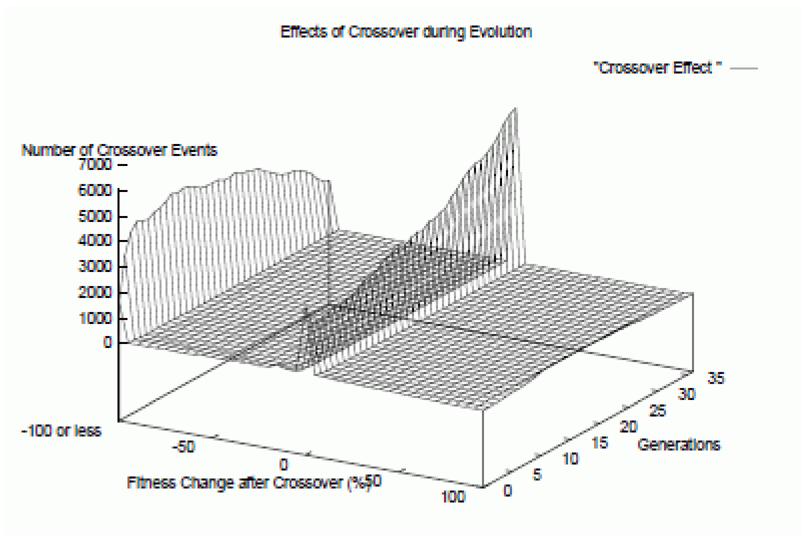


Figure 3.4
Distribution of Crossover Effects During Training. The bump over zero shows the change in the count of neutral crossover from generations one to thirty-five. The bump at -100 shows the change in destructive crossover across the same generations.
(Reproduced from (Nordin and Banzhaf, 1995a).)

$$p_j^{t+1} \approx p_j^t \cdot \frac{f_j}{Av(f^t)} \cdot \left(1 - \left(p_c \cdot \frac{C_{ej}}{C_{aj}} \cdot p_{dj}\right)\right) \tag{3.2}$$

In short, Equation (3.2) 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. 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 a term:

$$-p \cdot f_j \cdot C_{ej} \cdot \frac{1}{C_{aj}} \cdot p_{dj} \tag{3.3}$$

This could thus be interpreted as if there were a term Equation (3.3) in our fitness proportional to program size.

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

$$f_{ej} = f_j - \left(p_c \cdot f_j \cdot C_{ej} \cdot \frac{1}{C_{aj}} \cdot p_{dj}\right) \tag{3.4}$$

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

These equations suggest that there may be a number of strategies for an individual to increase its survival rate and the proportion of the next generations that contain its effective offspring. For example, it can:

1. Improve its fitness;
2. Increase its absolute size;

3. Decrease its effective size;
4. Rearrange its code segments to be less vulnerable to crossover; or
5. Take advantage of special representation to reduce the probability of destructive crossover.

In this paper, we address the later four strategies for an individual to improve its effective fitness in the following terms:

- *Global* protection of an individual against the destructive effects of crossover, refers to changes in effective fitness caused by changes in absolute or effective size.
- *Structural* protection refers to changes in effective fitness caused by rearranging the proportion of II's among code fragments to better protect the high fitness code from crossover.

Finally, EDI's provide a special representation that, we argue, can increase effective fitness by allowing the individual to improve both its global protection and its structural protection.

3.6.3 Intron Protection Analysis

Introns thus can protect against destructive crossover in at least two different ways: Global protection is protection of all the working code of an individual against destructive crossover, while structural protection is protection of portions of an individual's working code against destructive crossover. Either can improve the effective fitness of an individual.

3.6.3.1 Structural Protection Against Crossover

Let A, B, and C represent nodes of working code. Let II represent a node that is an implicit intron. Consider the following two individuals, the first with three possible crossover points and the second with only two possible crossover points:

$$A - B - II - C \tag{3.5}$$

$$A - B - C \tag{3.6}$$

The probability that crossover will occur at the A–B block of code in Equation (3.5) is 33%. In Equation (3.6), it is 50%. Therefore, in Equation (3.5), the A–B block of code has greater structural protection against destructive crossover than the same block of code in Equation (3.6). If the A–B block of code is highly fit, then the individual in Equation (3.5) has a higher "effective fitness" than the individual in Equation (3.6).

3.6.3.2 Global Protection Against Crossover

II's can also protect an entire individual from the destructive effects of crossover. Consider the following two individuals:

$$A - B - C - II \tag{3.7}$$

$$A - B - C \tag{3.8}$$

The probability that crossover will occur in a manner that will disrupt the working code (A, B, C) in the individual in Equation (3.7) is 66%. In the individual in Equation (3.8), the probability is 100%. Therefore, the working code in Equation (3.7) is better protected against destructive crossover than it is in the individual in Equation (3.8) and the individual in Equation (3.7) has higher "effective fitness" than the individual in Equation (3.8).

3.6.3.3 EDI's and Protection Against Crossover

The probability of crossover between the nodes separated by an EDI is proportional to the EDIV of that EDI (Section 3.5.5, infra).

In the following individual, therefore, the A–B code block is relatively more protected against being disrupted by crossover than the B–C block.

$$A - EDI(1) - B - EDI(2) - C \tag{3.9}$$

Likewise, the first of the following individuals is more highly protected against any of its working code (A and B) being disrupted by crossover than the second individual.

$$A - EDI(1) - B - EDI(100) - II \tag{3.10}$$

$$A - EDI(1) - B - II \tag{3.11}$$

3.6.3.4 Intron Equivalent Units (IEU's) and Protection Against Crossover

We have defined the IEU value between any two nodes as the sum of the number of II nodes and the sum of the EDIV's between these two nodes. Because our analysis suggests that both II's and EDI's should provide both global and structural protection against destructive crossover, we predict that the IEU value between two nodes should be a good measure of the amount of both global and structural crossover protection between those nodes.

We also predict that runs that use the EDI component of IEU's instead of the II component may save CPU time because the fitness calculation of an individual is unaffected by EDI's. II's, of course, consume as much CPU time as the working code. Similarly the CPU time consumed by crossover can be reduced by the less expensive computation and selection of number of nodes.

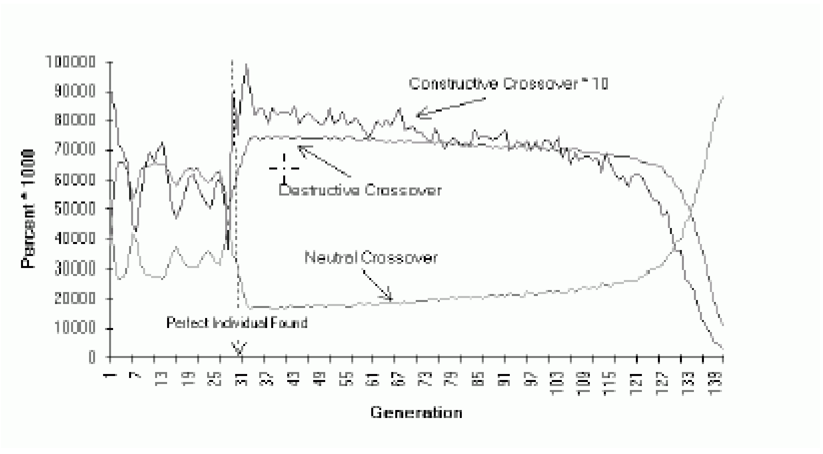


Figure 3.5
Destructive, Neutral and Constructive Crossover by Generation. Constructive Crossover Is Multiplied By 10 For Scaling Purposes. EDI enabled. Parsimony = 0.

3.7 Experimental Results

We divide this discussion into three sections. The first addresses the global effect of IEU's on protecting the entire individual from the destructive effects of crossover. The second addresses the structural effect of IEU's on protecting blocks of code from the destructive effects of crossover. The third discusses the effects of EDI's on fitness, generalization and CPU time.

3.7.1 Global Protection Against Crossover

Throughout most generations of all runs that we have measured, destructive crossover is by far the most prevalent effect of the crossover operation. (Figure 3.3 and Figure 3.4, infra). However, toward the end of most runs (with one notable exception discussed below) destructive crossover falls rapidly to but a fraction of its initial state.

Figure 3.5 shows the relative amounts of the different types of crossover by generation for a typical run. (In Figure 3.5, constructive crossover appears much higher than its actual level because, for scaling purposes, it is multiplied by 10.) At about generation 125, the proportion of destructive crossover events in Figure 3.5 starts to fall rapidly. By generation 141, it is only 10% of the total incidents of crossover. Obviously, something is protecting the population from destructive crossover.

The protection comes, we believe, from the concurrent growth of IEU’s. Figure 3.6 shows the growth of IEU’s for the same run. At about generation 125, both II’s, absolute size and EDIV increase rapidly. The IEU is simply the number of II’s plus the total EDIV. Thus, the predicted global protection effect of IEU’s on destructive crossover appears validated.

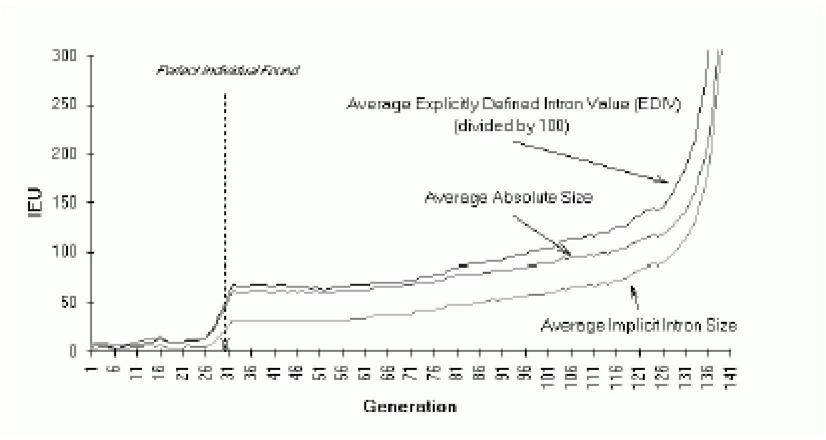


Figure 3.6
Average Absolute Size, Average Implicit Intron Size and Average EDIV Value by Generation. EDI enabled. Parsimony = 0. For scaling purposes, EDIV is divided by 100.

Table 3.2
Percent of Runs Where Destructive Crossover Disappeared.
(Zero destructive crossover events per generation.) In the last line destructive crossover did not fall but stabilized at around 70%.

EDI enabled	Parsimony	No. Runs	Population	Percent
Yes	0	10	300 each	100%
Yes	1	10	300 each	100%
No	0	10	300 each	100%
No	1	10	300each	0%

In these examples, we continue measurements after the best individual is found. Note, however, that there is no way for the system to “know” that this has happened. The system dynamics are the same as if the system gets stuck in a local optima for a shorter or longer period. The dynamics of rapid intron growth is typical for a system where fitness is hard to improve. The same phenomenon can be observed when the system is temporarily stuck at a local optima.

To test the persistence of this phenomenon, we performed the following forty runs for up to 1500 generations with the middle range of EDIV initialization (Table 3.2). In thirty out of the forty runs described in Table 3.2, destructive crossover eventually fell to less than 10% of the total crossover events. In those runs that we have examined, the fall in destructive crossover was always accompanied by a comparable increase in the average IEU value for the population.

The ten runs in which destructive crossover never fell, were runs in which there were no EDI's and there was a parsimony factor (Table 3.2). These runs are the exceptions that prove the rule. In these runs, the parsimony measure forces the number of II's to almost zero early in training. There were, of course, no EDI's in these runs. As a result, the entire population has a low IEU value throughout the run. Destructive crossover, therefore, remains very high.

On the other hand, when EDI's are enabled, destructive crossover does fall below 10% in every run – even where there is a parsimony penalty. In these runs, II's never grow rapidly—instead the EDIV's undergo the rapid growth late in training. Thus, the total IEU's (II's plus EDIV's) undergo rapid growth late in training in these runs. In so doing, they apparently inhibit destructive crossover in the same way that II's do when there is no parsimony factor (Table 3.2). In short, whenever IEU's grew rapidly, destructive crossover fell at the same time. Whenever IEU's did not grow rapidly, destructive crossover did not fall.

Table 3.2 suggests that the correlation between rapid IEU growth and an equally rapid decline in destructive crossover is 1.0. This fact confirms our prediction about the effect of IEU's on global protection against crossover.

3.7.2 Structural Protection Against Crossover

We argued above that IEU's, which include both II's and EDI's, have the theoretical ability to protect blocks of code from the destructive effects of crossover. We predict that the evolutionary algorithm may use this capability to protect blocks of code that bear the highest likelihood of producing fitter offspring. In Altenberg's terminology, we predict that IEU's may increase survivability of blocks of code with high constructional fitness and, therefore, increase the evolvability of the population as a whole (Altenberg, 1994).

3.7.2.1 Constructional Fitness and Protection Against Crossover

Assume that the code block, A–B, is working code and has a relatively high constructional fitness. Assume also that the code block K–A is also working code but has a relatively low constructional fitness. Consider the following two individuals with the following IEU configurations: $m = n \text{ and } k > j$

Parent One: *K–EIU(n)–A–EIU(j)–B*

Parent Two: *K–EIU(m)–A-EIU(k)–B*

The offspring of Parent 1 are more likely to survive than the offspring of Parent 2 because they are more likely to contain the more fit block, A–B. Thus, we would expect the A–B code block from Parent 1 to multiply through the population more rapidly than the A–B code block from Parent 2.

At first blush, this would imply that the average IEU per individual in a population should *decrease* as training continues. After all, the A–B code block from the first configuration will take its low EIU value with it when it replicates unbroken. However, there is another factor at work.

Consider two similar individuals but with $m > n \quad \text{and } j = k$

Parent One: *K–EIU(n)–A–EIU(j)–B*

Parent Two: *K–EIU(m)–A–EIU(k)–B*

In this configuration, Parent 2's offspring are more likely to survive because the highly fit block, A–B is less likely to be disrupted by crossover. As *m* increases, so does the amount of protection accorded to the A–B block. This factor would tend to increase the IEU in low fitness blocks of code as training continues.

Two countervailing factors should, therefore, be at work in setting the average IEU value for a population during the early, constructional phases of training: pressure to decrease the IEU values in blocks with high constructional fitness and pressure to increase the IEU values in blocks with low constructional fitness. However, the lowest possible value of j and k is 1. On the other hand, there is no upper limit to the value of m or n. Accordingly, selection pressure should work by providing pressure to increase the values of m and n.

We do not, however, expect the balance of these two factors to result in an exponential increase in overall IEU values during the early and constructional phase of training (Altenberg, 1994). There is yet a third factor at work. While the exponential increase strategy works well for highly fit individuals at the end of training (Figure 3.6), it would be counterproductive when constructive crossover is still likely early in training. Consider Figure 3.5 and Figure 3.6. The result of rapidly increasing IEU values is a dramatic increase in neutral crossover. In the early stages of training, individuals that have such high IEU values that they are protected globally from crossover will not survive long. Their contemporaries, who still allow for constructive crossover, will eventually surpass their performance.

One other factor must be considered. There is one significant difference in the way II's and EDI's function. Where parsimony does not altogether suppress the emergence of II's, II's are capable of chaining EDI's together. However, EDI's are not capable of chaining together other EDI's. We expect, therefore, to find differences in the behavior of EDI's and II's depending on the parsimony factor. It is also possible that we will find EDI's and II's working together in chains, instead of entirely supplanting each other.

