

Contents

1	Introduction	5
2	Evolutionary Computation	7
2.1	The Algorithm of Evolutionary Computation	8
2.2	Fitness Assignment	8
2.3	Selection Methods	9
2.4	Genetic Operators	10
2.5	Forms of Evolutionary Computation	11
3	Genetic Programming	15
3.1	Overview	15
3.2	Issues in Genetic Programming	21
3.2.1	Code Growth Control	21
3.2.2	Maintaining Population Diversity	23
3.2.3	Applying Genetic Programming to Difficult Problems	24
4	Methods for Controlling Code Growth in Genetic Programming	27
4.1	A Special Mutation : Simplification	27

4.1.1	Biological Background	28
4.1.2	Simplification as Mutation	29
4.2	Multiobjective Optimization	30
4.2.1	Overview of Multiobjective Optimization	30
4.2.2	Our Selection Scheme Based on the Pareto Nondomination Cri- terion	31
4.3	Experimental Results	33
4.3.1	The Simplifying Mutation Operator	34
4.3.2	Selection Based on the Pareto Nondomination Criterion	36
4.4	Discussion and Future Work	40
5	Fitness Sharing in Genetic Programming	45
5.1	Fitness Sharing	45
5.2	Distance Measures for Genetic Programs	47
5.3	Diversity Measures for Genetic Programs	47
5.4	The Proposed Metric	48
5.5	The Proposed Diversity Measure	51
5.6	Empirical Validation	51
5.7	Discussion	55
6	Genetic Programming and Decision Trees in Synthesis of Four Bar Mecha- nisms	57
6.1	Introduction	57

6.2	The Design Method	59
6.3	Creating the Structural Description	63
6.3.1	Decision Tree Learning	63
6.3.2	Genetic Programming	64
6.3.3	Constructive Induction	66
6.4	Experimentation	69
6.5	Discussion	75
7	Evolving Decision Principles for Multicriteria Decision Problems	79
7.1	Introduction	79
7.2	The Multicriteria Decision Problem	80
7.3	Generating the Decision Principle	84
7.4	Using the Genetic Programming Paradigm	85
7.5	Experimental Results	86
7.6	Discussion	92
8	Conclusions	93

Chapter 1

Introduction

Nowadays we can witness the more and more growing interconnection between the research fields of genetics and computer science. The amazing results on *human genome sequencing* could not have been achieved without today's effective computers, the internet and the advanced data mining techniques. At the same time, both the concept and the working of artificial evolutionary systems imitate natural evolution.

On one side *computers help natural genetics to develop*, and on the other side, *the lessons learned from natural evolution are applied in artificial evolutionary systems*.

The field of *evolutionary computation* was formed in the 1970's when three independent research groups (conducted by John Holland, Lawrence Fogel and Ingo Rechenberg, respectively) began to investigate the possibilities of creating artificial systems that solve certain problems through *artificial evolution* [4, 27].

However, the research in this area made a very rapid progress only in the last decade. *Genetic programming* is the newest form of evolutionary computation that was conceived in the late 1980's as a possible means for *automatic programming*.¹

The possibility of automatic programming has been a central question in computer science from the beginning. Genetic programming provides a partial answer. In a genetic programming system (like in every evolutionary method) one must carefully tune the representation, the evaluation method and the parameters in order to obtain worthwhile result. Still, many intriguing questions are raised even in the case of simple tasks. E.g., how can one assure that the evolution explores the best regions of the program space? How can one define and measure efficiently the diversity of programs? We do not know in advance the size and form of a good program. Then how can we restrict or guide the genetic search in order to keep it off from unnecessary extensive calculations? The benchmark problems seem quite simple. How can one apply the genetic programming paradigm to difficult, real world problems? In the case of complex tasks, one can encounter difficulties that did not cause much trouble in the case of simple tasks. How can the results on test

¹Genetic programming performs an evolutionary search in the space of computer programs and selects the program that solves a given task according to certain criteria.

problems be transmitted to more complex problems? Can genetic programming improve the performance of a system when added as a component?

In this work we try to provide answers to the above questions. The dissertation is structured in three main parts. In the first part we give an overview of evolutionary computation (Chapter 2) and in particular genetic programming (Chapter 3). In Chapter 2 we give short descriptions for the different forms of evolutionary computation. Then we show in Chapter 3 the specific features of genetic programming. We raise key issues for genetic programming: code growth, diversity, real world applications.

In the second part we present our contribution to the theory of genetic programming. In Chapter 4 we demonstrate two methods for limiting the code growth. The first method consists in applying an additional mutation operator that simplifies the structure of a genetic program without altering its behavior. The second method applies multiobjective optimization for the objectives of fitness and program size. Both studied methods are successful in reducing code growth without significant loss of accuracy. In Chapter 5 we define a distance metric for genetic programs and use it for applying the *fitness sharing* technique.² We propose a simple *diversity measure* based on our metric and study the effects of fitness sharing with the help of this diversity measure.

In the third main part we show the application of genetic programming in two complex real world problems. The first problem comes from *mechanical engineering*. Four bar mechanisms play a very important role in practical mechanism design.³ We describe in Chapter 6 our four bar mechanism design system. We demonstrate how genetic programming can be a vital component of a complex design system. We integrate genetic programming with decision trees⁴ into a powerful learning machine.

The second problem belongs to the *decision support* domain of *economics*. The decision-makers have to make many subjective decisions. Consequently, the final decision is sensitive to even small changes in these subjective values. We present in Chapter 7 our genetic programming based system that helps the decision-makers to make stable decisions.

²Fitness sharing is a technique used in genetic algorithms for increasing the diversity in a population of individuals.

³Four bar mechanisms are present in windshield-wipers, door-closing mechanisms, rock crushers, movie cameras, sewing machines, suspension systems of automobiles, etc.

⁴A machine learning method that is employed with great success in extracting useful information from large data sets.

Chapter 2

Evolutionary Computation

The methods of evolutionary computation constitute a special class of search methods [4, 25, 27].¹ Their common feature is that they transpose the notions of natural evolution to the world of computers and imitate natural evolution, as it is shown in Table 2.1. Evolutionary algorithms perform the search by *evolving* solutions to the given problem. They maintain a collection of solutions, and so perform a multidirectional search. In analogy with nature, the fit individuals live to reproduce and the weak individuals, which do not fit to the environment, die off. Usually the offspring are similar to their parents, but mutations can take place. In other words, in a new generation there will appear individuals that resemble the fit individuals from the previous generation. The mutated individuals survive if they are fit to the given environment.

Table 2.1: The terms taken from natural evolution.

Nature	Evolutionary computation
Individual	Solution to a problem
Population	Collection of solutions
Fitness	Quality of a solution
Gene	Part of representation of a solution
Crossover	Binary search operator
Mutation	Unary search operator
Selection	Reuse of good (sub)solutions

Evolution is an *emergent* property of evolutionary computation systems. The computer is only told to (1) maintain a population of solutions, (2) allow the fitter individuals to reproduce, and (3) let the less fit individuals die off. Inherently, the offspring inherit the properties of their parents, and the fitter ones survive for the next generation. The final solutions will be much better than their ancestors from the first generation.