Thus, we expect that the amount of IEU in a population will be a result of the balance of the above three factors plus the ability of II's to string together EDI's. This theory suggests that, on balance, the evolutionary algorithm should select for the presence of II's and IEU's during the constructional phase of training, but not exponentially. A finding that the evolutionary algorithm was not selecting for or against the amount of IEU in the population or that it was selecting against the presence of IEU would be inconsistent with our hypothesis.

3.7.2.2 Testing Intron Equivalent Unit Contribution To Constructional Fitness

Although the global protection effect, discussed above, is dramatic, it really only signals that the training is decisively over. Spotting the effect is rather easy, as long as there is a way to measure IEU's. Measuring the structural effect of II's and EDI's is considerably more difficult than measuring the global effect. Unlike the global protection effect, we would not expect the structural protection effect to cause easily measurable exponential increases in EDIV's or the number of II's. We were, nevertheless, able to devise two tests for our hypothesis.

Test 1. Measuring Selection for IEU Values

We discarded the absolute level of IEU's per individual in the population as a good measure of whether or not the evolutionary algorithm is or is not selecting for the presence of IEU's because of the "hitchhiking" phenomenon. Researchers have pointed out that one way useless code replicates throughout the population is by hitchhiking with adjacent blocks of highly fit code (Tackett, 1995). Our findings are not in any way inconsistent with this observation. But the hitchhiking phenomenon implies that average IEU per individual would be a poor way to measure whether the evolutionary algorithm is selecting for the presence of IEU's. Because the average amount of working code changes substantially as training progresses, we would expect the amount of hitchhiking IEU's in the population also to fluctuate. Thus, the hitchhiking phenomenon precludes average IEU per individual as a good measure of whether the evolutionary algorithm is selecting for or against the presence of IEU's.

Instead, we chose average IEU's per node in the population as our measure. In other words, we look at the average of the sum of the II's and the EDIV's per node. This measure eliminates the possibility that we are really measuring changes in IEU's caused by hitchhiking instead of measuring whether or not the evolutionary algorithm is selecting for or against the presence of IEU's. Here are the predictions we make regarding this measure:

- If our hypothesis is false and hitchhiking is the only source for IEU growth, then IEU per node should remain more or less constant or fluctuate randomly until the late stages of training.
- If our hypothesis is correct, on the other hand, IEU per node should increase during early training, but not exponentially.

We calculated average IEU per node over 80 runs with and without parsimony and with and without EDI's. We then plotted that figure, along with average best individual fitness, by generation. The results are reported below.

The tests using no EDI's and a parsimony measure were not helpful in evaluating our hypothesis one way or the other. There were no EDI's to measure and the parsimony measure suppressed the growth of II's. Since these are the only two components of IEU's, we regard any result from these runs as unhelpful either way.

The other three tests were considerably more helpful. Figure 3.7, Figure 3.8, and Figure 3.9 show the results of these three tests over 60 runs with 180,000 individuals in their populations. The results with all three parameter sets are consistent with our hypothesis. Figure 3.7 and Figure 3.8 illustrate this with the greatest clarity. Average IEU per node increases during training until the average best individual fitness over the 10 runs stops improving. At that point IEU per node drops. After that IEU per node again rises. A climb in IEU per node until best individual fitness stops improving is consistent with our prediction that the evolutionary algorithm will select for the existence of IEU's during the constructional phase of training.

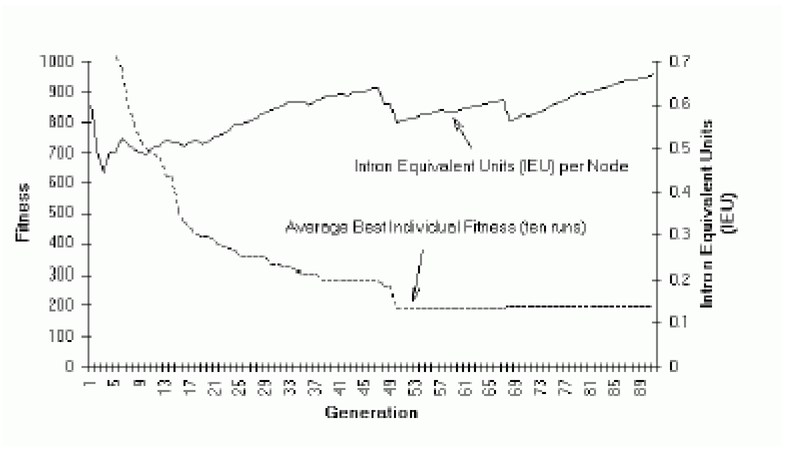


Figure 3.7
IEU per Node and Best Individual Fitness By Generation. Averaged over Ten Runs with No EDI and No Parsimony

Table 3.3
EDI parameters

EDI enabled?	Parsimony	No. Runs	Population
Yes	0	10	3000
Yes	1	10	3000
No	0	10	3000

Figure 3.9, which presents the results with EDI's enabled and no parsimony measure, also shows a clear pattern. The figure is built from data over 40 runs. The data in Figure 3.9 is scaled or normalized. The normalization point is the generation where the best individual was found.

In summary, the results of Test 1 suggest that II's and EDI's are not merely passive units during training. If they were, we would not expect to find evidence that the evolutionary algorithm was actively selecting for their presence. We conclude, therefore, that Test 1 is consistent with our hypothesis and inconsistent with the notion that II's and EDI's are only to be regarded as useless code.

Test 2. Measuring Interactions between II's and EDI's

Our hypothesis also predicts that EDI's and II's may interact with each other–either replacing each other or working together or both. If EDI's and II's are merely useless code, II's should come and go of their own accord unaffected by the presence or lack EDI's in the population.

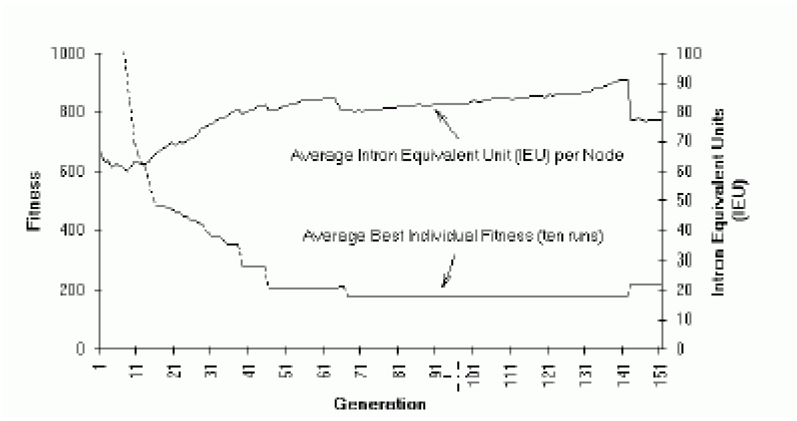


Figure 3.8
IEU per Node and Best Individual Fitness By Generation Averaged over Ten Runs with EDI Enabled and Parsimony = 1

Table 3.4
Effect of adding EDI's on the Percentage of the Average Absolute Size of Individuals that is Composed of II's. (For runs that found a perfect individual).

EDI enabled?	Parsimony	Implicit Introns	Sample Size (No. of Individuals)
No	0	53%	9,000
Yes	0	45%	9,000
No	1	25%	21,000
Yes	1	19%	18,000

We tested for such interactions in two ways. First, we measured the percentage of average absolute size of the population that was comprised of II's. This test was performed on the same runs used in Test 1. The test was performed with and without EDI's enabled. We then measured the effect of adding EDI's on the percentage of II's[1]. Table 3.4 contains the results[2].

In runs that found a perfect individual, the addition of EDI's reduced the percentage of II's in the population at the time a perfect individual was found to a significant degree.

1. Because we do not include EDI's in the measure of absolute size, the addition of the EDI's cannot affect this measurement except indirectly, by affecting the number of II's.
2. Because the point where a run finds the best individual appeared to be important in our prior reported results, we measured the change in percentage at the point in each run where best individual fitness stopped improving.

When the same figures for all forty runs was examined, the same pattern was found, the percentage of II's in the population drops when EDI's are added. However, the drop in all runs was less than half the drop for the runs that found a perfect individual. This suggests that the runs that did best (that is, runs that found a perfect individual), were the runs in which EDI's replaced II's to the greatest extent.

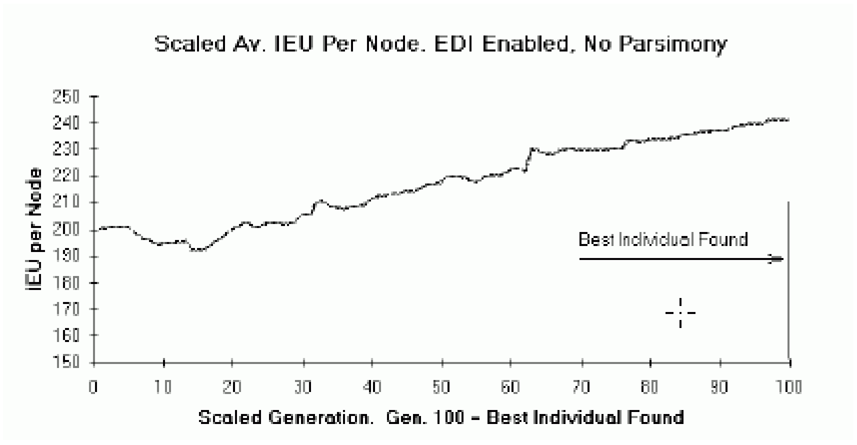


Figure 3.9

IEU per Node and Best Individual Fitness By Generation Averaged over Ten Runs with EDI Enabled and No Parsimony. Scaled and Normalized Around Generation of Best Individual.

However this data is viewed, it supports the notion that, to some extent, EDI's replace II's when EDI's are added to the population and may do so more in runs that successfully find a perfect individual. This is consistent with our hypothesis and inconsistent with the notion that II's and EDI's are merely "useless" code.

The second method we used to test for interactions was as follows. We looked for evidence that II's and EDI's work together. One such interaction is very suggestive. Refer to Figure 3.6. With no parsimony measure in this run, II's and EIVs apparently change their values in lockstep. When one changes, so does the other. When we added a parsimony factor, the result was very different. Figure 3.10 details that run using the same random seed.

Note that before the discovery of a perfect individual, the EDIV and the II values in Figure 3.10 move in lockstep. What is important here is that, despite the parsimony pressure, II's persisted in the population in a proportion more typical of a run without a parsimony factor. As soon as the best individual is found, the number of II's drops to, effectively, zero—much more typical of our parsimony runs.

We interpret Figure 3.6 and Figure 3.10 as evidence that the presence of EDI's can improve the survival value of II's until a perfect individual is found. After a perfect individual is found, II's lose that selection advantage and revert to their normal pattern in runs using parsimony. We speculate that this effect of adding EDI's is based on the ability of II's to string together long chains of EDI's and thereby rapidly increase or decrease the amount of protection accorded to various blocks of code. EDI's, by themselves do not have this ability.

In any event, were EDI's and II's merely blocks of useless code, we would not expect such an interaction between them. This evidence, also, is consistent with our hypothesis.

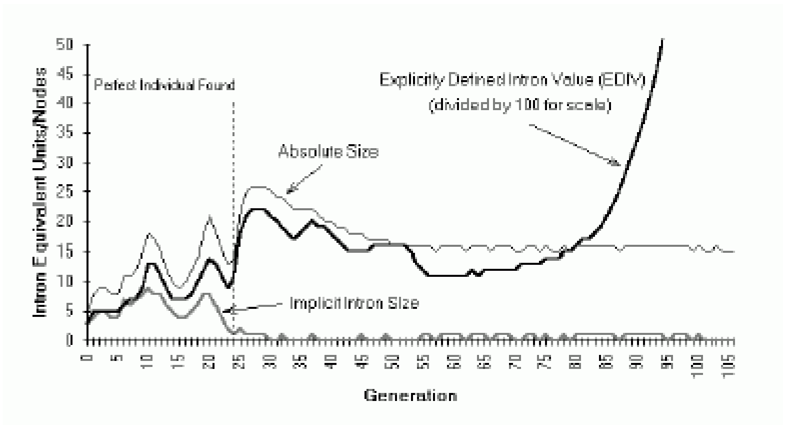


Figure 3.10
Explicitly Defined Intron Value and Implicit Intron Size for
Typical Run With EDI Enabled and Parsimony = 1.

One intriguing possibility raised by Figure 3.10 is that the average II values in a population may be a very practical way to improve training, at least where a parsimony factor is used. There are two possibilities.

First, we ran these same tests with a much higher parsimony measure–parsimony factor equal to 5. In those runs, the II's were suppressed altogether and did not demonstrate the pattern in Figure 3.10. Fitness and generalization were both worse when the higher parsimony factor was used. It may be that the pattern in Figure 3.10 means that the parsimony factor is "just right." Looking for this pattern in preliminary runs may be a good way to set the parsimony factor.

Second, if the pattern in Figure 3.10 persists for other types of problems—that is, if II size generically fluctuates until the best individual is found and then falls to close to zero, this measure may be a way to determine when to stop training. Having a measure of when the population can do no better would be an invaluable tool in GP training. We regard this as an important area for further research.

Conclusion Regarding The Structural Role of IEU's

Both of the tests we devised tend to reject the hypothesis that, in the early stages of training, II's and EDI's are only useless code that happens to be adjacent to highly fit blocks of code. Rather, the results suggest that II's and EDI's can play an important role in finding the most highly fit individual.

Proving that EDI's and II's are not useless do not, by itself, prove our position that the role played by II's and EDI's is to protect code blocks with high constructional fitness from the destructive effects of crossover. It is merely consistent with such a role. Some of the evidence is highly suggestive of such a role and the theoretical reasoning that they can play such a structural role is strong. However, we regard this as an area ripe for further research.

Table 3.5
Average Fitness of Best Individual, All Runs (small is better)

EDI Enabled	Parsimony	EDI Range	Average Fitness
Yes	1	Medium	108
Yes	0	Narrow	122
No	1	N/A	156
Yes	1	Narrow	186
Yes	0	Wide	196
No	0	N/A	280
Yes	1	Wide	290
Yes	0	Medium	354

Table 3.6
Average Generalization of Best Individual, All Runs (small is better)

EDI Enabled	Parsimony	EDI Range	Average Generalization
Yes	0	Narrow	413
Yes	1	Medium	473
Yes	0	Wide	537
No	1	N/A	542
Yes	1	Narrow	632
Yes	1	Wide	815
No	0	N/A	860
Yes	0	Medium	1176

3.7.3 Effect of EDI’s on Fitness, Generalization and CPU Time

Although the principal focus of this paper is the study of intron dynamics in GP, we additionally measured the affect EDI’s have on three measures of macro GP performance: average fitness of runs on training data at the conclusion of the runs, fitness of the best program from each run on the testing data (generalization), and average time to find a perfect individual.

To do so, we performed ten runs each at the different combinations of EDI’s (enabled/not-enabled), parsimony (enabled/not-enabled) and EDI initialization range (narrow, medium, wide). The average performances by category in rank order are shown in Table 3.5, Table 3.6 and Table 3.7 as follows:

- Table 3.5 shows the average fitness by ranked category;
- Table 3.6 shows the error of the best individual in each run on unseen testing data; and
- Table 3.7 shows the average time, in second, for runs in each category to find the best program located in the run.