This evolution is directed by fitness evaluation. The search is conducted towards better regions of the search space on the basis of the fitness measure. Each solution in a popula-

¹Search algorithms in general consist of systematically walking through the *search space* of possible solutions until an acceptable solution is found.

tion is evaluated based on how well it solves the given problem and correspondingly, it is assigned a fitness value.

In this chapter we discuss the properties of evolutionary algorithms in general and then briefly describe the four main forms of evolutionary computation.

2.1 The Algorithm of Evolutionary Computation

When solving a problem by evolutionary algorithms, first the proper representation and fitness measure must be designed. Devising the termination criterion² should be the next step. The evolutionary algorithm then works as follows (also shown in Figure 2.1):

1. The initial population is filled with individuals that are generally created at random.³
2. Each individual in the current population is evaluated using the fitness measure.
3. If the termination criterion is met, the best solution is returned.
4. A new population is formed by applying the genetic operators (reproduction, crossover, mutation) to selected individuals from the current population. The parents are selected based on the previously computed fitness values.
5. The actions starting from step 2 are repeated until the termination criterion is satisfied. An iteration is called *generation*.

2.2 Fitness Assignment

The probability of survival of any individual is closely related to its fitness: through evolution the fitter individuals overtake the less fit ones. In order to evolve good solutions, the fitness assigned to a solution must directly reflect its “goodness”, i.e., the fitness function must indicate how well a solution fulfills the requirements of the given problem.

Fitness assignment can be performed in several different ways:

²The termination criterion usually allows at most some predefined number of iterations and verifies whether an acceptable solution has been found.

³Sometimes, the individuals in the initial population are the solutions found by some method resulting from the problem domain. In this case, the scope of the evolutionary algorithm is to obtain more accurate solutions.

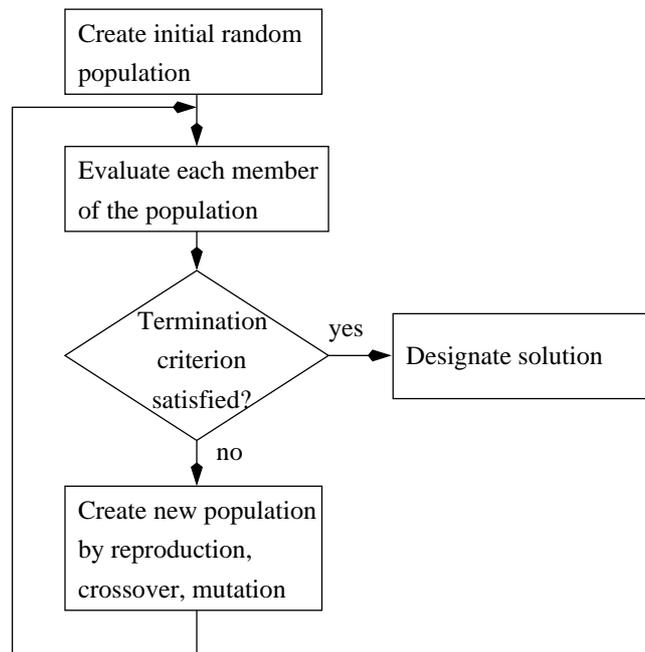


Figure 2.1: The flowchart for evolutionary computation.

- We define a fitness function and incorporate it in the evolutionary system. When evaluating any individual, this fitness function is computed for the individual.
- Fitness evaluation is performed by a dedicated analysis software that is provided to us. In such cases the evaluation could be time-consuming, thus slowing down the whole evolutionary algorithm.
- Sometimes there is no explicit fitness function, but a human evaluator assigns a fitness value to the solutions presented to him.
- Fitness can be assigned by comparing the individuals in the current population. For example, if the problem is about game-playing, and a solution corresponds to a game-playing strategy, its fitness depends on the number of other solutions in the population that are defeated by it.

2.3 Selection Methods

Only selected individuals of a population are allowed to have offspring. The selection is based on fitness: individuals with better fitness are picked more frequently than individuals with worse fitness.

Fitness-proportional selection When using this selection method, a solution has a probability of selection directly proportional to its fitness. The mechanism that allows fitness-proportional selection is similar to a *roulette wheel* that is partitioned into slices. Each individual has a share directly proportional to its fitness. When the roulette wheel is rotated, an individual has a chance of being selected corresponding to its share, as it is shown in Figure 2.2.

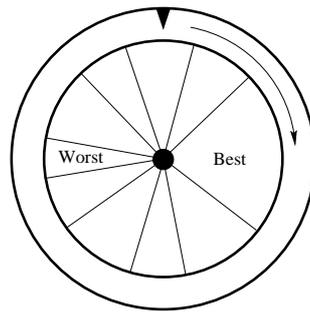


Figure 2.2: The roulette wheel selection mechanism.

Ranked selection The problem of fitness-proportional selection is that it is directly based on fitness. In most cases, we cannot define an accurate measure of “goodness” of a solution, so the assigned fitness value does not express exactly the quality of a solution.⁴ In rank based selection, the individuals are ordered according to their fitness, the worst individual being first and the best individual being last. The individuals are then selected with a probability based on some linear function of their rank. A *selection bias* towards better individuals may be employed.

Tournament selection In tournament selection, a set of n individuals are chosen from the population at random. Then the best of the pool is selected. For $n = 1$, the method is equivalent to random selection. The higher is the value of n , the more directed is the selection towards better individuals.

2.4 Genetic Operators

In each generation, the genetic operators are applied to selected individuals from the current population for creating a new population. Generally, the three main genetic operators

⁴Still, an individual with better fitness value *is* a better individual.

of reproduction, crossover and mutation are employed. By using different probabilities for applying these operators, the speed of convergence can be controlled.

Reproduction A part of the new population can be created by simply copying without change selected individuals from the present population. This is a possibility of survival for the already developed fit solutions.

Crossover New individuals are created as offspring of two parents. One or more crossover points are selected (usually at random) within the representation⁵ of the parents. The marked parts are then interchanged between the parents. The individuals resulting in this way are the offspring. The rationale for using this operator is that the already evolved good parts of solutions (the building blocks, or so-called *schemata*) can be transferred to subsequent generations through crossover. For evolutionary algorithms based on sequential structures, it is easy to prove that the schemata of small size and above average performance are sampled at exponentially increasing rates in consecutive generations [25]. The so-called *schema theorem* states that good schemata are expected to get authority throughout evolution at an exponential rate.

Mutation A new individual is created by making modifications to one selected individual. The modifications can consist in changing one or more values in the representation or in adding/deleting parts of the representation. In different evolutionary computation methods mutation has different roles: in genetic algorithms and genetic programming mutation is considered a source of variability, and is applied in addition to crossover and reproduction, whereas in evolutionary programming and evolution strategies mutation is the only employed genetic operator.