The results are equivocal. Although the best two elements in each table have EDI’s enabled, on the measures of average fitness and CPU time to find best individual, the results are not statistically significant. On the generalization metric (Table 3.6), we measured the significance of the results as a two-by-two contingency table, in which the first column contains three “Yes” and zero “No” in the first column and in which the second column contains three “Yes” and two “No.” Because of the small sample size, we used Fishers Exact Test for independence of the two columns. The probability that the distribution of Table 3.6 was generated by random noise is 0.18. Thus, we can state that EDI's have an effect on generalization in GP at the 82% confidence level. Although this does not meet the 95% confidence threshold, it suggests this is an area in which more research should be performed.

We also note that runs with EDI's seem quite sensitive to the range with which the EDI's were initialized. One example illustrates possible pitfalls of training with EDI's and directions for

Table 3.7
Average Time in Seconds To Find Perfect Individual

EDI Enabled	Parsimony	EDI Range	Average CPU Time
Yes	1	Wide	31
Yes	1	Narrow	56
No	1	N/A	56
Yes	1	Medium	59
Yes	0	Medium	98
Yes	0	Wide	116
No	0	N/A	164
Yes	0	Narrow	179

further research. By far the best average CPU time to find a perfect individual was with parsimony equal to 1 and the wide initialization range for EDI's. Yet the average fitness in that same category was toward the bottom of the list. The reason is that only four of the ten runs in this category found a perfect individual. Those four runs, however, found the perfect individuals very quickly.

The reason for this apparent discrepancy may be that protection against destructive crossover is a two edged sword. While the wide range of EDI's helped these runs to find a perfect individual very quickly, it may also have helped the remaining 6 runs find local minima quickly—and get stuck there. A little less protection against crossover (the narrow and medium ranges for the same parameters) resulted in slower CPU performance but 60% of the runs found perfect individuals.

3.7.4 Future Work

We would like to extend our current results to a more canonical GP system with hierarchical crossover and tree representation (Koza and Rice, 1992). That would also shed light on any potential differences in behavior during evolution induced by representation and crossover operators. We have so far done a few initial experiments with a canonical GP system doing symbolic regression. The results indicate a distribution of destructive crossover similar to that in the system used in this paper. Figure 3.11 shows the distribution of crossover effect of S-expression based GP system doing symbolic regression over 60 generations. Figure 3.11 suggests that our results may apply to a wider domain of systems. See also (Rosca, 1995). In that regard, we plan to perform intron size measurements in a tree based GP system.

In addition, we believe that further investigation into the structural effects of IEU's is warranted, as well as investigations into continuously defined intron properties (Section 3.5.4). We would also like to study how exons and introns are distributed in the genome during evolution.

Finally, we believe that EDI's must be tested on real world problems with more intractable solution spaces.

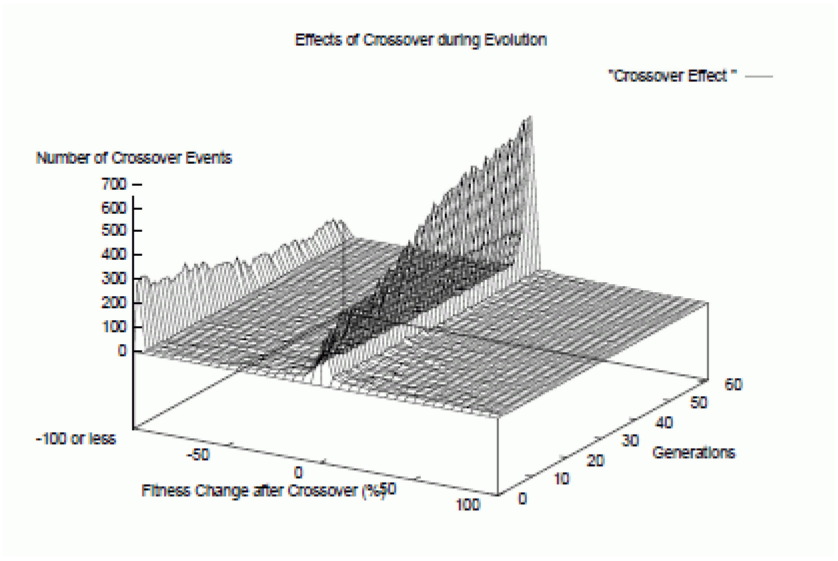


Figure 3.11
Crossover Effects In S-Expression Style GP

Acknowledgments

We would like to thank Walter Tackett for the use of his GP system, from the GP-archive. This research has been supported by the Ministry for Wissenschaft und Forschung (MWF) of Nordrhein-Westfalen, under Grant I-A-4-6037.I.

Chapter 4

The Effect of Extensive Use of the Mutation Operator on Generalization in Genetic Programming using Sparse Data Sets

4.1 Chapter Preface

In this chapter, the term "CGPS" is used to refer to our RISC-based LGP system. At the time it was published, the conventional wisdom in the GP community was that the probability of mutation in GP (*P(Mutation)* from Chapter 2) should be very low—certainly it should be applied less than 5% of the time (Koza and Rice, 1992). This was partly due to GP's early reliance on the biological metaphor for its operation. In nature mutation is a rare event.

This publication argued and presented evidence that GP performed significantly better with a high mutation rate—in the 50% plus range. It was originally published as (Banzhaf et al., 1996a).

4.2 Introduction

Evolutionary algorithms may be classified into two different groups based on their relative use of crossover and mutation. Genetic Algorithms ("GA") and Genetic Programming ("GP") tend to use very low mutation rates. In these paradigms, crossover is by far the dominant operator (Koza and Rice, 1992), (Goldberg, 1989). On the other hand, Evolutionary Strategies ('ES') and Evolutionary Programming ('EP') have their traditional focus on the mutation operator (Rechenberg, 1994), (Schwefel, 1995), (Fogel and Slayton, 1994).

Recently, the crossover operator has been challenged by researchers as being relatively ineffective in both Genetic Algorithms (Fogel and Slayton, 1994) and in GP (Lang, 1995). This paper therefore undertakes the first systematic study of the effect of changing the relative balance of mutation and crossover in GP.

We undertook this study with our Compiling Genetic Programming System ('CGPS'). Compiling GP is the direct evolution of binary machine-code. A CGPS program is a sequence of 32-bit binary machine instructions in memory. Genetic operators act directly on the 32-bit instructions. When executed, the instructions in a CGPS program cause the CPU to perform simple operations on the CPU's hardware registers.[1] By way of example, a CGPS program might, when executed, cause the CPU to add the value in register 1 to the value in register 2 and

1. For a more detailed description of CGPS, please see (Nordin, 1994), (Nordin and Banzhaf, 1995b), (Nordin and Banzhaf, 1995a) and (Nordin et al., 1996).

then place the result in register 3. Each such CGPS instruction corresponds to three nodes in a hierarchical (tree-based) GP-system. Functions of great complexity can be evolved with simple arithmetic functions in a register machine (Nordin and Banzhaf, 1995a).

4.3 Specification of the Experiments

Table 4.1 summarizes our experimental setup (Koza and Rice, 1992). Some items merit separate discussion and for those items, the experimental setup is described in the text.

4.3.1 The CGPS Mutation Operator

The CGPS mutation operator flips bits inside the 32-bit instructions that comprise a CGPS program. The mutation operator ensures that only the instructions in the function set are generated and that the register and constant values are within the predefined ranges allowed in the experimental setup. To keep the number of individuals evolved even as between mutation and crossover, we either crossover or mutate two individuals each time a genetic operator is executed.

4.3.2 The Data Sets Used In This Study

The Gaussian 3D Data Set. Gaussian 3D is an artificially generated classification problem. Both output classes are comprised of points with zero mean and normal distribution. The standard deviation of Class 0 is 1. The standard deviation of Class 1 is 2 (Jutten et al., 1995). Using 5000 training samples, the ELENA KNN benchmark error rate is 22.2%. Therefore, a CGPS hit rate of 77.8% would match the benchmark. The Gaussian problem is by far the most difficult problem reported in this paper.

Phoneme Recognition Data Set. The Phoneme recognition database requires a machine-learning system to classify two sets of spoken vowels into oral or nasal vowels. Using 5404 training samples, the KNN benchmark error rate is 14.2% (Jutten et al., 1995). Therefore, a CGPS hit rate of 85.8% would be match the benchmark. This problem is the next most difficult problem reported in this paper.

IRIS Data Set. The IRIS database requires a machine-learning system to classify measurements from photos of Irises into three different types of Irises. Using 150 training samples, the KNN benchmark error rate is between 0% and 7.3% (95% confidence interval) (Jutten et al., 1995) (p. 35-6). Therefore, a CGPS hit rate of 92.7% would match the benchmark. The Iris data set is by far the easiest of the problems attacked here. It turned out to be almost trivial for CGPS to solve.

Sparse Data Sets. We made each of the above problems more difficult for CGPS by withholding between 50% and 98% of the data upon which the benchmarks were calculated. On Phoneme and Gaussian, we gave our system only 100 data points for training and another 100 for testing rather than the over 5000 data points used for the KNN Benchmarks (Jutten et al., 1995). For the Iris problem, we used 75 training examples as opposed to the 150 training samples for the benchmark (Jutten et al., 1995).

Table 4.1
Experimental Specification. EDI’s were enabled on half of the runs.

Description	**Value**
Objective:	Classification of data sets from ELENA Database
Terminal set:	Integer constants initialized from 0 - 255
Function set:	Addition, Multiplication, Subtraction, Division
Number of Hardware Registers:	One more than the number of classes in the data
Number of Fitness/Testing Cases:	100/100
Wrapper:	None
Population Size:	3000
Selection:	Tournament. 4/2
Maximum Individual Size:	256 instructions.
Total Number of Runs:	480 (Gauss), 240 (Phoneme), 240 (Iris)

4.4 Other Important Features of the System

Variable Parameters/Number of Total Runs. We varied three of the parameters from run to run. These parameters were: Parsimony Factor (0, 0.1, 1), EDI’s (enabled/not enabled), and crossover/mutation Mix (5%, 20%, 50%, 80% mutation). There are twenty-four different combinations of the above parameters (Table 4.1). We ran each such combination on ten different random seeds. As a result, we conducted a total of 240 runs per problem set.

Measuring Fitness. For our fitness function, we selected one hardware register as the output register. We also arbitrarily assigned a target output value to each class (the ‘Class Target Value’). If a particular fitness case should have been classified as being a member of Class n, then the fitness of the individual for that fitness case is the absolute value of the difference between the value in the output register and the Class Target Value for class n. The sum of these absolute deviations over all fitness cases was the fitness of the individual.

Measuring Hits. We counted a prediction as a ‘Hit’ if the absolute value of the difference between the output register and the Class Target Value was less than 50.

Measuring Generalization. We assessed generalization by separating our data into training and testing sets. Each generation, the best individual on the training set is evaluated on the testing set and the results of that test are preserved.[1]

Terminating a Run. We have previously reported that where destructive crossover falls to below 10% of all crossover events, all effective training is over (Nordin et al., 1996). Accordingly, we monitored the rate of destructive crossover during training. When destructive crossover fell to 10% of the total crossover events for any given generation, we terminated the run. This approach saves a substantial amount of CPU time. Otherwise, we terminated a run at 200 generations.

1. While our method does provide much useful information about the generalization capabilities of our system (Masters, 1995) (pp. 335-376), the use of a third data set would improve it. The ability of the best generalizer from a run to generalize on this third data set would measure the performance of that run more accurately. We plan to implement this capability in the future.

4.5 Results

We use two measures of how much mutation affected the performance of our CP system—the mean of all runs for the hit-rate on the testing data and the probability that a particular mutation rate would result in a particularly good result on the testing data.

4.5.1 Results on the Iris Data Set

We report the results on the Iris data set here because they are so straightforward. The Iris problem was very easy for CGPS. Almost all runs matched or exceeded the benchmark—most of the runs did so by the end of the first generation. On this data set, changes in the mutation operator had no measurable effect on generalization. The remainder of this Results section applies only to the more difficult Phoneme and Gaussian data sets.

4.5.2 Effect of Varying Mutation Rates on the Mean Testing Hit-Rate

Table 4.2 reports the effect of varying the mutation rate on the generalization capabilities of CGPS. The measure of generalization capabilities used is the performance in percentage of hits on the testing set as a percentage of the ELENA KNN Benchmark.

Table 4.2
Effect of Varying the Mutation Rate on CGPS Generalization, including a Statistical Analysis: C.C.: Correlation Coefficient, S.S.: Statistical Significance

Mutation Rate	Gaussian 3D	Phoneme	Both Problems
5%	72.3%	90.8%	81.5%
20%	75.3%	90.6%	82.9%
50%	75.7%	91.5%	83.6%
80%	73.4%	91.4%	82.4%
C.C.	0.199	0.111	0.157
s.s.	99%	92%	99%

Thus, the effect of changing the mutation rate from 5% to 50% on the mean of 480 runs varies from a 0.8% to a 3.4% improvement on the testing set, depending on the problem. This effect is both important and statistically significant. Important, because in higher ranges small changes in generalization are disproportionally difficult to achieve and often make the difference between a successful prediction model and a failure. The result is also statistically significant. Table 4.2 shows the correlation coefficients and statistical significance level over all 480 runs for the correlation between the negative of the square of the mutation rate and the mean generalization capability over all 480 runs on both the Gaussian and Phoneme problem sets.

4.5.3 Effect of Varying the Mutation Rate on the Probability of Evolving Outstanding Runs

In real world applications, the point is not to evolve a lot of runs with a somewhat higher mean. The point is to evolve one or a few runs where the best individual's generalization capability is outstanding. Our results strongly suggest that increasing the mutation rate dramatically increases the probability of generating such outstanding runs. Table 4.3 shows the proportion all runs that are in the top 5% of runs by mutation rate.

Table 4.3
Percentage of Top 5% of Runs Comprised of Various Mutation Rates. 480 Runs.
Phoneme and Gaussian Data Sets.

Mutation Rate	Gaussian 3D	Phoneme	Total
5%	0%	0%	0%
20%	33%	8%	21%
50%	58%	58%	58%
80%	8%	33%	21%

This effect is statistically significant. If we convert the “Total” column from Table 4.3 into frequencies for the 50% mutation rate versus all other mutation rates, we get a two-by-two contingency table. The Chi-squared statistic for this table is 139 and, on one degree-of-freedom, the probability of Chi-squared is 0.0. In addition, Table 4.4 presents the correlation coefficients and the statistical significance levels of the coefficients for the top 5%, 10% and 20% of runs.

Table 4.4
Correlation Coefficients and Statistical Significance Levels for Correlation Between Mutation Rate and the Probability that A Run Will Be Among the Best Runs. 480 Runs. Gaussian and Phoneme Data Sets.

	C.C.	S.S.
Top 5% of Runs	0.887	99%
Top 10% of Runs	0.760	95%
Top 20% of Runs	0.828	99%

Table 4.3 somewhat overstates the effect of higher mutation rates on the actual probability of a run being in the best 5% of runs. Simply put, it took longer for higher mutation runs to find a best generalizer than it took lower mutation runs. Table 4.5 shows the increases in the probability of a run being in the top 5% and the top 10% of runs adjusted for this difference in CPU Time.

Table 4.5
Effect in increasing Mutation Rate On Probability of A Run Being One of the Best Runs. Adjusted for CPU Time. Factors of Increase in Probability of Run being in Top 5 and 10%, respectively. 480 Runs. Gaussian and Phoneme Data Sets.

Change in Mutation Rate	Factor 5%	Factor 10%
From 5% to 20%	1.8	1.7
From 5% to 50%	3.4	3.0
From 5% to 80%	1.6	2.3

A factor of 3.4 in Table 4.5 means that, adjusted for CPU time, a run with a 50% mutation rate is 3.4 times more likely to be in the top 5% of runs than a run with a 5% mutation rate. Put another way, one effect of higher mutation rates is to increase the size of the high generalization tail of the distribution of multiple runs.