2.5 Forms of Evolutionary Computation

The field of evolutionary computation formed around four main types of evolutionary algorithms. Three types of evolutionary algorithm were independently developed more than 30 years ago: the *genetic algorithm* was created by John Holland and further developed

⁵The different forms of evolutionary computation employ different types of representations for their individuals ranging from strings to trees and graphs.

by David Goldberg [25], the *evolutionary programming* was created by Lawrence Fogel and the *evolution strategies* were created by Ingo Rechenberg. *Genetic programming* is a relatively new development initiated by John Koza [34, 35]. Detailed comparative descriptions of these methods and their fields of application can be found in [2, 4].

Genetic algorithms The genetic algorithms are the most widely used evolutionary algorithms. Genetic algorithms were introduced by Holland for explaining the adaptive processes of natural systems and for creating new artificial systems that work on similar bases.

Genetic algorithms use separate *search space* and *solution space*. The search space is the space of coded solutions, i.e., *genotypes*, while the solution space is the space of actual solutions, i.e., *phenotypes*. Any *genotype* must be transformed into the corresponding *phenotype* before its fitness is evaluated.

Genetic algorithms maintain a population of individuals. The individuals usually have sequential representation, namely, they are represented as strings (binary, real-valued or containing more complex structures as elements). Each individual has a genotype and a corresponding phenotype. The genotypes of new individuals are generated by using all three types of genetic operators: crossover, mutation and reproduction (copying). The parents are selected based on their fitness by using one of the selection mechanisms described in subsection 2.3.⁶ Advanced systems (1) contain additional genetic operators, (2) detect the end of evolution,⁷ or (3) permit the survival of best fit individuals for more generations.

A common characteristic of genetic algorithms is the fast evolution toward better individuals during the early generations. In later generations the population begins to converge and the individuals become more and more similar. Then the population eventually converges to a single solution [25]. For the situation when *premature convergence* occurs,⁸ or the problem is difficult to solve by genetic algorithms, new advanced types of genetic algorithms were introduced. We mention the most important extensions:⁹

- *Steady-state genetic algorithms*, where only one individual is generated in each

⁶Most commonly roulette wheel or tournament selection.

⁷Namely, the individuals of consecutive generations do not differ significantly.

⁸The population converges early to some non-optimal local minimum.

⁹A more complete list can be found in [4].

generation. Convergence is slower, but very fit individuals can be kept during evolution.

- *Multiobjective genetic algorithms*, which allow several objectives to be optimized by the genetic algorithm.
- *Hybrid genetic algorithms*, where genetic algorithms are combined with other (local) search algorithms.
- *Parallel genetic algorithms*, where the genetic algorithms are run on multiple processors.
- *Distributed genetic algorithms*, where several populations are evolved in parallel, with interactions among them.

Evolutionary programming Fogel developed evolutionary programming in the 1960s for generating machine intelligence.¹⁰ Initially, evolutionary programming was applied to the evolution of finite state machines.

Evolutionary programming works directly on the variables of the problem, the search space and the solution space being identical. In contrast to genetic algorithms, in evolutionary programming no crossover is applied. Instead, inheritance from one parent is used, that is, the new individuals are created by mutation. The search method is based on the assumption that mutation can simulate the effects of crossover. In each generation, the population is extended to its double by newly created offspring, and then the best half of the doubled population goes over to the next generation. The parents are selected by tournament selection.

New techniques allow the evolution of control parameters for guiding mutation toward better solutions, thus achieving *self-adaptation*.

Evolution strategies Evolution strategies were created by Bienert, Rechenberg and Schwefel in the 1960s.¹¹ Initially, real physical designs were built, where mutation consisted in changing joint positions or adding and removing segments.

In the computer version, the individuals are represented as real-valued vectors, and, like in evolutionary programming, no distinction between genotype and phenotype is

¹⁰Intelligent behavior was viewed as the ability to predict the environment and to give proper response so as to reach a certain goal.

¹¹Evolutionary strategies and evolutionary programming were developed independently.

made. The simple evolution strategy uses two individuals: a parent and a child. The child is created by mutating the parent. If the child is fitter than the parent, it is selected as parent for the next step. Otherwise the child is discarded and another mutation is applied to the parent.

Newer evolution strategies are the $(\mu + \lambda)$ and the (μ, λ) evolution strategies, where μ is the number of parents and λ is the number of children. These methods maintain populations of solutions, but they are different from genetic algorithms and evolutionary programming in selecting the parent solutions for the next generation deterministically:

- in the $(\mu + \lambda)$ evolution strategy the best μ individuals from the actual parents and children are selected,¹² and
- in the (μ, λ) evolution strategy the best μ individuals from the actual children are chosen, where $\mu < \lambda$.

The mutation consists in randomly changing the components of the parent. Each component is independently mutated by a normal-distributed random variable with zero mean. Mutation is guided by the so-called *strategy parameters*, such as the standard deviation. The strategy parameters evolve in parallel with the solution, and in this way *self-adaptation* is achieved.

Genetic programming Genetic programming was developed later by Koza [34] as an extension of genetic algorithms. Genetic programming operates on computer programs represented in various forms¹³ by using modified genetic operators. Genetic programming is a means for automatic programming, it answers the question raised by Samuel in the late 1950s: *How could computers learn to solve problems without being explicitly programmed?* We only have to specify the possible ingredients of a solution and a proper evaluation method, and genetic programming will evolve a solution that satisfies the requirements encoded in the evaluation method.

In the next chapter we describe genetic programming in detail and we present the related problems that will be treated in the subsequent chapters.

¹²In this notation, the simple evolution strategy uses the (1+1)-selection scheme.

¹³The programs can be represented as trees, sequential structures or graphs.

Chapter 3

Genetic Programming

3.1 Overview

Genetic programming is an extension of genetic algorithms, where the structures undergoing adaptation are not strings but hierarchical computer programs of varying size and shape. This method can successfully find the right program for many different domain problems, such as

- Boolean function identification;
- symbolic function identification;
- obstacle avoidance for robots;
- classification of protein segments; and
- design of electrical circuits.

Representation of programs We consider only tree-based genetic programming where the programs are represented as LISP S-expressions. The motivation underlying this representation is three-fold:

- any – even randomly created – LISP program consisting of combinations of functions and terminals is an executable structure;
- the quality of each LISP program can be easily assessed when executing the program; and
- executable LISP structures can be represented as trees and genetic operations on trees can be easily defined.

Recently, substantial work has been done with linear and graph-based systems, too [3].

Genetic programs are built from *functions* and *terminals*, referred together as nodes. By definition [3], the *terminal set* consists of the inputs of the genetic program, the constants and the eventual functions with no explicit arguments. At the end of each branch

of a genetic tree there is some terminal. The *function set* consists of the statements, operators, and functions available to the genetic programming system. The set of all possible constructs of any programming language is very large, thus it is advisable to restrict the function set to functions specific to the domain of the problem.

Consider for example a simple symbolic regression problem, i.e. the problem of finding a real function of one variable. Then the function set might be $F = \{+, -, *, /\}$, and the terminal set $T = \{x, \text{randomconstants}\}$. A possible genetic program is shown in Figure 3.1.