4.5.4 Effect of Varying the Mutation Rate on Indicia of Training

Introduction. Increasing the mutation rate profoundly changes the way a CGPS run trains. The purpose of this section is to describe some of those changes.

Table 4.6 sets forth the effect of various mutation rates on various training indicators. We discuss each of these items below.

Table 4.6
Various Training Indicators as a Function of Mutation Rate. Mean of 480 Runs: Introns As Percent Of Total Instructions; Effective Size of Individual; Number of Generations to Best Generalizer / Number of Generations to Termination of Run

Mutation Rate	Introns	Effective Size	Best Generalizer	Run Termination
5%	74%	7	12	151
20%	68%	12	22	167
50%	63%	12	24	176
80%	58%	9	26	175

Introns As a Percentage of Total Instructions. First order introns are evolved single instructions that have no effect on the fitness calculation (Nordin and Banzhaf, 1995b). We have frequently observed first order introns in our previous work (Nordin et al., 1996). A typical first order intron is:

$$register1 = register1 + 0 \tag{4.1}$$

During training we measured the percentage of all instructions in the population that were first order introns at the time the best generalizer was found. We found that increasing the mutation rate greatly reduces the proportion of the entire population that is comprised of introns (Table 4.6). Our previous work suggests that changes of this magnitude shown in Table 4.6 are related to important changes in the training of a GP run (Nordin et al., 1996).

Mean Individual Effective Size. Effective size is the number of instructions in a GP individual that have an effect on the fitness of the individual (Nordin et al., 1996). We measured the effective size of all individuals in each GP run at the time the best generalizer was found and present the average for the population in Table 4.6. Here, raising the mutation rate significantly increases the average effective size of all the individuals in the population.

Mean Generations To Best Generalizer. Increasing the mutation rate effectively doubles the number of generations before a run locates the best individual generalizer (Table 4.6).

Mean Generations To Run Termination. Increasing the mutation rate increases the number of generations that it takes for a run to terminate (Table 4.6). In our system, this is mostly a measure of the time it takes for the number of introns to multiply so much that effective training is over (Nordin et al., 1996). This measure is, therefore, consistent with our observation above that there are more introns in low mutation rate runs (Table 4.6).

4.6 Discussion

Increasing mutation rates is a powerful way to improve the generalization capabilities of GP. Over a wide variety of parameters and over 480 runs, a 50/50 balance of mutation and crossover consistently performed better than the low mutation rates traditionally used in GP. However, higher mutation rates should only be expected to work on difficult data sets. As the data sets that we studied increased in difficulty, the effect of higher mutation rates also increased. The data sets we studied got more difficult in the following order: Iris, Phoneme, and Gaussian. The effect of mutation on the mean of the generalization capabilities increases in the same order as follows: 0%, +0.8%, + 3.4%.

That said, the mechanism by which the mutation operator acts to improve generalization is not entirely clear. Several factors, however, point to increased diversity as a factor.

Diversity and Introns. Increasing the mutation rate reduces the number of introns (Table 4.6). Consider the typical first order intron described in (4.1). Now, imagine the effect of the possible mutations that could be applied to (4.1). Changing the operator from “plus” to “times” or “minus” will convert the intron into working code. Changing the constant or changing the register of the argument or the result are also likely to convert the intron into working code. In short, mutation tends with high probability to destroy typical first order introns by converting them into code that affects the fitness calculation.[1]

Diversity And Effective Length. Table 4.6 also shows that higher mutation rates are associated with longer effective length in the entire population. Effective length is a measure of the number of instructions in an individual that affect fitness. Longer effective length could easily be a reflection of a constant infusion of new effective code into the population. Such an infusion could easily be the result of the mutation operator converting typical introns, such as (4.1), into effective code. Of course, such an infusion of fresh genetic material would tend to maintain the diversity of the population longer.

But for this mechanism to supply a continual flow of new genetic material into the population, the supply of introns in the population must somehow replenish itself. Otherwise, the mutation operator in high mutation rate runs should rapidly exhaust the population’s supply of introns. But Table 4.6 shows that introns are merely reduced, not eliminated. Our previous research strongly suggests that introns are formed by a GP population in response to the crossover operator (Nordin et al., 1996). The mechanism that suggests itself then is that crossover creates introns and mutation changes them into fresh genetic material. That the mutation operator works best at a 50/50 balance of crossover and mutation suggests the balance between crossover and mutation is a key to maintaining the flow of new genetic material into the population.

Diversity And Length of Effective Training. The effect of mutation on length of training is also consistent with our diversity explanation. Higher mutation runs continue to evolve better generalizing individuals for almost twice as many generations as do lower mutation runs, see Table 4.6. Of course, higher diversity in the population would be expected to cause such an effect. This observation hits at a central problem in GP—GP is plagued by premature convergence. That is, GP populations often lose their ability to evolve before the problem is solved. Increasing the mutation rate makes the runs continue to evolve for about twice as long (Table 4.6)—that is, a high mutation run maintains its ability to evolve for longer. This observation explains the better results in high mutation runs on difficult problems—with more time to explore the search space, such runs did better. This observation also explains why higher mutation rates did not affect the IRIS results. CGPS solved the IRIS problem almost immediately—most runs equalled or exceeded the benchmark by the end of the first generation. Simply put, CGPS did not need the extra evolvability lent by high mutation rates to solve the IRIS problem.

Conclusion. In conclusion, many factors point to the improved diversity of the population as a primary candidate for further research to explain how mutation improves the generalization in

1. It is possible to imagine first order introns that would be resistant to mutation under the correct circumstances. An example of such an intron would be:
`registerl = register^ >> register3(2);` “Shift-right” `(>>)` is effectively a division by powers of 2. In this example, mutations that change argument registers or the content of argument registers are less likely to have effects on fitness than the similar changes in the typical intron as shown in (1), provided values in the argument registers are in certain broad ranges. In short, a type (2) intron may be relatively more resistant to mutation than a type (1) intron. We have, indeed, observed these type (2) introns to proliferate heavily in individuals from runs with very high mutation rates. We have never observed such introns in our previous work in low mutation runs. This new type of intron is an area in which further research is suggested.

GP runs. One other area for future research would be to incorporate some of the ES and EP type mutation strategies into GP mutation operators.

Acknowledgments

We would like to thank the ELENA partners for the invaluable work that they undertook in assembling the ELENA databases and in their analyses of the various data sets therein. We would also like to acknowledge partial support from the Ministerium fur Wissenschaft und Forschung des Landes Nordrhein-Westfalen, under grant I-A-4-6037.I.

Chapter 5

Homologous Crossover in Genetic Programming

5.1 Chapter Preface

5.1.1 Terminology and Prior Publication Reference

This chapter, starting at Section 5.2, uses the term "AIM-GP" to refer to our CISC-based LGP system described in more detail in Chapter 2. It was originally published in (Francone et al., 1999).

5.1.2 The GP Metaphor to Biological Exchange of Genetic Material

The work in this chapter was motivated by our research reported in Chapter 3 and Chapter 4 above. It was triggered by four observations:

1. The biological metaphor for the traditional GP concept for transformation operators is weak insofar as it is used to set the relations between the various operators. For example, LGP works better with a mutation rate that would be outlandishly high in an evolving biological system;
2. GP crossover methods (both canonical tree-based GP and traditional LGP crossover) are overwhelmingly either neutral (no effect at all due to exchange of introns) or highly destructive (the children are much less fit than their parents). On the other hand, biological crossover almost always results in viable offspring, with minor, but real differences from their parents;
3. Mathematical models of biological crossover suggest that its primary function is to weed negative mutations out of a population, in addition to providing small variations as between parents and progeny (Maynard-Smith, 1994) and (Watson et al., 1987). By way of contrast, traditional GP crossover is much *like* mutation—it produces large and dramatic jumps in fitness and, usually, the progeny are less fit than the parents; and
4. GP crossover is performed without reference to the function performed by the exchanged code or to the position of the exchanged code in the two parents. On the other hand, biological crossover is overwhelmingly homologous—that is, it results in the exchange of genetic material between the two contributing organisms of very similar *functional* genes.

Our hypothesis in this chapter is that the differences between traditional GP concept for crossover (canonical and LGP) and biological crossover stem from the functional differences described in point 3 above and that making GP crossover more like biological crossover would improve GP performance. In addition, we wish to investigate the intriguing possibility that a GP crossover operator that is more biological in design will work in conjunction with higher mutation rates to weed-out the negative mutations caused by a high mutation rate (Maynard-Smith, 1994).

5.1.3 An Overview of Biological Exchange of Genetic Material

We use the generic term “exchange of genetic material” because eucaryotic crossover resulting from sexual reproduction, while widely understood and embraced amongst eucayotes, is actually only one of several known mechanisms in biology in which genetic material is exchanged between two “parents” resulting in an altered genome in the offspring. Examples of such exchange in procaryotic organism (bacteria) are:

1. Hfr Conjugation. A bacterium in the Hfr state actually injects a copy of part of its genetic material into another bacterium, where partial recombination of the genomes occur.
2. Transposons. A transposon is able to insert entire genes into the genetic sequence of the recipient bacterium. Transposons are thought to be responsible for the spread of antibiotic resistance among bacteria of different species (Banzhaf et al., 1998).

Indeed, Maynard-Smith writes:

> The widespread occurrence of phages, plasmids, and transposons means that pieces of DNA are rather readily transferred between bacteria, even between rather distantly related bacteria (Maynard-Smith, 1994).

These various procaryotic exchange mechanisms, as well as eucaryotic crossover in sexual reproduction, have in common that the DNA sequences from the two “parents” strongly tend to align with each other where their base pairs are identical or nearly identical before the genetic exchange occurs between the DNA sequences. Then the exchange of genetic material occurs between the aligned DNA strands. The ability to align in this manner is a property of the DNA molecule resulting from base-pair attraction between different DNA molecules (Watson et al., 1987) and (Maynard-Smith, 1994).

No such inherent mechanism for aligning “base-pairs” exists in GP. If GP crossover is to mimic the biological genetic exchange mechanism, the alignment mechanism must be imposed by the GP system. Further, there is no reason to believe a randomized GP population will even possess the equivalent of biological genes so that alignment of comparable sections of different evolving programs will be meaningful. Thus, for a homologous crossover mechanism to work in GP, it will be essential to design a mechanism that will permit alignable sections of evolving programs to emerge from evolution. This paper on which this chapter is based, and which starts in the next section, addresses both problems.

5.2 Introduction

Crossover in GP has, in recent years, been described as highly destructive (Nordin et al., 1996) and as performing no better than mutation (Angeline, 1997). By way of contrast, crossover in nature appears robust and rarely produces lethally defective offspring.

This paper describes and tests a new form of crossover for linear GP. It was inspired by our observation that homology in a population and constraints on crossover appear to be important to the success of crossover in nature. This new operator implements a limited form of forced alignment between genomes during crossover and may encourage the emergence of positional homology in GP populations.

5.3 The Emergence of Homology

Natural crossover is highly constrained (Banzhaf et al., 1998). For example, natural crossover is strongly biased to exchanging genes that are in the same *position* on the chromosome and that express similar *functionality*. That bias arises from two features of crossover:

1. *Homology.* Crossover almost always occurs between organisms that have nearly identical base pair sequences in their chromosomes. For example, in organisms that reproduce sexually, mating occurs only between members of the same species. Within a species, there is a high degree of similarity (homology) between the genomes of the various members of that species.
2. *Base Pair Bonding*. Two strands of DNA combine into a single double helix because of base-pair bonding between the bases in the two strands. As a result, there is a strong tendency for two strands of DNA to recombine (crossover) in a manner that exactly matches the complementary base-pair sequences in the other (Watson et al., 1987). Indeed, during crossover, "complementary base-pairing between strands unwound from two different chromosomes puts the chromosomes in exact register. Crossing over thus generates homologous recombination; that is, it occurs between two regions of DNA containing identical or nearly identical sequences (*Id*)."

Homology in nature is, therefore, both *positional* and *functional*. These rigid constraints on crossover suggest that it is an evolved operator—a form of emergent order. Homology permits the meaningful exchange of functionally similar genes through the base-pair bonding mechanism. At the same time, the existence of base pair bonding tends to encourage the emergence of homology. When these biases are applied through many iterations, one can envision the evolution of emergent homology in nature. By this view, sexual reproduction in species represents the evolution of evolvability in nature (Altenberg, 1994).

5.4 Genetic Programming Homology

With few exceptions, crossover in GP is characterized by swapping code between two parents without regard to *position* of the code in the parent programs or the *function* the code performs. This is true regardless whether trees, graphs, or linear structures are used to represent the genome (Koza and Rice, 1992), (Teller, 1996) and (Nordin, 1994).

Furthermore, GP runs have no obvious *homology* in the population, at least at the beginning. After all, the programs in a population are initialized randomly—the opposite of the almost identical genomes shared by members of a sexually reproducing species. This problem is well illustrated by an example we have given elsewhere:

> "Crossing over two programs [in GP] is a little like taking two highly fit word processing programs, Word For Windows and WordPerfect, cutting the executables in half and swapping the two cut segments. Would anyone expect this to work? Of course not (Banzhaf et al., 1998)."

The reason the conclusion in the quoted material is obvious—there is no homology between Word Perfect and Word for Windows. Accordingly, although these two programs perform the same overall function in a very similar manner, they are in a sense, from different "species." As such, we do not expect them to mate successfully any more than we would expect a zebra to mate successfully with a cow, even though both animals eat grass and have four legs.

But on closer reflection, a type of homology does emerge in GP populations and does so frequently. The term, "introns" in GP has come to refer to nodes or instructions in evolved programs that have no effect on the output of the program, such as `x=x*1` (Banzhaf et al., 1998), (Nordin et al., 1996), (Soule et al., 1996) and (Soule and Foster, 1997). Code-bloat (or growth of introns as a portion of the contents of evolved programs) is a commonly remarked feature of GP runs. When code-bloat has come to dominate a population, crossover becomes a matter of swapping sequences of code that have no effect on the fitness of the individual. Such crossover is necessarily non-destructive because it is completely neutral (Nordin and Banzhaf, 1995b).

Code-bloat is a form of emergent homology, albeit of a perverse type. Where code has bloated, there is a high degree of homology between almost any two randomly chosen nodes on any two evolved programs in this population. That is, any two nodes chosen for crossover are likely to contain functionally useless code. Because crossover between two such nodes is almost always neutral, the population—dominated by introns and neutral crossover—has become a “species” in the sense that all members can mate with each other with a high probability that the offspring will not be lethally defective. This is, in effect, position independent homology.

5.5 Encouraging Useful Homology

The homology represented by code-bloat is not especially useful or even interesting. Its existence does, however, establish that homology can emerge from GP runs, depending on initial conditions. The issue presented by this paper is whether it is possible to redesign the crossover operator so that it encourages useful homology. In that regard, we need to identify a crossover operator that:

1. Encourages a GP run to evolve useful homology in the population; and
2. Exploits the evolved homology usefully.

Such a crossover operator could be expected to be less-and-less destructive as the run continues, tend to discourage code-bloat and, perhaps, improve system performance.

5.5.1 Sticky Crossover

We propose to make our program genomes “sticky,” somewhat like DNA, during crossover. We refer to this as a “homologous crossover operator” or, in lighter moments, as “sticky crossover.”

In simple terms, the homologous crossover operator only permits instructions at the same position in the genome to be exchanged with other instructions in the same position. More precisely, sticky crossover chooses a sequence of code randomly from one parent program. The sequence of code in the same *position* from the second parent is then chosen. The two segments are then swapped.

5.5.2 Sticky Crossover Dynamics

Our new operator does not actively seek out functionally equivalent segments of code in evolved programs and cross them over. So in that sense, it does not duplicate the base pair bonding of biological crossover.

Instead, our proposed operator provides a high probability that, say, the second instruction in an evolved program, will be swapped during crossover with the second instruction from another program. This may encourage the evolution of functionally similar code at equivalent positions in the genome.

Why should this be so? In (Nordin and Banzhaf, 1995b), we proposed that the probability that an individual will propagate in increasing numbers from one generation to the next depends not just on its own fitness, but on expected value of its offspring’s fitness. We referred to this aggregate measure of fitness as “*effective fitness*.” Effective fitness describes the notion that even highly fit programs will not long survive if they cannot produce fit offspring. Thus, effective fitness measures a program’s ability to be successfully replicated by means of crossover.

The effective fitness of an evolved program may be quite different for different types of crossover. For example, evolved programs containing a high proportion of code that actually affects their output have little defense against destructive ordinary crossover, even if they are highly homologous as to other individuals in the population. This is because ordinary crossover is constantly moving the functional code around in the genome. Each time that happens, positional homology among members of the population changes. So there is little chance that positional homology will evolve out of functional code when exposed to ordinary crossover.

By way of contrast, we propose that programs containing a substantial proportion of functional code and that are relatively homologous to other programs in the population will have higher effective fitness than those that are not so homologous, when exposed to the sticky crossover operator.

Assume the following about a GP population:

1. Two programs in the population are more than randomly homologous as between themselves;
2. The rest of the population is randomly homologous as to all other members of the population; and
3. The sticky exchange of homologous sequences is less likely to produce low fitness children than the exchange of non-homologous sequences, as appears to be the case in nature.

Given these premises, and all other things being equal, the two relatively homologous programs will be more likely to produce fit offspring when subjected to sticky crossover than other programs in the population. That is, *relatively homologous programs have higher effective fitness*. Because of the reproductive advantage of their children, these two programs will be more likely to propagate their genetic material through generations of evolution than randomly homologous programs.

There is another, and more emergent aspect to this analysis. When two programs that are relatively homologous are crossed over (using sticky homologous crossover), they necessarily produce children that are themselves relatively homologous to the other children *and* to the parents. In other words, the number of relatively homologous programs in the population increases after crossover of two relatively homologous individuals because sticky crossover does not alter the relative homology as between the parents and now, the two offspring. As the offspring of two evolved programs that are relatively homologous to each other (and their offspring) spread throughout the population, the descendants' relative reproductive advantage deriving from that homology spreads because there are now more homologous programs in the population derived from the same ancestors. Thus, we would predict that the homology of the overall population should also increase and that homology should appear as emergent order.

But does emerging homology contribute to *useful* fitness from a problem-solving viewpoint? We know from runaway code-bloat that emergent homology is not always useful. In this regard, however, sticky, homologous crossover is quite different than standard linear crossover. It respects the position of the swapped code; that is, it respects positional homologies.

With standard crossover, the only code that could be the cause of emergent homology is introns (useless code) because that is the only code that is probably homologous *no matter where it appears in the genome*. But with our homologous crossover operator, code that is homologous to other programs in the population need not also be homologous at other positions on the genome because sticky crossover does not move the code around. Obviously, *functional code may exhibit homologous properties when exposed to sticky crossover as long as that code only needs to be homologous in one location on the genome.*

This reasoning does not, of course, prove that useful homology will emerge. Only that we can expect homology with this new operator, to look considerably different than the perverse homology of code-bloat, that it will be homology of, at least partially functional code, and that this functional code may well evolve through normal selection to be useful code.

There is one prediction we can make from this reasoning. If useful homology emerges, it will do so because crossover is less destructive when performed between homologous elements of the population. Previous research indicates that intron growth in GP is caused in large part by the destructive effect of the crossover operator (Nordin et al., 1996), (Soule and Foster, 1997). If crossover becomes less destructive because of emerging homologies, we would predict that increasing the amount of homologous crossover in the population should decrease the rate of growth of the length of programs in the population.

5.5.3 Previous Related Work

Our approach is not entirely novel. We are aware of two previous sets of experiments in tree-based GP that added positional dependence to the crossover operator.

D'Haeseleer devised strong context preserving crossover in 1994. In that work, D'Haeseleer permitted crossover to occur only between nodes that occupy exactly the same position in the two parents. D'Haeseleer got modest improvement by combining ordinary crossover with his strong context-preserving crossover (D'Haesseleer, 1994).

Similarly, Poli and Langdon introduced one point crossover in 1997 (Poli and Langdon, 1997). One point crossover is even more restrictive than strong context preserving crossover because it only permits crossover between nodes that match in position and arity.

But homologous crossover with linear genomes is considerably different than these two prior approaches. These previous works allow very large subtrees and very small subtrees to be exchanged in crossover, as long as the base node of the two subtrees occupies the same position in the genome. By way of contrast, our homologous linear operator requires that the exchanged code sequences are very similar in size *and* that they occupy the same position in the genome. Thus, our linear operator is forced to exchange segments of code that are more likely to be similar (in semantics and, therefore, function) than the code exchanged in these two tree based systems.

5.6 Implementation

AIMGP is an acronym for "**A**utomatic **I**nduction of **M**achine Code, **G**enetic **P**rogramming." AIMGP uses linear genomes made up of native machine-code functions. It performs crossover, mutation and fitness evaluations directly on the machine-code (Nordin et al., 1998).

Our first experiments with homologous crossover in AIMGP began about two years ago at the University of Dortmund. During that time, we have become increasingly convinced of its beneficial effects.

The experiments reported here were run using Discipulus™ 1.0 software (Register Machine Learning Technologies, Inc., 1998-2009), which is a commercial version of AIMGP written for WINTEL machines running 486DX, Pentium® and Pentium II® processors. It implements over fifty native machine-code instructions from the floating-point processor in these systems.

Although we will provide some implementation details here, the implementation and operation of this software is exhaustively documented in (Francone, 1998).

5.6.1 Program Representation

During evolution, AIMGP systems hold evolved programs as native machine-code functions. Each program is comprised of the following parts: header, body, footer and a return instruction as shown in Figure 5.1.

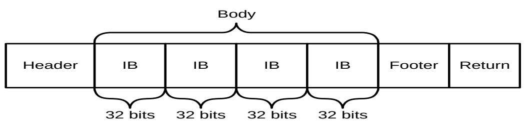


Figure 5.1
The Structure of an AIMGP Program. ("IB" Refers to instruction blocks).

The body of the evolved programs is where evolution takes place. The crossover and mutation operators operate only in the body and the output of each evolved program is calculated in the body of the program.

The body of an evolved program is composed of one or more 32 bit “instruction blocks.” An instruction block may be comprised of a single 32-bit instruction or any combination of 8, 16, or 24-bit instructions that add up to 32 bits. Crossover occurs only at the boundaries of the instruction blocks and without reference to the content of the instruction blocks.

5.6.2 Crossover And Mutation

In these experiments, we blended “ordinary” crossover and sticky, homologous crossover. In this system, some non-homologous crossover is necessary because that is the only way for genomes to change length.

Ordinary Crossover. Ordinary crossover is traditional AIMGP crossover—two-point crossover, with the selected code fragments being chosen randomly from the two parents. Figure 5.2 represents traditional (non-homologous) crossover in Discipulus™. Each block represents an instruction block. Each instruction block may contain one or more instructions. Crossover occurs at the boundary of instruction blocks.

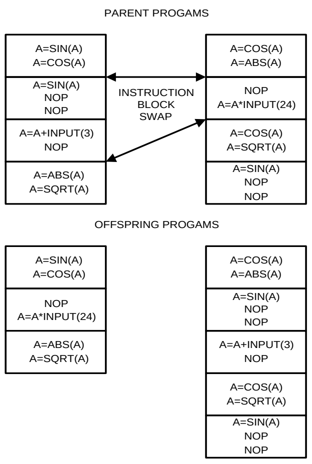


Figure 5.2
Non-Homologous Crossover in AIMGP using instruction blocks.

Homologous Crossover. We implemented homologous crossover as shown in Figure 5.3. What distinguishes homologous from non-homologous crossover is that in homologous crossover, instruction blocks can only be swapped with an instruction block at the *same position* in the other parent’s genome.

Mutation. Mutation makes a random change to a randomly chosen instruction block. There are three kinds of mutation:

- **Block Mutation**. Random regeneration of an entire instruction block;
- **Data Mutation**. Randomly pick one instruction in the instruction block and randomly change an operand; and
- **Instruction Mutation**. Randomly pick one instruction in the instruction block and randomly change its operator.

5.7 Experimental Setup[1]

We chose to test the effect of adding homologous crossover to LGP on the Gaussian 8D problem. Gaussian 8D is a classification problem comprising two classes and eight independent variable inputs. In addition to the eight 'real' inputs, we added sixteen false inputs—that is, sixteen data series containing random variables are provided to the system in addition to the eight real inputs. This makes the system engage in variable selection as well as classification.

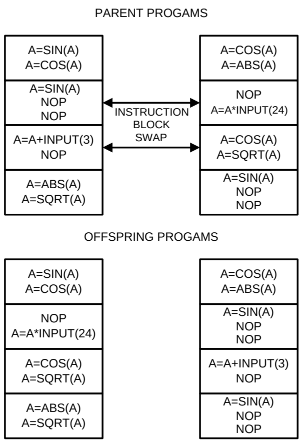


Figure 5.3
Homologous Crossover in AIMGP.

The eight 'real' inputs were generated as follows:

- For Class 0 examples, each of the inputs are normally-distributed, random values with a mean of 0 and a standard deviation of 1.
- For Class 1 examples, each of the inputs are normally-distributed, random values with a mean of 0 and a standard deviation of 2.

We divided the data into training and validation sets of one thousand examples each.

We measured the performance of a run by the best classification rate attained on the validation set (in percentage correct classifications).

To test the effect of homologous crossover, we used the default parameters of Discipulus™ with the following noted exceptions. The only parameters varied were the mutation rate and the homologous crossover rate as follows:

- The mutation rate was varied between 5% and 80% (5%, 20%, 50%, and 80%).
- The overall crossover rate was fixed at 100%. The composition of the crossover operator as between homologous (sticky) and non-homologous crossover was varied as follows: 0%, 20%, 40%, 60%, 80%, and 95%. (Any crossover that is not homologous is traditional or non-homologous crossover.)

Altogether we performed twenty-five runs at each parameter setting using a different random seed for each of the twenty-five runs. Because there were twenty-four different parameter combinations tested, we performed 600 runs to obtain the data reported here.

Each run was allowed to continue for two minutes before termination. Although two minutes may seem like a very short run for a difficult problem, this is machine-code GP. It is

1. All data, software, parameters and configuration files necessary to duplicate these experiments in their entirety may be found at (Register Machine Learning Technologies, Inc., 1998-2009).

approximately sixty times faster than typical GP systems. Accordingly, a run of two minutes on this system is comparable to a run of about two hours on a typical GP system.

Table 5.1
Best Percentage Hit Rate on Validation Set by Mutation Rate (horizontal legend) and Homologous Crossover Rate (vertical legend). Each box is average of 25 runs.

	5%	20%	50%	80%
0%	71.1	72.6	72.9	74
20%	69.5	71.4	74.9	73.9
40%	70.6	73.2	73.7	73.4
60%	68	72.4	74.5	74.6
80%	70	74.4	74.9	75.7
95%	70.8	75.3	77.6	77.4

5.8 Results

We measured overall performance by the percentage of correct classifications attained by the best individual on the validation set.

Table 5.2
Number of Tournaments before Average Length Equaled or Exceeded 500 bytes, in thousands of tournaments (000), by Mutation Rate (horizontal legend) and Homologous Crossover Rate (vertical axis). Each box is average of 25 runs.

	5%	20%	50%	80%
0%	10	11	10	12
20%	9	10	11	11
40%	9	11	9	9
60%	11	11	9	10
80%	16	18	12	14
95%	20	24	22	22

Table 5.1 reports the results of our runs cross-tabulated by mutation rate and homologous crossover rate. As is apparent, runs with high homologous crossover rates tend to perform better than runs with lower homologous crossover rates.

Table 5.2 shows the average number of tournaments that transpired before the average length of the evolved programs in the population exceeded 500 bytes in length. The more homologous crossover, the longer runs continue before they reach 500 bytes in length. This effect is pronounced at each level of mutation.

5.9 Discussion

On these data, both of our predictions are confirmed for this problem. Homologous crossover does apparently have a significant and beneficial effect on the fitness of the best individual on the validation data and on code-bloat.

5.10 Further Work

The homologous crossover operator introduced here is only the beginning of what we believe should be a concerted effort into improving the crossover operator by analogy to biological systems. For this operator, no attempt was made to guide the evolution of homology within the population other than permitting the genomes to have a degree of stickiness. Further, in our current experimental setup, only one species is able to evolve per population. Additional research is planned into mechanisms that would permit differential speciation within the population. Finally, we expect future work to look more deeply into the actual operation of the crossover operator at run-time rather than viewing only the collective output of a run as a single entity.

Acknowledgments

Peter Nordin gratefully acknowledges support from the Swedish Research Council for Engineering Sciences.

The authors would like to thank Register Machine Learning Technologies for use of the Discipulus™ software.

Chapter 6

Comparative Evaluation of a Linear Genetic Programming System using High Mutation Rates and Homologous Crossover

6.1 Chapter Preface

This chapter was originally published as part of (Francone and Deschaine, 2004). The comparison studies reported were conducted with Discipulus™ GP software, a commercial version of our machine-code LGP system. All comparative runs were conducted at the default parameters of Discipulus™. The capability of homologous crossover and high mutation rates are built into the system. The default parameters in this regard are:

1. Probability of mutation—95%
2. Probability of crossover—50%. Once it is determined to apply crossover to a tournament, then the relative proportion of Homologous and traditional LGP crossover is as follows:
   - Probability of homologous crossover—95%
   - Probability of traditional LGP crossover—5%

6.2 Linear Genetic Programming

Genetic Programming (GP) is the automatic, computerized creation of computer programs to perform a selected task using Darwinian natural selection. GP developers give their computers examples of how they want the computer to perform a task. GP software then writes a computer program that performs the task described by the examples.

GP is a robust, dynamic, and quickly growing discipline. It has been applied to diverse problems with great success—equaling or exceeding the best human-created solutions to many difficult problems (Koza et al., 1999), (Deschaine, 2000), (Deschaine et al., 2001), and (Banzhaf et al., 1998).

This paper presents approximately three years of analysis of machine-code-based LGP. To perform the analyses, we used Versions 1 through 3 of an off-the-shelf commercial software package called Discipulus™ (Register Machine Learning Technologies, Inc., 2002). Discipulus™ is a LGP system that operates directly on machine-code.

6.2.1 The Genetic Programming Algorithm

Good, detailed treatments of GP may be found in (Banzhaf et al., 1998) and (Koza et al., 1999). In brief summary, the LGP algorithm in Discipulus™ is surprisingly simple. It starts with a population of randomly generated computer programs. These programs are the "primordial soup" on which computerized evolution operates. Then, GP conducts a "tournament" by selecting four programs from the population—also at random—and measures how well each of the four programs performs the task designated by the GP developer. The two programs that perform the task best "win" the tournament.