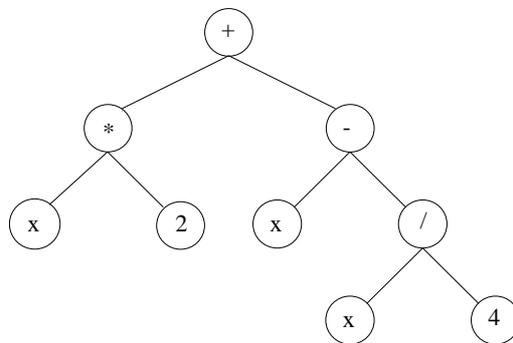


Figure 3.1: A possible genetic program for the symbolic regression problem.

Thus, the search space of genetic programming is the space of all trees that can be formed using the chosen function set and terminal set. There is no difference between genotype and phenotype, the search space and the solution space are the same. For the existence of a solution, the function set and the terminal set must satisfy the properties of

- closure – each function in the function set can accept as arguments any value returned by any function in the function set and any value corresponding to any member of the terminal set; and
- sufficiency – it is possible to represent a solution of the problem as a combination of some functions of the function set and some terminals of the terminal set.

For any problem, the decision must be made for

- a function set consisting of independent elementary functions (neither of these functions could be expressed as a function of the others), or

- a function set extended with functions that are combinations of the elementary functions but are specific to the problem domain (by this extension, one expects to save evolution time and obtain shorter solutions).

For the symbolic regression problem, one could extend the above function set by the exponential function, the logarithmic function, the trigonometric functions, etc. However, if the solution can be easily expressed by using the elementary functions, then this extension will cause unnecessary enlargement of the search space.

Generating the initial population The first step of any genetic programming run is the initialization of the starting population. That is, a number of programs are created for subsequent evolution.

Usually the programs in the initial population are randomly generated. Since we do not know in advance the size and structure of prospective solutions, we define a maximum initial tree depth (or the maximum number of nodes in a tree) and then generate random trees of at most that depth.

For a given maximum depth,

- the *full* method generates full trees of the given depth by selecting nodes randomly from the function set until the maximum depth is reached, and then selecting nodes randomly from the terminal set;
- the *grow* method generates trees of random structure of depth at most equal to the maximum depth by selecting nodes randomly from the function set and terminal set;
- the *Ramped Half-and-Half* method is intended to generate more diverse program structures: an equal number of trees of each depth up to the maximum depth will be created,¹ half of the trees of each depth with the *full* method and the other half with the *grow* method.

Fitness assignment Each program is assigned a fitness value that shows how well the program solves the problem, i.e., how close is the solution given by the program to the ideal solution.

¹If the maximum depth is 5, the same number of trees of depth 2, 3, 4 and 5 will be generated.

Usually the program is evaluated over a set of so-called *fitness cases* consisting of data given as (*input, output*) pairs. The program is run on the input of each fitness case and its result is compared to the output of the fitness case. The fitness value is then computed based on these differences/similarities. The quality of the solution produced by a genetic programming system² can be most reliably evaluated on data that are unknown to the system. For this purpose, the available sample data can be split in two sets: the *training data* and the *test data*. Then, the genetic programming system uses the *training data* for the computation of fitness and *at the end of the run* the final solution is evaluated against the *test data*.

The *raw fitness* is the fitness function that results from the nature of the problem domain. For example, in the case of the symbolic regression problem, the raw fitness could be the sum of errors (or the sum of squared errors) made by the program on the fitness cases.

The *standardized fitness* is a transformed fitness function where zero is assigned to the fittest individual. For the symbolic regression problem, the listed raw fitness functions can be used unchanged as standardized fitness functions.

The *normalized fitness* is a transformed fitness function where the fitness value is always between zero and one.

The genetic operators In tree-based genetic programming crossover consists in exchanging two randomly selected subtrees of two parents, as it is shown in Figure 3.2.

The resulting offspring are valid genetic programs, they satisfy the syntax rules. Besides, crossover is commutative, i.e., once the subtrees are selected, the same offspring are obtained regardless of the order of the parents. As it can be seen, the shape of genetic programs also changes through crossover.

There are two types of mutation commonly used: *point mutation* and *subtree mutation*. Point mutation selects a mutation point at random in the tree, and changes the contents of that node. If it is a function node, another element of the function set will be chosen at random, and if it is a terminal node, another element of the terminal set is chosen at random. This new element then replaces the contents of the selected node (Figure 3.3(a)). Subtree mutation selects a random mutation point in the tree, the subtree below this mutation point is removed and replaced with a randomly generated new subtree

²Or any machine learning system

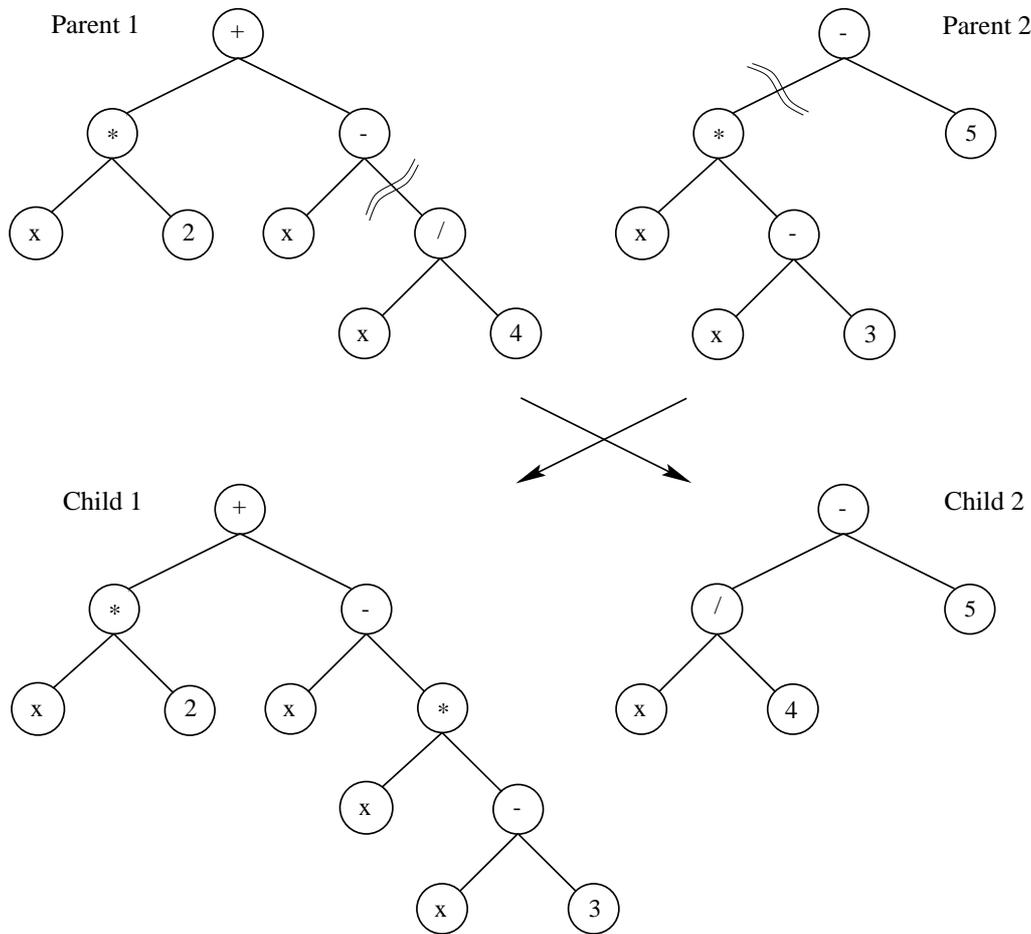


Figure 3.2: The crossover operator applied to genetic programs.