The GP algorithm then copies the two winner programs and transforms these copies into two new programs via crossover and mutation transformation operators—in short, the winners have "children." These two new child programs are then inserted into the population of programs, replacing the two loser programs from the tournament. GP repeats these simple steps over and over until it has written a program that performs the selected task.

GP creates its "child" programs by transforming the tournament winning programs. The transformations used are inspired by biology. For example, the GP mutation operator transforms a tournament winner by changing it randomly—the mutation operator might change an addition instruction in a tournament winner to a multiplication instruction. Likewise, the GP crossover operator causes instructions from the two tournament winning programs to be swapped—in essence, an exchange of genetic material between the winners. GP crossover is inspired by the exchange of genetic material that occurs in sexual reproduction in biology.

6.2.2 Linear Genetic Programming using Direct Manipulation of Binary Machine-Code

Machine-code-based LGP is the direct evolution of binary machine-code through GP techniques (Nordin, 1994), (Nordin, 1999), (Nordin and Banzhaf, 1995a), (Nordin and Banzhaf, 1995b) and (Nordin et al., 1998). Thus, an evolved LGP program is a sequence of binary machine instructions. For example, an evolved LGP program might be comprised of a sequence of four, 32-bit machine-code instructions. When executed, those four instructions would cause the central processing unit (CPU) to perform operations on the CPU's hardware registers. Here is an example of a simple, four-instruction LGP program that uses three hardware registers:

```
register 2 = register 1 + register 2
register 3 = register 1 - 64
register 3 = register 2 * register 3
register 3 = register 2 / register 3
```

While LGP programs are apparently very simple, it is actually possible to evolve functions of great complexity using only simple arithmetic functions on a register machine (Nordin and Banzhaf, 1995a), (Nordin et al., 1998).

After completing a machine-code LGP project, the LGP software decompiles the best evolved models from machine-code into Java, ANSI C, or Intel Assembler programs (Register Machine Learning Technologies, Inc., 1998-2009). The resulting decompiled code may be linked to the optimizer and compiled or it may compiled into a DLL or COM object and called from the optimization routines.

The linear machine-code approach to GP has been documented to be between 60 to 200 times faster than comparable interpreting systems (Fukunaga et al., 1998), (Nordin, 1994), and (Nordin et al., 1998). As will be developed in more detail in the next section, this enhanced speed may be used to conduct a more intensive search of the solution space by performing more and longer runs.

6.3 Why Machine-Code-Based, Linear Genetic Programming?

At first glance, it is not at all obvious that machine-code, LGP is a strong candidate for the modeling algorithm of choice for the types of complex, high-dimensional problems at issue here. But over the past three years, a series of tests were performed on both synthetic and industrial data sets—many of them data sets on which other modeling tools had failed. The purpose of these tests was to assess machine-code, LGP's performance as a general-purpose modeling tool.

In brief summary, the machine-code-based LGP software (Register Machine Learning Technologies, Inc., 2002) has become our modeling tool of choice for complex problems like the ones described in this work for several reasons:

- Its speed permits the engineer to conduct many runs in realistic time frames on a desktop computer. This results in consistent, high-precision models with little customization;
- It is well-designed to prevent overfitting and to produce robust solutions; and
- The models produced by the LGP software execute very quickly when called by an optimizer.

We will first discuss the use of multiple LGP runs as a key ingredient of this technique. Then we will discuss our investigation of machine-code, LGP over the past three years.

6.4 Multiple Linear Genetic Programming Runs

GP is a stochastic algorithm. Accordingly, running it over and over with the same inputs usually produces a wide range of results, ranging from very bad to very good. For example, Figure 6.1 shows the distribution of the results from 30 runs of LGP on the incinerator plant modeling problem mentioned in the introduction—the R^2 value is used to measure the quality of the solution. The solutions ranged from a very poor R^2 of 0.05 to an excellent R^2 of 0.95.

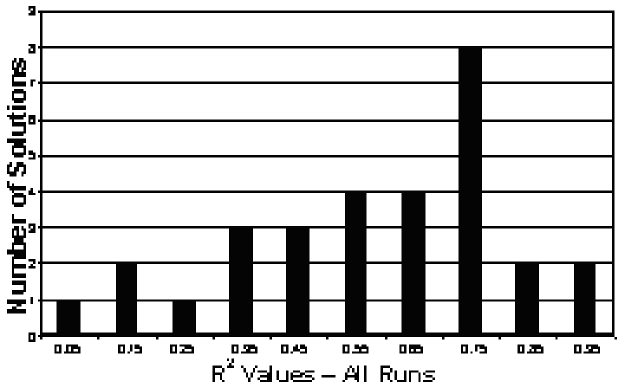


Figure 6.1
Incinerator Control Data. Histogram of Results for 30 LGP Runs. X-Axis is the binned, R2 statistic of the best evolved programs generated over all runs. Higher R2 is a better evolved program. Y-Axis is the count of the best evolved programs in the various R2 bins.

Our investigation to date strongly suggests the typical LGP distribution of results from multiple LGP runs includes a distributional tail of excellent solutions that is not always duplicated by other learning algorithms. For example, for three separate problem domains, an LGP system produced a long tail of outstanding solutions, even though the average LGP solution was not necessarily very good. By way of contrast, and in that same study, the distribution of many neural networks runs on the same problems often produced a good average solution, but did not produce a tail of outstanding solutions like LGP (Francone et al., 1996).

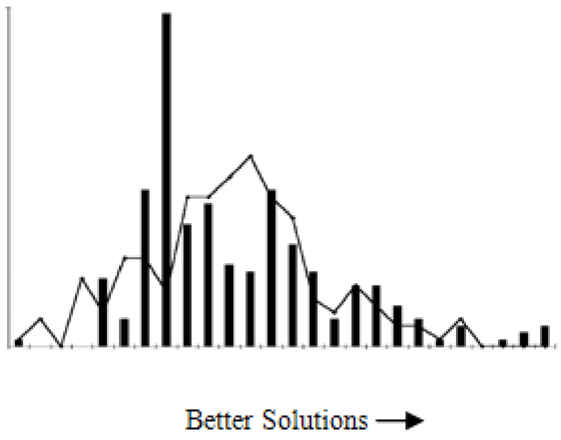


Figure 6.2
Typical Comparative Histograms of the Quality of Solutions Produced by
LGP Runs (bars) and Neural Network Runs (lines). X-Axis shows quality of solution.
Y-Axis shows solution counts.

Figure 6.2 shows a comparative histogram of LGP results versus neural network results derived from 720 runs of each algorithm on the same problem. Better solutions appear to the right of the chart. Note the tail of good LGP solutions (the bars) that is not duplicated by a comparable tail of good neural network solutions. This same pattern may be found in other problem domains (*Id*).

To locate the tail of best solutions on the right of Figure 6.2, it is *essential* to perform many runs, regardless whether the researcher is using neural networks or LGP. This is one of the most important reasons why a machine-code approach to GP is preferable to other approaches. It is so much faster than other approaches, that it is possible to complete many runs in realistic time frames on a desktop computer. That makes it more capable of finding the programs in the good tail of the distribution.

6.5 Configuration Issues in Performing Multiple LGP Runs

Our investigation into exploiting the multiple run capability of machine-code-based LGP had two phases—largely defined by software versioning. Early versions of the Discipulus™ LGP software permitted multiple runs, but only with user-predefined parameter settings.

As a result, our early multiple run efforts (described below as our Phase I investigation) just chose a range of reasonable values for key parameters, estimated an appropriate termination criterion for the runs, and conducted a series of runs at those selected parameter settings. For example, the chart of the LGP results on the incinerator CO2 data sets (Figure 6.1) was the result of doing 30 runs using different settings for the mutation parameter.

By way of contrast, the second phase of our investigation was enabled by four, key new capabilities introduced into later versions of the LGP software. Those capabilities were:

- The ability to perform multiple runs with randomized parameter settings from run to run;
- The ability to conduct hill climbing through LGP parameter space based on the results of previous runs;
- The ability to automatically assemble teams of models during a project that, in general, perform better than individual models; and
- The ability to determine an appropriate termination criterion for runs, for a particular problem domain, by starting a project with short runs and automatically increasing the length of the runs until longer runs stop yielding better results.

Accordingly, the results reported below as part of our Phase II investigation are based on utilizing these additional four capabilities.

6.6 Investigation of Machine-Code-Based, Linear Genetic Programming—Phase I

We tested Versions 1.0 and 2.0 of the Discipulus™ LGP software on a number of problem domains during this first phase of our investigation. This Phase I investigation covered about two years and is reported in the next three sections.

6.6.1 Deriving Physical Laws

Science Applications International Corporation's (SAIC's) interest in LGP was initially based on its potential ability to model physical relationships. So the first test for LGP to see if it could model the well-known (to environmental engineers, at least) Darcy's Law. Darcy's Law describes the flow of water through porous media. The equation is:

$$Q = K \cdot I \cdot A \tag{6.1}$$

where *Q = flow [L3/T]*, *K = hydraulic conductivity [L/T]*, *I = gradient [L/L]*, and *A = area [L2]*.

To test LGP, we generated a realistic input set and then used Darcy's law to produce outputs. We then added 10% random variation to the inputs and outputs, and ran the LGP software on these data. After completing our runs, we examined the best program it produced.

The best solution derived by the LGP software from these data was a four-instruction program that is precisely Darcy's Law, represented in ANSI C as:

```
Q = 0.0
Q += I
Q *= K
Q *= A
```

In this LGP evolved program, Q is an accumulator variable that is also the final output of the evolved program.

This program model of Darcy's Law was derived as follows. First, it was evolved by LGP. The "raw" LGP solution was accurate though somewhat unintelligible. By using intron removal (Nordin et al., 1996) with heuristics and evolutionary strategies the specific form of Darcy's Law was evolved. This process is coded in the LGP software; we used the Interactive Evaluator module, which links to the intron removal," automatic Simplification, and evolved program optimization functions. These functions combine heuristics and ES optimization to derive simpler versions of the programs that LGP evolves (Register Machine Learning Technologies, Inc., 1998-2009).

6.6.2 Incinerator Process Simulation

The second LGP test SAIC performed was the prediction of CO2 concentrations in the secondary combustion chamber of an incinerator plant from process measurements from plant operation. The inputs were various process parameters (e.g., fuel oil flow, liquid waste flow, etc.) and the plant control settings. The ability to make this prediction is important because the CO2 concentration strongly affects regulatory compliance.

This problem was chosen because it had been investigated using neural networks. Great difficulty was encountered in deriving any useful neural network models for this problem during a well-conducted study (Deschaine et al., 2002).

The incinerator to be modeled processed a variety of solid and aqueous waste, using a combination of a rotary kiln, a secondary combustion chamber, and an off-gas scrubber. The process is complex and consists of variable fuel and waste inputs, high temperatures of combustion, and high velocity off-gas emissions.

To set up the data, a zero and one hour off-set for the data was used to construct the training and validation instance sets. This resulted in a total of 44 input variables. We conducted 30 LGP runs for a period of 20 hours each, using 10 different random seeds for each of three mutation rates (0.10, 0.50, 0.95) (Deschaine, 2000). The stopping criterion for all simulations was 20 hours. All 30 runs together took 600 hours to run.

Two of the LGP runs produced excellent results. The best run showed a validation data set R2 fitness of 0.961 and an R2 fitness of 0.979 across the entire data set.

The two important results here were: (1) LGP produced a solution that could not be obtained using neural networks; and (2) Only two of the 30 runs produced good solutions (Figure 6.1), so we would expect to have to conduct all 30 runs to solve the problem again.

6.6.3 Data Memorization Test

The third test SAIC performed was to see whether the LGP algorithm was memorizing data, or actually learning relationships.

SAIC constructed a known, chaotic time series based on the combination of drops of colored water making their way through a cylinder of mineral oil. The time series used was constructed via a physical process experimental technique discussed in (Scientific American, November 1999).

The point of constructing these data was an attempt to deceive the LGP software into predicting an unpredictable relationship, that is, the information content of the preceding values from the drop experiment are not sufficient to predict the next value. Accordingly, if the LGP technique found a relationship on this chaotic series, it would have found a false relationship and its ability to generalize relationships from data would be suspect.

The LGP was configured to train on a data set as follows:

- The inputs were comprised of eight consecutive values from the drop data; and
- The target output was the next-in-sequence value of the drop data.

Various attempts were tried to trick the LGP technique, including varying parameters such as the instructions that were available for evolving the solution.

The results of this memorization test are shown on Figure 6.4. The “step” function shown in Figure 6.4 represents the measured drop data, sorted by value. The noisy data series is the output of the best LGP model of the drop data.

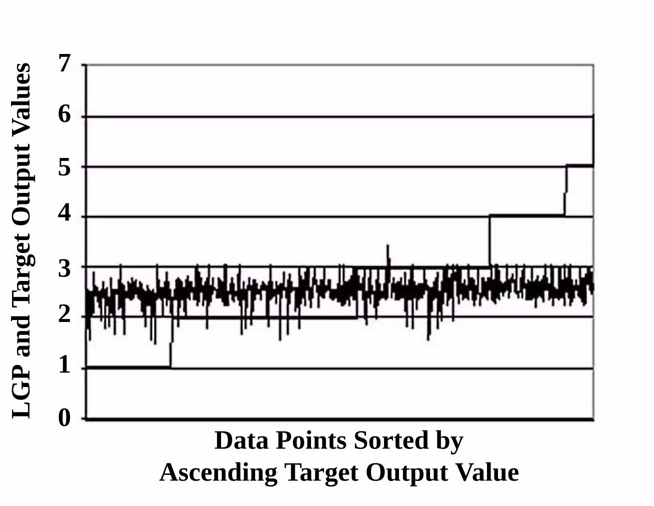


Figure 6.3
Results of Modeling an "Unmodelable," Chaotic Relationship. X-Axis shows individual data points. Y-Axis shows the corresponding target outputs to be modeled and the LGP output for these data points. The stepped series is the target output.
The noisy series is the LGP output.

It is clear that the LGP algorithm was not fooled by this data set. It evolved a program that was approximately a linear representation of the average value of the data set. But it did not memorize or fit the noise.

6.7 Investigation of Machine-Code-Based, Linear Genetic Programming—Phase II

Phase II of our investigation started when we began using Version 3.0 of the LGP software (Register Machine Learning Technologies, Inc., 2002). As noted above, this new version automated many aspects of conducting multiple runs, including automatically randomizing run parameters, hillclimbing to optimize run parameters, automatic determination of the appropriate termination criterion for LGP for a particular problem domain, and automatic creation of team solutions.

6.7.1 Incinerator Problem, Phase II

SAIC used the new software version and re-ran the R&D problem involving CO2 level prediction for the incinerator plant problem (described above). A total of 901,983,797 programs were evaluated to produce the distribution of best 30 program results shown in Figure 6.4.

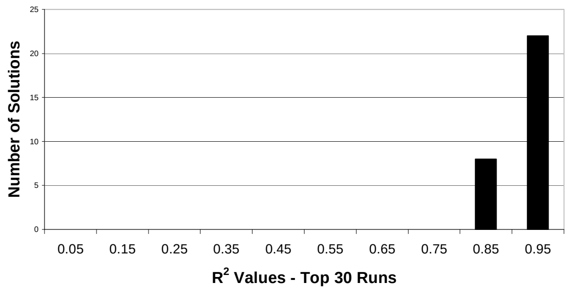


Figure 6.4
Distribution of 30 Best LGP Runs using Randomized Run Parameters for
300 Runs on Incinerator Problem

The enhanced LGP algorithm modeled the Incinerator plant CO_2 levels with better accuracy and much more rapidly than earlier versions. The validation-data-set, seven-team, R^2 fitness was 0.985 as opposed to 0.961 previously achieved by multiple single runs. The CPU time for the new algorithm was 67 hours (using a PIII-800 MHz/100 MHz FSB machine), as opposed to 600 hours (using a PIII 533 MHz /133 FSB machine) that was needed in Phase I. It is important to note that the team solution approach was important in developing a better solution in less time.

6.7.2 UXO Discrimination

The preceding examples are regression problems. The enhanced LGP algorithm was also tested during Phase II on a difficult classification challenge–the determination of the presence of subsurface unexploded ordnance (UXO).

The Department of Defense has been responsible for conducting UXO investigations at many locations around the world. These investigations have resulted in the collection of extraordinary amounts of geophysical data with the goal of identifying buried UXO.

Evaluation of UXO/non-UXO data is time consuming and costly. The standard outcome of these types of evaluations is maps showing the location of geophysical anomalies. In general, what these anomalies may be (i.e., UXO, non-UXO, boulders, etc.) cannot be determined without excavation at the location of the anomaly.

Figure 6.5 shows the performance of ten published industrial-strength, discrimination algorithms on the Jefferson Proving Grounds UXO data—which consisted of 160 targets (Jefferson Proving Grounds, 1999). The horizontal axis shows the performance of each algorithm in correctly identifying points that *did not* contain buried UXO. The vertical axis shows the performance of each algorithm in correctly identifying points that *did* contain buried UXO. The angled line in Figure 6.6 represents what would be expected from random guessing.

Figure 6.5 points out the difficulty of modeling these data. Most algorithms did little better than random guessing; however, the LGP algorithm derived a best-know model for correctly identifying UXO's and for correctly rejecting non-UXO's using various data set configurations (Deschaine et al., 2002) and (Jefferson Proving Grounds, 1999). The triangle in the upper right hand corner of Figure 6.6 shows the range of LGP solutions in these different configurations.

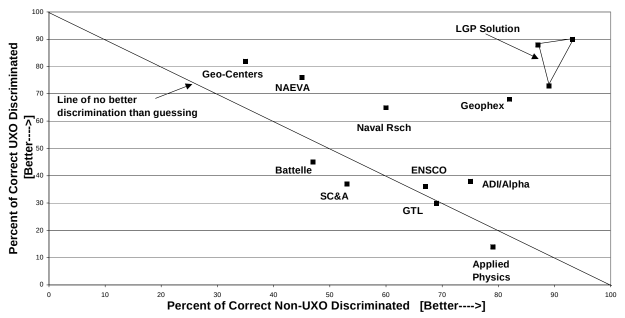


Figure 6.5
LGP and Ten other Algorithms Applied to UXO Discrimination Data
(Jefferson Proving Grounds, 1999)

6.7.3 Eight-Problem Comparative Study

In 2001, we concluded Phase II of our LGP study with a comparative study using machine-code-based LGP, back-propagation neural networks, Vapnick Statistical Regression (Vapnick, 1998), and C5.0 (Quinlan, 1998) on a suite of real-world, modeling problems.

The test suite included six regression problems and two classification problems. LGP and Vapnick Statistical Regression were used on all problems. In addition, on regression problems, neural networks were used and on classification problems, C5.0 was used.

LGP was run at its default parameters and they are documented in (Francone, 2002). In summary, each algorithm was trained on the same data as the others and was also tested on the same held-out data as the others. The figures reported below are the performance on the *held-out, testing data.* Each algorithm was run so as to maximize its performance, except that the LGP system was run at its default parameters in each case.

6.7.3.1 Classification Data Sets Results

Table 6.1 reports the comparative classification error rates of the best LGP, Vapnick Regression, and C5.0 results on the classification suite of problems on the held-out, testing data.

Table 6.1
Comparison of Error Rates of Best LGP, C5.0, and Vapnick Regression Results on Unseen Data for Two Industrial Classification Data Sets.

Problem	Linear Genetic Programming	C5.0 Decision Tree	Vapnick Regression
Company H Spam Filter	3.2%	8.5%	9.1%
Predict Income from Census Data	14%	14.5%	15.4%

6.7.3.2 Regression Data Sets Results

Table 6.2 summarizes the R^2 performance of the three modeling systems across the suite of regression problems on the held-out testing data.

Table 6.2
Comparison of LGP, neural networks and Vapnick Regression on Six Industrial Regression Problems. Value Shown is the R2 Value on Unseen Data Showing Correlation between the Target Function and the Model's Predictions. Higher Values are Better. (N/A means that the algorithm could not be configured to produce an output on that data set.)

Problem	Linear Genetic Programming	Neural Network	Vapnick Regression
Dept. of Energy, Cone Penetrometer,	0.72	0.618	0.68
Kodak, Software Simulator	0.99	0.9509	0.80
Company D, Chemical Batch Process Control	0.72	0.63	0.72
Laser Output Prediction	0.99	0.96	0.41
Tokamak 1	0.99	0.55	N/A
Tokamak 2	0.44	.00	.12

6.7.3.3 Two Examples from the Eight-Problem Study

This section will discuss two examples of results from the eight-problem comparison study—the Laser Output prediction data and the Kodak Simulator data.

Laser Output Problem. This data set comprises about 19,000 data points with 25 inputs. This is sequential data so the last 2,500 data points were held out for testing. The problem is to predict the output level of a ruby laser, using only previously measured outputs.

This is an easy data set to do well upon, measured by R2; but it is very difficult to model the phase with precision. Most modeling tools pick up the strong periodic element but have a difficult time matching the phase and/or frequency components—they generate their solutions by lagging the actual output by several cycles. Figure 6.6 and Figure 6.7 show the output of Vapnick Regression and LGP, respectively, plotted against a portion of the unseen laser testing data.

Figure 6.6 is the result from the Vapnick tool. It picks up the strong periodic element but critically, the predicted output lags behind the actual output by a few cycles. By way of contrast, Figure 6.7 shows the results from LGP modeling. Note the almost perfect phase coherence of the LGP solution and the actual output of the laser both before and after the phase change. The phase-accuracy of the LGP models is what resulted in such a high R^2 for the LGP models, compared to the others.

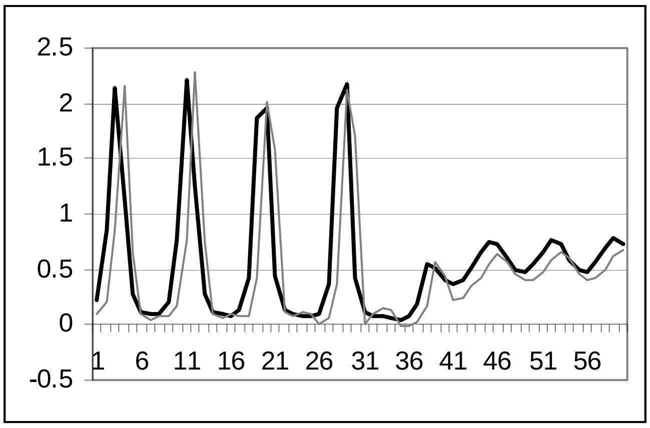


Figure 6.6
Best Vapnick Regression Model on Laser Problem (Light Gray Line) Compared to Target Output (Heavy Line) on Held-Out Data

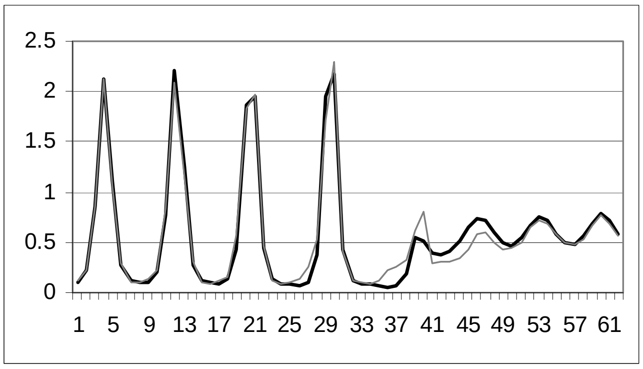


Figure 6.7
Best LGP Model (Light Gray Line) on Laser Problem Compared to Target Output (Dark Line) on Held-Out, Testing Data

Simulating a Simulator. In the Kodak Simulator problem, the task was to use LGP to simulate an existing software simulator. Past runs of the existing simulator provided many matched pairs of inputs (five production related variables) and the output from (Rice and Walton of Eastman Kodak Company). The data set consisted of 7,547 simulations of the output of a chemical batch process, given the five input variables common to making production decisions. Of these data points, 2,521 were held out of training for testing the model.

The results on the testing or held-out data for LGP, Vapnick Regression, and neural networks are reported in Table 6.2. Figure 6.8 and Figure 6.9 graph the LGP and Vapnick models against the target data.

The LGP solution (Figure 6.9) so closely models the Target Output that the predictions completely obscure the target output line. In fact, for all but six of the 2,521 data points, the agreement between the LGP prediction and the actual value is very good. The R^2 fitness on the applied data set for the best team solution was 0.9889. (A second batch of 232 LGP runs achieved a similar R^2 fitness on the applied data set of 0.9814, using a team of seven programs. The range of R^2 for the top 30 programs of this second batch was 0.9707 to 0.9585. This demonstrates analysis repeatability using LGP.)

The Vapnick (Figure 6.8) and neural network solutions were not nearly so close—the R^2 for the Vapnick Model was only 0.80, for example.

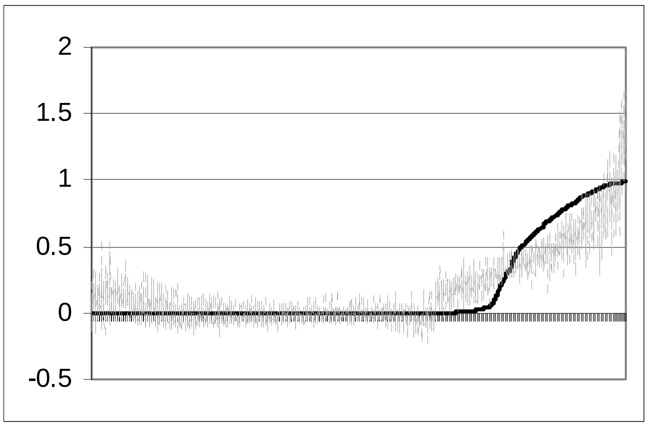


Figure 6.8
Best Vapnick Predictions of Kodak Simulator Data (light-gray series) vs. the Target Data (dark line) on Held-out Data.

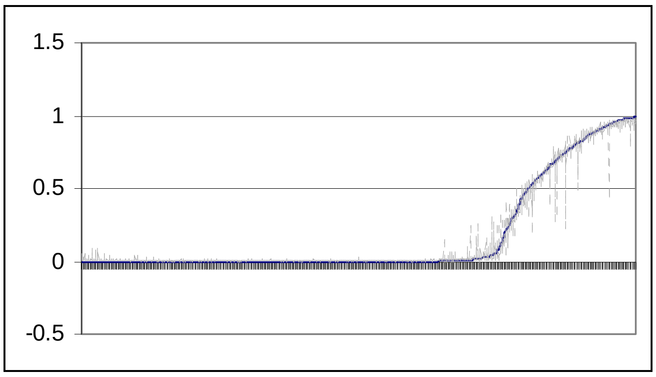


Figure 6.9
Best LGP Model of Company K Simulator Problem (light gray series) vs Target Data (dark series) on the Held-out Data.

6.8 Conclusion Regarding Empirical Studies

The key results of the two phases of our empirical studies of the LGP algorithm are as follows.

First: The LGP software we used consistently produces excellent results on difficult, industrial modeling problems with little customization of the learning algorithm. Note: LGP did not always produce *better* results than all other algorithms studied. However, on every problem studied, LGP produced a model that was as good as, or better than, any other algorithm.

The performance of other learning algorithms was decidedly up-and-down. For example, Vapnick Regression did quite well on the Cone Penetrometer and Company D data but quite poorly on the laser and Company K problems. Neural networks did quite well on the laser and Company K problems but not so well on the Tokamak and incinerator CO2 data sets. C5.0 did well on the census problem but not well on the spam filter problem.

We speculate that one important reason behind the consistently good performance of LGP is that it performs, by default, many runs. Accordingly, it locates the tail of good performing solutions discussed above. Our comfort level that LGP will arrive at a good solution to most problems without customization or 'tweaking" is one of the principal reasons we have settled on LGP as our modeling algorithm of choice for complex and difficult modeling problems.

Second: LGP produces robust models compared to other learning algorithms. Much less attention had to be paid to overfitting problems with the LGP software than with other algorithms. This is not to say that LGP will never overfit data. Give the right data set, it will. But it does so less frequently than the neural network, Vapnick Regression, and C5.0 alternatives we studied.

The LGP system identifies which are the important inputs, and which are not. For example, we screened a wastewater treatment plant with 54 inputs and identified 7 important ones. This reduces the number of inputs to monitor, allows assessment of what will happen if an input goes off-line (for security and contingency planning), and enhances accelerated optimization by reducing the number of decision variables, as discussed below.

Acknowledgments

The results presented in this work are part of a three-plus year collaborative effort between SAIC and Register Machine Learning Technologies to advance the state of the-art of evolutionary computation as applied to complex systems. Specifically thanked for funding and latitude from SAIC are Joseph W. Craver, John Aucella, and Janardan J. Patel. Dr. Gregory Flach, Dr. Frank Syms, and Mr. Robert Walton are gratefully acknowledged for providing the input data sets used in some of the work—as are the researchers who need to keep their identity confidential for business reasons. Christopher R. Wellington–Ad Fontes Academy, Centreville, Virginia conducted the chaotic drop experiment. All computations were performed by, and responsibility for their accuracy lies with, the authors.

Chapter 7

Summary and Conclusions

This thesis details the results of several years research and development that resulted in the current configuration a widely-used Genetic Programming software system, Discipulus™. The ultimate result of this research is that Discipulus™ relies on high values for the probability of mutation and a high proportion of homologous crossover. Our path to reach those conclusions was as follows.

We first investigated introns (or code-bloat) and, in particular, we introduced the notion of explicitly defined introns. The purpose of this research was to use explicitly defined introns to learn why introns emerge in GP and how they may be controlled. Our conclusions in that regard were straightforward:

1. Introns emerge at least in part, as a way for programs to increase their effective fitness, given the destructive effect of the traditional LGP crossover operator on the fitness of their offspring.
2. During the constructional phase of an LGP run, when actual fitness is improving, LGP selects for introns in a linearly increasing fashion;
3. After a GP run has found the best program it is going to find, LGP selects for introns in an exponentially increasing fashion; and
4. The addition of explicitly defined introns to LGP may modestly improve the generalization performance of LGP solutions on unseen data.

All of the above conclusions were consistent with our theory of effective fitness—that is, the ability of an evolving program to produce fit progeny during crossover. The effective fitness theory suggests that programs may improve their effective fitness by: (1) improving actual fitness (their ability to solve the problem at hand); (2) Decreasing the amount of working code in the evolving program; or (3) Increasing the number of introns in the program. We presented evidence consistent with all of these fitness-increasing strategies, manifesting themselves at different times in the run.

We next investigated the probability of the mutation operator and came to the surprising conclusion that LGP performed significantly better at completely implausible mutation rates, from a biological perspective. In this work, using a combination of traditional LGP crossover and aggressive values for probability of mutation, we found that the optimal mutation rate was in the neighborhood of 50%.