(Figure 3.3(b)).

The preparatory steps Before any genetic programming system could start running, several preparations are needed. The following items must be carefully selected:

- the terminal set – the set of possible inputs to the program to be discovered;
- the function set – the set of primitive functions that may be used by the genetic program, consisting of arithmetic operations, standard programming operations, standard mathematical functions, logical functions or domain specific functions;
- the fitness measure – the method for evaluating how well each program performs, i.e., to what extent it meets the requirements for being a solution;
- the control parameters for a run – size of population, number of generations, proba-

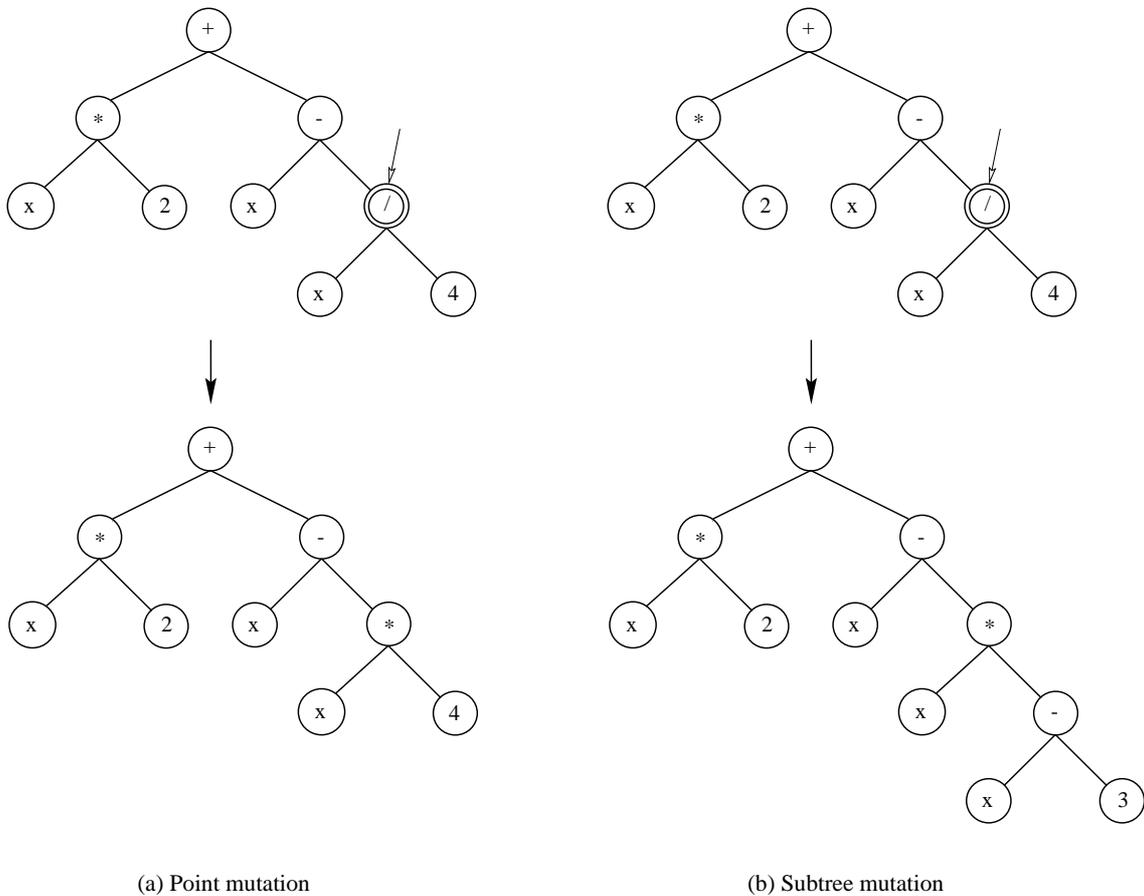


Figure 3.3: The two types of mutation operator for genetic programs.

bility of reproduction, probability of crossover, probability of mutation, maximum program size;

- the method for designating a solution – the definition of a satisfactory solution; and
- the criterion for terminating a run – usually a run ends after a predefined number of generations or when a solution is found.

The algorithm of genetic programming The algorithm of genetic programming systems is the following:

1. Generate an initial population of individuals (computer programs consisting of random compositions of functions and terminals from the function and terminal set).
2. Execute each program of the population on the fitness cases and assign it a fitness value, using the fitness measure.

3. Create a new population of individuals by reproduction, crossover and mutation.
4. Iterate through steps 2-3 until the termination criterion is satisfied.
5. Designate as solution the fittest individual that appeared in any generation.³

3.2 Issues in Genetic Programming

3.2.1 Code Growth Control

An important problem with genetic programming systems is that, in the course of evolution, the size of the programs is continuously growing [1, 34, 59, 60]. The programs contain more and more apparently *non-functional* code. A part of program is non-functional, if for any given input the output of the program is the same whether this part of program is present or not.

Several studies have shown that these seemingly irrelevant or redundant code fragments – called *introns* – shield the highly-fit building blocks of programs from the destructive effects of crossover [1, 47]. Angeline [1] points out that the formation of introns should not be hindered, since they provide a better chance for the transfer of complete subtrees during crossover. Nordin, Francone and Banzhaf [47] introduce so-called *Explicitly Defined Introns* that are inserted in the code and serve as a control mechanism for the crossover probability of the neighboring nodes.

However, better individual programs evolve due to the fact that crossover disrupts the parental code. If crossover disrupted non-functional code, the new individual program would not be better than its parents.

The shielding effect of introns is one possible explanation of code growth. The general causes of code growth are studied in detail in [37, 38, 39]. Genetic programming search finds in each generation the most common programs of the current best fitness. They argue that since in the program space there are much more large programs than short ones, the search of genetic programming proceeds quickly in the direction of large programs.

On the other hand, the fitness evaluation of the genetic programs is the most time-consuming step of any genetic programming run. As the programs grow in size, their evaluation needs more and more CPU time. At the same time, the programs become

³Since we do not keep the best individual in any generation explicitly, the performance of the best individual in subsequent generations may become worse.

unintelligible for humans. Thus, the limitation of code growth without significant detrimental effect on the accuracy of the evolved programs would considerably improve the performance of genetic programming systems.

Koza [34] proposes the use of a maximum permitted size for the evolved genetic programs as a parameter of genetic programming systems. Therefore, genetic programs are allowed to grow until they reach a predefined size. In the same work, a so-called editing operation is proposed for making the output of genetic programming more readable and producing simplified output or improving the overall performance of genetic programming. The editing operation consists of the recursive application of a set of editing rules.

Hooper and Flann [28] apply expression simplification in a symbolic regression problem. They conclude that the accuracy of genetic programs could be improved by simplification. Additionally, simplification could (1) prevent code growth and (2) introduce new constants.

Langdon [38] introduces two special operators, the so-called size fair and homologous crossovers. These operators create an offspring by replacing a subtree of one parent with a carefully selected similar-size subtree of the other parent. By using these operators, code growth is considerably reduced without affecting the accuracy of genetic programs.

There are several studies [31, 60, 70], in which the fitness value is computed on the basis of error and program size. Iba, de Garis and Sato [31] define a fitness function based on a *Minimum Description Length* (MDL) principle. The structure of the tree representing the genetic program is reflected in its fitness value:

$$mdl = Error_Coding_Length + Tree_Coding_Length,$$

i.e., the fitness is computed as the sum of a term based on error and a term based on program size.

Zhang and Mühlenbein [70] demonstrate the connection between accuracy and complexity in genetic programming by means of statistical methods. They use a fitness function based on the MDL principle:

$$Fitness_i(g) = E_i(g) + \alpha(g) C_i(g),$$

where $E_i(g)$ and $C_i(g)$ stand for the error and the complexity of individual i in generation g . The Occam factor $\alpha(g)$ is computed as a function of the least error in the previous

generation $E_{best}(g - 1)$ and the estimated best program size for the current generation $\hat{C}_{best}(g)$. Thus, their fitness function is adaptively changing from generation to generation.

Soule, Foster and Dickinson [60] compare two methods for reducing code growth in a robot guidance problem: (1) editing out irrelevant and redundant parts of code and (2) the use of a fitness function that penalizes longer programs. They conclude that for the robot guidance problem applying this penalty outperforms *any kind* of editing out, so they provide new evidence for [31, 70].

In Chapter 4 we present two different methods for controlling code growth in genetic programming. Our first method [15, 17] uses simplification and thus takes both advice into account:

- Code growth in genetic programming should be limited in order to obtain a comprehensible solution in a reasonable amount of time.
- Introns should be preserved, since they shield the highly-fit building blocks from the harmful effects of crossover.

On the other hand, a straightforward method is optimizing the genetic programs for both fitness and size. The idea is to use multiobjective optimization [11, 29, 56] for these two objectives. We describe our new selection procedure based on the Pareto nondomination criterion that directs evolution towards more accurate and meanwhile – when possible – shorter genetic programs [22]. We do not modify any genetic operator, we just use a selection scheme that forces genetic programming search to prefer the shorter programs of better fitness.

3.2.2 Maintaining Population Diversity

The search space of a search problem may contain more than one extreme.⁴ In artificial genetic search, the population may converge to one of these extremes. In genetic algorithms there are several so-called *niching* techniques that prevent the convergence of the whole population to just one of the extremes [25, 27]. Niching proceeds by penalizing individuals that are “close” to other individuals in the population. It is based on the fact that the search space of genetic algorithms is a metric space (the individuals are represented as real-valued or boolean vectors).

⁴Peak or valley, depending on whether the problem concerns maximization or minimization.

Fitness sharing [25, 27] was introduced as a technique for maintaining population diversity in genetic algorithms. The population is distributed over the different extremes in the search space: in the neighborhood of each extreme, only a number of individuals proportionate to the height/depth of that extreme are allowed.

In Chapter 5 we extend the applicability of fitness sharing to tree-based genetic programming by defining a distance function for genetic programs that reflects the structural difference of trees [20]. For monitoring the diversity of the population throughout evolution, we define a new diversity measure for genetic programs. Our diversity measure is based only on the structural differences of programs within the population, since the diversity of a population does not depend on the performance of its programs on a given data set. By using fitness sharing, we maintain the diversity of the population at a certain level depending on the size of program neighborhood.

3.2.3 Applying Genetic Programming to Difficult Problems

One of the most important challenges in artificial intelligence raised by Arthur Samuel in the late 1950's is to make the computer automatically solve a problem, i.e., without explicitly programming it. Furthermore, the goal is to make the computer behave in such a way that if it were a human being, we would say it is intelligent. In other words, we would be satisfied if automatically created programs were competitive with the products created by human programmers, mathematicians, engineers, and designers.

A system is capable of automatically creating programs if it possesses certain features⁵, such as

- It starts from a high-level specification of problem requirements.
- It produces an executable program.
- It automatically determines the number of necessary steps that the program should take.
- It produces results that are competitive with those produced by human programmers, engineers, mathematicians, and designers (i.e., the results are of similar quality).

⁵A more complete list of 16 desiderata can be found in [32].

Based on the description of genetic programming given earlier in this chapter, one can see that genetic programming meets the first three requirements. Bennett, Koza, Keane and Andre [32] show 14 cases where the results of genetic programming are competitive with the results produced by human scientists or designers. All these cases are instances of difficult problems:

- classification of protein segments,
- discovery of biologically meaningful information in large databases of DNA sequences and protein sequences,
- design of both the topology and the numerical component values for electrical circuits (filters, amplifiers, computational circuits, time-optimal controller of a robot, an electronic thermometer).

Since genetic programming is such a powerful tool for automatic programming, it is especially suitable in cases where

- the problem has no (known) analytical solution,
- the relationships among the variables are poorly understood,
- there is no need for exact solutions, and an approximate solution is acceptable,
- there is a large amount of data that must be processed and/or classified, or
- the size and form of the solution is not known in advance.

We applied genetic programming to mechanism design [14, 16, 18] and multicriteria decision problems [19]. Both problems belong to the previously listed difficult problems that call for automatic programming.

In Chapter 6 we present the application of genetic programming to four bar mechanism synthesis. Kinematic synthesis of four bar mechanisms is a design problem that is difficult to solve by generative methods. We show here a variant based method that combines the genetic programming and decision tree learning methods. The result is the structural description for the class of mechanisms that produce desired coupler curves. For the purpose of finding and characterizing feasible regions of the design space, we integrate genetic programming into a constructive induction system.

In Chapter 7 we show the application of genetic programming to multicriteria decision problems. In multicriteria decision problems many subjective decisions must be made,

such as judging the importance of the different criteria and assigning the values of the alternatives with respect to subjective criteria. Since subjective decisions are approximate, it is very important to analyze the sensitivity of results when small modifications of the subjective values are made. When solving a multicriteria decision problem, it is desirable to choose a decision principle that conducts to a solution, which is as stable as possible. We show here a method based on genetic programming that produces better decision principles than the commonly used ones. The theoretical expectations are validated by two case studies.

Chapter 4

Methods for Controlling Code Growth in Genetic Programming

As outlined in Subsection 3.2.1, code growth is a general problem of genetic programming systems, whose causes are not completely understood. Reducing this unwanted code growth with minor side-effects is a major objective of genetic programming research. We present here two methods for reducing the size of genetic programs without significantly affecting their accuracy: a method based on simplification of programs and a method based on multiobjective optimization.