Finally, we designed and tested the homologous crossover operator as an alternative to the traditional LGP crossover operator. Our theory was that homologies would emerge in the population, when homologous crossover was introduced, that would make this operator less destructive than the traditional LGP crossover operator and that this would lead to better performance of the LGP run and to a reduction in the number of introns that emerge. In fact, the evidence presented is consistent with both tested hypotheses.

In addition, we discovered that introducing the homologous crossover operator changed the optimal mutation setting from around 50% to around 90-95% and that LGP generalized best to unseen data when using a combination of 90-95% mutation *and* 90-95% homologous crossover. In other words, these two operators interact. Each performs better in the presence of more of the other. In practical terms, we determined that LGP systems should preferentially use high mutation and homologous crossover rates by default.

These results were incorporated into our LGP software. That software was tested over a period of several years in a large-scale comparative study as between LGP and other leading machine-learning tools. The conclusion of that study is that LGP, at its default settings of high-mutation and high homologous crossover, always produced as good or better results than the best alternative machine-learning algorithm.

We conclude with a question and an observation.

The question is, if biological mutation occurs at very low rates (<<1%), why does LGP work so much better at very high mutation rates? After all, both biological and LGP mutation are usually lethal to the offspring.

We speculate that the answer to that question is lies in the fact that, before life can exist and evolve, organisms and their offspring *must* be able to survive reproduction events. Life, we speculate, could not have emerged and thrived if mutation rates for every real-world reproduction event approached 100%. The reason, of course, is that mutation is almost always lethal to the offspring in nature. Thus, reality may limit how much mutation is possible in nature and still permit effective reproduction. Fortunately for us, mutation is rare in the real world.

In LGP, however, there is no such constraint imposed by virtual reality. No matter how many lethal mutations occur, we force the evolving LGP programs and their children to “survive” and have offspring in the computer. In short, we do not permit a high mutation rate to eliminate “life” in LGP populations.

Our final observation returns to Maynard-Smith. He noted that the likely role of crossover in nature is to prevent negative mutations from becoming fixed in a population—it strongly selects against negative mutations. When we switched LGP crossover from all traditional crossover to predominantly homologous crossover, we saw that the LGP system tolerated a much higher mutation rate and produced more optimal results. Similarly, a high mutation rate produced best results with a high homologous crossover rate.

These results suggest that mutation is introducing changes, mostly negative, and that our homologous crossover operator prevents the negative mutations from becoming fixed in the population. In other words, our new crossover operator mimics, to some degree, the effects of biological crossover, which was one of our principal goals in starting this entire line of research.

References

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

Angeline P.J. (1994). Genetic Programming and Emergent Intelligence. In K. Kinnear, Jr. (ed.): *Advances in Genetic Programming*. MIT Press, Cambridge, MA..

Angeline, P.J. (1997). Subtree Crossover: Building Block Engine or Macromutation. In Koza, J., Deb, K., Dorigo, M., Fogel, D., Garzon, M., Iba, H., and Riolo, R., (eds.): *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 9-17. Morgan Kaufmann Publishers, San Francisco, CA.

Back, T. (1993). Optimal Mutation Rates in Genetic Search. In Forrest, S. (ed.): *Proceedings of the Fifth International Conference on Genetic Algorithms, ICGA-93.* Morgan Kaufmann Publishers, San Francisco, CA.

Bäck, T. and Schwefel, H.P. (1993). An Overview of Evolutionary Algorithms for Parameter Optimization. *Evolutionary Computation*, 1(1): pp. 1-23.

Banzhaf, W., Nordin, J.P., Keller, W., Francone, F. (1998). *Genetic Programming, An Introduction*, Morgan Kauffman Publishers, San Francisco, CA.

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

Banzhaf, W. Francone, F., Nordin, J.P (1996a). The Effect of Extensive Use of the Mutation Operator on Generalization in Genetic Programming Using Sparse Data Sets. In, *Parallel Problem Solving from Nature IV, Proceedings of the International Conference on Evolutionary Computatio*n, LNCS, Vol. 1141, pp. 300-309, Springer Verlag.

Banzhaf, W., Francone, F.D., Nordin, J.P (1996b). Explicitly Defined Introns and Destructive Crossover in Genetic Programming. In Kinnear, K. and Angeline, P. (eds.): *Advances in Genetic Programming 2*. MIT Press, Cambridge, MA.

Brameier, M. and Banzhaf, W. (2007). *Linear Genetic Programming*, Springer.

Cramer, N.L. (1985). A Representation for Adaptive Generation of Simple Sequential Programs. In Grefenstette, J. (ed.): *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, p.183-187. Lawrence Erlbaum Associates, Hillsdale, NJ.

Crepeau, R.L. (1995). Genetic Evolution of Machine Language Software, In Rosca, J. (ed.): *Proceedings of the Genetic Programming Workshop at Machine Learning 95*, Tahoe City, CA. University of Rochester Technical Report 95.2, Rochester, NY.

Darwin, C. (1859). *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life.*

Deschaine, L.M. (2000). Tackling Real-World Environmental Challenges with Linear Genetic Programming. *PCAI Magazine,* Volume 15, Number 5, pp. 35-37.

Deschaine, L.M., Patel, J.J., Guthrie, R.G., Grumski, J.T., and Ades, M.J. (2001). Using Linear Genetic Programming to Develop a C/C++ Simulation Model of a Waste Incinerator, *The Society for Modeling and Simulation International: Advanced Simulation Technology Conference*, Seattle, WA, USA April, ISBN: 1-56555-238-5, pages 41-48.

Deschaine, L.M., Hoover, R.A. Skibinski, J. (2002). Using Machine Learning to Complement and Extend the Accuracy of UXO Discrimination Beyond the Best Reported Results at the Jefferson Proving Grounds, In *Proceedings of Society for Modeling and Simulation International, Advanced Technology Simulation Conference.* San Diego, CA.

D'Haesseleer, P. (1994). Context Preserving Crossover in Genetic Programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, Volume 1, pp. 256-261, Orlando, FL, IEEE Press, New York.

Eigen, M. (1992). *Steps Toward Life: A Perspective on Evolution.* Oxford University Press, Oxford.

The ELENA Partners: C. Jutten, Project Coordinator (1995). ESPRIT Basic search Project Number 6891, Document Number R3-B1-P. Available ftp at either ics.uci.edu or at satie.dice.ucl.ac.be.

Fausett, L.V. (2000). A Neural Network Approach to Modeling a Waste Incinerator Facility, *Society for Computer Simulation's Advanced Simulation Technology Conference*, Washington, DC, USA.

Fogel, D.B. (1992). *Evolving Artificial Intelligence*. PhD Thesis, University of California, San Diego, CA.

Fogel, D, Slayton, L. (1994). On the Effectiveness of Crossover in Simulated Evolutionary Optimization. *Biosystems* 32 171 - 182.

Forrest, S. and M. Mitchell (1992). Relative Building Block Fitness and the Building Block Hypothesis. In D. Whitley (ed.): *Foundations of Genetic Algorithms 2*. Morgan Kaufmann Publishers, San Francisco, CA.

Francone, F., Conrads, M., Banzhaf, W., Nordin, P. (1999). Homologous Crossover in Genetic Programming. In Banzaf, W. *et al.*(eds): *Proceedings of the Genetic and Evolutionary Computation Conference*, Vol. 2, pp. 1021-1026, Morgan Kaufmann Publishers, San Francisco, CA.

Francone, F., Deschaine, L. (2004). Extending the Boundaries of Design Optimization by Integrating Fast Optimization Techniques with Machine-Code-Based Linear Genetic Programming. *Information Sciences Journal—Informatics and Computer Science,* Vol. 161, Number 3-4, pp. 99-120. Elsevier Press, Amsterdam, The Netherlands.

Francone, F., Nordin, P., and Banzhaf. W. (1996). Benchmarking the Generalization Capabilities Of a Compiling Genetic Programming System Using Sparse Data Sets, In

Koza *et al.*: *Proceedings of the First Annual Conference on Genetic Programming*, Stanford, CA.

Francone F. (2002). *Comparison of Discipulus™ Genetic Programming Software with Alternative Modeling Tools*. Available at www.rmltech.com.

Francone, F. (1998-2009). *Discipulus™ Owner's Manual*. www.rmltech.com.

Fukunaga, A., Stechert, A., Mutz, D. (1998). A Genome Compiler for High Performance Genetic Programming. In: *Proceedings of the Third Annual Genetic Programming Conference,* pp. 86-94, Morgan Kaufmann Publishers, San Francisco, CA.

Goldberg, D (1989). *Genetic Algorithms in Search Optimization & Learning, Reading:* Addison-Wesley, Upper Saddle River, N.J.

Government Accounting Office (2001). DOD Training Range Clean-up Cost Estimates are Likely Understated. *Report to House of Representatives on Environmental Liabilities,* USA General Accounting Office, April, Report No. GAO 01 479.

Gruau, F. (1993). Automatic Definition of Modular Neural Networks. *Adaptive Behaviour*, 3(2):151–183.

Hansen, N., and Ostermeier, A. (2001). Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation* 9(2): 159-195.

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

Huelsberger L. (1996). Simulated Evolution of Machine Language Iteration, In: *Proceedings of: The First International Conference on Genetic Programming*, Stanford, USA.

Jefferson Proving Grounds (1999). *Jefferson Proving Grounds Phase IV Report*: Graph ES-1, May, Report No: SFIM-AEC-ET-CR-99051.

Koza, J. & Rice, J. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA.

Koza, J., Bennet, F., Andre, D., and Keane, M. (1999). *Genetic Programming III*. Morgan Kaufmann Publishers, San Francisco, CA.

Lang, K. (1995). Climbing Beats Genetic Search on a Boolean Circuit Synthesis Problem of Koza's. In: A. Prieditis and S. Russell (eds.): *Proceedings of the Twelfth International Conference on Machine Learning*. Morgan Kaufmann Publishers, San Francisco, CA.

Levenick, J.R. (1991). Inserting Introns Improves Genetic Algorithm Success Rate: Taking a Cue From Biology. In, Belew, R.K. and Booker, L.B. (eds.): *Proceedings of the Fourth International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, San Francisco, CA.

Masters, T. (1995). *Advanced Algorithms for Neural Networks*. John Wiley & Sons, NY, NY.

Maynard-Smith, J (1994). *Evolutionary Genetics*. Oxford University Press, Oxford, UK.

Nordin, J.P. (1994). A Compiling Genetic Programming System that Directly Manipulates the Machine Code. In K. Kinnear, Jr. (ed.): *Advances in Genetic Programming*. MIT Press, Cambridge, MA.

Nordin, J.P. (1999). *Evolutionary Program Induction of Binary Machine Code and its Applications,* Krehl Verlag.

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

Nordin, J.P., Banzhaf, W. (1995b). Complexity Compression and Evolution. In Eshelman, L., (ed.): *Genetic Algorithms: Proceedings of the Sixth International Conference*, pp. 310-317, Pittsburgh, PA. Morgan Kaufmann Publishers, San Francisco, CA.

Nordin, J.P., Francone, F., and Banzhaf, W. (1996). Explicitly Defined Introns and Destructive Crossover in Genetic Programming. In Angeline, P.J. and Kinnear, Jr., K.E. (eds.): *Advances in Genetic Programming 2*, chapter 6, pages 111-134. MIT Press, Cambridge, MA.

Nordin, J.P., Francone, F., and Banzhaf, W. (1998). Efficient Evolution of Machine Code for CISC Architectures Using Blocks and Homologous Crossover. In *Advances in Genetic Programming 3*. MIT Press, Cambridge MA.

Poli, R. and Langdon, W.B. (1998). On the Search Properties of Different Crossover Operators in Genetic Programming. In *Proceedings of the Third International Conference on Genetic Programming*. Morgan Kaufmann Publishers, San Francisco, CA.

Poli, R. and Langdon, W.B. (1997). A New Schema Theory for Genetic Programming with One-Point Crossover and Point Mutation. In Koza, J., Deb, K., Dorigo, M., Fogel, D., Garzon, M., Iba, H., and Riolo, R. (eds.): *Genetic Programming 1997: Proceedings of the Second Annual Conference,* pages 278-285. Morgan Kaufmann Publishers, San Francisco, CA.

Quinlan, R. (1998). *Data Mining Tools See5 and C5.0.*, Technical Report, RuleQuest Research.

Rechenberg, I. (1994). *Evolutions Strategie '94*. Stuttgart: Holzmann-Froboog (2nd. ed.).

Rechenberg, I. (1993). *Evolutions Strategie '93,* Fromann Verlag, Stuttgart, Germany.

Register Machine Learning Technologies, Inc. (1998-2009). Discipulus™ 1.0-4.0.

Register Machine Learning Technologies, Inc. (2002). *Discipulus™ Users Manual, Version 4.0*. Available at www.rmltech.com.

Rice, Brian, Walton, R. of Eastman Kodak Company. Industrial Production Data Set.

Rosca, J.P. (1995). Entropy-Driven Adaptive Representation. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, Technical Report 95.2, University of Rochester, USA.

Samuel (1963). Some Studies in Machine Learning Using the Game of Checkers. In Feigenbaum, E and Feldman, J. (eds.): *Computers and Thought.* McGraw-Hill, New York.

Schwefel, H.P. (1995). *Evolution and Optimum Seeking. Sixth-Generation Computer Technology Series.* John Wiley & Sons, NY, NY.

Schwefel, H.P. and Rudolph, G. (1995). Contemporary Evolution Strategies, In *Advances in Artificial Life*, pages 893-907. Springer-Verlag, Berlin.

Scientific American, (November 1999). Drop Experiment to Demonstrate a Chaotic Time Series.

The SPARC Architecture Manual (1991). SPARC International Inc., Menlo Park, California.

Soule, T. Foster, J.A. and Dickinson, J. (1996). Code Growth in Genetic Programming. In Koza, J.R., Goldberg, D.E., Fogel, D.B., and Riolo, R.L. (eds.): *Proceedings of the First Annual Genetic Programming Conference*, pages 215-223. MIT Press, Cambridge MA.

Soule, T. and Foster, J.A. (1997). Code Size and Depth Flows in Genetic Programming. In Koza, J., Deb, K., Dorigo, M., Fogel, D., Garzon, M., Iba, H., and Riolo, R. (eds.): *Genetic Programming 1997: Proceedings of the Second Annual Genetic Programming Conference*, pp. 313-320. Morgan Kaufmann Publishers, San Francisco, CA.

Spector, L., Stoffel, K. (1996). Automatic Generation of Adaptive Programs. In *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior: From Animals to Animats* 4, pp. 476-483, MIT Press, Cambridge, MA.

Tackett, W.A. (1995). Greedy Recombination and Genetic Search on the Space of Computer Programs. In Whitley, D. and Vose, M. (eds.): *Foundations of Genetic Algorithms III.* Morgan Kaufmann Press, San Francisco, California.

Teller, A. (1996). Evolving Programmers: The Co-Evolution of Intelligent Recombination Operators. In Angeline, P.J. and Kinnear, Jr., K.E. (eds.): *Advances in Genetic Programming 2*, chapter 3, pages 45-68. MIT Press, Cambridge, MA.

Vapnick V. (1998). *The Nature of Statistical Learning Theory.* Wiley-Interscience Publishing.

Watson, J.D., Hopkins, N.H., Roberts, J.W., Steitz, J.A., and Weiner, A.M. (1987). *Molecular Biology of the Gene*. Benjamin-Cummings, Menlo Park, CA.