We faced the problem of code growth during the development of the mechanism design application that is presented in Chapter 6. There, genetic programming is used for evolving arithmetic expressions, that is, for solving a symbolic regression problem. At the same time, symbolic regression is an important benchmark problem for testing genetic programming systems. Therefore, it was straightforward to test our code growth limiting methods on symbolic regression problems. The methods can be applied to other genetic programming systems with the following remarks:

- When applying the simplification method, the corresponding rules that simplify the structure of programs have to be defined.
- For both our methods, the appropriate parameter setting should be found by conducting a few preliminary experiments.

The results shown in this chapter are based on our papers [15, 17, 22].

4.1 A Special Mutation : Simplification

Our first method consists in applying an additional mutation operator that simplifies the structure of a genetic program without altering its behavior.

4.1.1 Biological Background

The DNA of bacteria contains continuous coding sequences. For many years it was believed that the genes of higher organisms are also continuous. This view was changed in 1977, when it was discovered that the genes of some eukaryotic organisms (made of nucleated cells) are discontinuous, and the non-coding sequences are much longer than the coding sequences.

Generally, the genes of living organisms consist of:

- exons - base sequences that encode proteins or polypeptides; and
- introns - base sequences that do not participate directly in the production of proteins.

The introns can influence the amount and the quality of the proteins expressed by the gene in which they occur. But the mechanism of the indirect effect of introns on protein production is not known. They represent regions in which DNA can break and recombine without affecting the encoded proteins [61].

There is evidence suggesting that “*introns were present in ancestral genes and were lost in the evolution of organisms that have become optimized for very rapid growth, such as eubacteria and yeast*” [61]. Gilbert [23] points out that “*introns have been used to assemble those genes that are the late product of evolution*”. At the same time, he brings evidence for the *loss* of introns during evolution. Thus, introns play an important role in evolution, when shielding the exons from destruction through crossover. But they can disappear in the course of evolution.

On the other hand, the main source of variability is mutation. From the many existing types of mutation, we consider the following [3, 66, 67]:

- point mutation - change of one base pair to another;
- neutral mutation - a genetic change that is neither advantageous nor disadvantageous for the organism;
- frameshift mutation - insertion or deletion of one or more base pairs; and
- large DNA sequence rearrangement.

Usually, in genetic programming systems the analog of the first type is implemented. In addition, we also consider here the other mutation types.

4.1.2 Simplification as Mutation

We designed a special mutation operator that modifies only the structure of a genetic program; the interpretation and the fitness value remain the same. This modification is intended to eliminate the occasional introns and simplify the structure of the genetic program, without altering its accuracy (in a similar way to the editing operation of Koza [34] and the expression simplifier of Hooper and Flann [28]). For the symbolic regression problem, this mutation operator performs the algebraic simplification of the expression of a genetic program. The simplifier is implemented in Prolog and consists of approximately 250 clauses. The simplifier works in three stages:

1. The simplifications regarding multiplication and division are performed.
2. The unnecessary parentheses are eliminated.
3. The simplifications concerning addition and subtraction are performed.

These steps are repeated until no more simplifications are possible. For each stage, we show several simplification rules in Table 4.1, where k_1 , k_2 and k are constants and X , Y and Z are variables.

Table 4.1: Some simplification rules.

Stage	Original expression	Simplified expression	Binding
1	$k_1 k_2 X$ $k (X + Y)$	$k X$ $k X + k Y$	$k = k_1 k_2$
2	$X - (-Y)$ $X - (Y + Z)$	$X + Y$ $X - Y - Z$	
3	$0 + X$ $k_1 X + k_2 X$	X $k X$	$k = k_1 + k_2$

We apply the simplification (simplifying mutation) operator in addition to the usual recombination operators (crossover and point mutation). Applying the simplification in every generation might be too drastic and time-consuming and, therefore, we make the frequency of its application a parameter of the genetic programming system (also suggested by Koza [34]). The experimental results are shown in Subsection 4.3.1.

4.2 Multiobjective Optimization

Our second method applies multiobjective optimization for the objectives of fitness and program size.

4.2.1 Overview of Multiobjective Optimization

Multiobjective optimization is the problem of optimizing more objective functions (a vector function) over the same variable(s). The objective functions are usually in conflict with each other, i.e., they do not yield the optimum for the same value of the variable(s).

Formally, the problem can be stated [11] as finding $x \in D$ that optimizes

$$\bar{f}(x) = [f_1(x), f_2(x), \dots, f_k(x)],$$

where $\bar{f} : D \rightarrow R^k$.

Multiobjective optimization methods range from linear combination of the multiple objectives into one scalar objective to techniques based on the Pareto nondomination criterion [11, 64]. Supposing that we want to minimize the objective functions,

an element $u \in D$ dominates element $v \in D$ if

$$\forall i \in \{1, 2, \dots, k\}, f_i(u) \leq f_i(v) \text{ and } \exists i \in \{1, 2, \dots, k\}, f_i(u) < f_i(v).$$

In other words, a potential solution u dominates another potential solution v if it is better over at least one criterion and not worse over any other criterion. u belongs to the Pareto optimum set if there is no $v \in D$ that dominates u . The Pareto optimum consists of the set of nondominated solutions (each solution is better than the others over some criteria but worse over the other criteria).

Since our goal is to find the fittest genetic program that is also limited in size, we can reformulate our problem as a multiobjective optimization problem with two objectives: fitness and program size.

Multiobjective Optimization in Evolutionary Computation There are two approaches in evolutionary multiobjective optimization surveyed by Coello [11] that we would like to discuss here.¹

¹For a comparison of various approaches see [72].

Schaffer [57] uses an extension of the Simple Genetic Algorithm (SGA), called Vector Evaluated Genetic Algorithm (VEGA) that differs from the SGA only in the selection method: for a problem with k objectives, in each generation k subpopulations of size N/k are generated (population size N). For each objective, one subpopulation is generated by proportional selection. Then, the genetic operators are applied in the usual way. The disadvantage of the method is that by preferring individuals with excellent performance over one dimension, individuals with moderately good values for all dimensions will not survive. That is why we did not try this method for our problem of finding fit and also small genetic programs. The result would have been the formation of two species: one with very fit (but generally complex) genetic programs and one with very small (but unfit) genetic programs.

Horn, Nafpliotis and Goldberg [29] propose a tournament selection scheme based on the Pareto domination criterion. Tournament selection is used as follows: two individuals are picked randomly from the population, a comparison set is also picked randomly from the population and the two individuals are compared against the individuals from the comparison set. If one individual is Pareto nondominated and the other is Pareto dominated by the comparison set, then the first individual is selected. If both are dominated or nondominated, the winner is the competitor with less neighbors from the comparison set. By modifying the size of the comparison set, one can control the speed of convergence (selection pressure).

The other methods based on the Pareto nondomination criterion use different kinds of ranking [64]. Generally, the preferred individuals are assigned the same rank while the other individuals are assigned some less desirable rank. Then selection is performed based on these ranks.

4.2.2 Our Selection Scheme Based on the Pareto Nondomination Criterion

Since our goal is to minimize standardized fitness² [34] and program size, we can formulate the problem as a multiple objective minimization problem over the program space P :

$$\text{find } gp \in P \text{ that minimizes } \bar{f} : P \rightarrow R_+^2, \text{ with } f_1(gp) = \text{fitness}(gp) \text{ and}$$

²A lower value of standardized fitness is a better value, indicating a fitter individual.

$$f_2(gp) = size(gp).$$

Our selection scheme is based on tournament selection. Generally, tournament selection consists of randomly picking a number of individuals from the population and selecting the one with best fitness. Our method instead selects an individual that is not dominated by any of these random individuals. The selection algorithm is the following:

1. a number of individuals (the comparison set) is randomly picked from the population;
2. another individual is also randomly picked from the population;
3. if the new individual is not dominated by any individual from the comparison set, then it is selected;
4. if the new individual is dominated by some individual from the comparison set, the procedure is repeated from step 2.

The initial random population may contain very small but unfit individuals. Then the smallest individual is not dominated by any individual, independently from its fitness, and thus has the same chance of being selected, as the fittest individual (which is also not dominated). Such small (and erroneous) individuals can slow down or even hinder convergence to any solution.

In order to prevent evolution from leading to very short programs (in fact too short for being solutions), we allow the selection of a fitter program consisting of more nodes over a less fit program consisting of few nodes. That is, we allow any individual to dominate another one even if the first is somewhat larger than the second, provided that it has less errors:

gp_1 dominates gp_2 if

$$fitness(gp_1) < fitness(gp_2) \text{ and } size(gp_1) < size(gp_2) + bias(gp_2).^3$$

The most straightforward choice of $bias(gp_2)$ is the **constant function** $bias(gp_2) = k$. But allowing the same difference for larger programs could cause unnecessary code growth. Thus, a better choice is an **adaptively changing bias**: a decreasing function of program

³Equality is omitted because if two individuals have the same fitness, the larger one cannot dominate the shorter one.

size, such as $bias(gp_2) = k/size(gp_2)$. If a program is small, then it might be dominated by a much bigger program that has better fitness, but if a program is already big, it might be dominated by a smaller or similar size program of better fitness.

By introducing the bias, the domination relation loses the transitivity property, i.e., there exist cases when gp_1 dominates gp_2 , gp_2 dominates gp_3 , but gp_1 does not dominate gp_3 . We note that in our selection scheme, we do not need transitivity: we compare one individual with each element of a set and if only one element of the set dominates this individual, it will not be selected. The relations among the elements of the comparison set have no effect on selection.

There is a third possibility of extension that keeps transitivity: instead of *size* we minimize the **thresholded function**

$$f_2(gp) = \begin{cases} k & \text{if } size(gp) < k \\ size(gp) & \text{otherwise.} \end{cases}$$

Practically, in the case of small programs domination is determined by better fitness only (as in normal tournament selection), and in the case of large programs the initially proposed domination criterion is used.

In all three cases the value of constant k depends on the difficulty of the studied problem: the more difficult the problem, the greater the value of the constant. Then constant k reflects our definition of big program, i.e., program size by which we expect that solutions exist. The experimental results are shown in Subsection 4.3.2.

4.3 Experimental Results

We conducted experiments on several symbolic regression problems. The test problems were randomly selected from the set of polynomials of degree up to 7 having real roots in the $[-1, 1]$ interval. The goal was to evolve programs that approximate the selected functions in the $[-1, 1]$ interval. For each problem, the training was performed on 50 randomly selected data points in the $[-1, 1]$ interval. The resulted best programs were then tested on 1000 randomly selected data points that were different from the training data. The mean error on the test data served as the basis for comparing the accuracy of different methods.

The parameter setting is shown in Table 4.2. Since the results for polynomials of

the same degree are very close, we show here representative examples of polynomials of degree 2, 5, and 7, respectively.

Table 4.2: The genetic programming parameter setting.

Objective	Evolve a function that approximates the selected polynomial function in the $[-1, 1]$ interval
Terminal set	x , real numbers $\in [-1, 1]$
Function set	$+$, $*$, $/$
Fitness cases (Training data)	$N = 50$ randomly selected data points (x_i, y_i) , #1 $y_i = x_i(x_i + 0.3)$ #2 $y_i = (x_i + 0.2)^2(x_i - 0.5)(x_i + 0.5)(x_i - 0.7)$ #3 $y_i = (x_i + 0.9)(x_i - 0.9)(x_i - 0.6)^2(x_i + 0.8)(x_i + 0.3)(x_i + 0.4)$
Test data	$K = 20$ sets of $N = 50$ randomly selected data points, each different from the training data
Raw fitness and also standardized fitness	$\sqrt{\frac{1}{N} \sum_{i=1}^N (gp(x_i) - y_i)^2}$, $gp(x_i)$ being the output of the genetic program for input x_i
Population size ⁴	100
Crossover probability	90%
Probability of Point Mutation	10%
Selection method	Tournament selection, size 10
Termination criterion	The error of the best genetic program is less than 0.01
Maximum number of generations	50 for simplification, 100 for Pareto methods
Maximum depth of tree after crossover	unlimited
Initialization method	Ramped Half and Half

4.3.1 The Simplifying Mutation Operator

In this case, only the error is included in the fitness measure. There is no term related to program size, *a shorter program with more errors will not be preferred to a longer program containing less errors*. The mechanism for limiting code growth is distinct from

⁴We made several runs with population size 200, 500 and 1000 and the differences between the results of runs using the different selection methods (and the same population size) were similar to those on population size 100. Of course, convergence to some solution occurs in earlier generations, but the required effort for getting one solution is similar.

the selection mechanism. We introduced simplification as a mutation operator in order to reduce code size without affecting the fitness-based selection mechanism.

We added two parameters: the frequency of simplification (F) and the probability of simplification (P). Simplification is applied every F -th generation, for each individual program with probability P . We made experiments with P ranging from 10 % to 100 % and $F \in \{1, 2, 5\}$. For each parameter setting we performed 50 runs and recorded their average. We discuss the results for the regression of the polynomial of degree 5 (problem #2), the other cases are similar.

In order to establish a good ratio between the two parameters P (probability of simplification) and F (frequency of simplification), we compared the results of the runs with the same overall simplification ratio P/F ($P/F = 10\%$ is achieved when (1) 10% probability of simplification in each generation, (2) 20% probability of simplification in every second generation, or (3) 50% probability of simplification in every fifth generation is applied). Considering program size, the best results were obtained when simplification was applied in every generation. In the case when each individual of every fifth generation was simplified, the program size had an alternating behavior: after every fifth generation the average program size was reduced by simplification, then programs were allowed to grow for the next five generations. In the meantime, the accuracy of the best program was better when simplification was applied less frequently (and with higher probability, keeping P/F constant). In particular, the results for $P = 20\%$ and $F = 2$ show a good balance of accuracy and program size.

Table 4.3: The results of simplification when using $P = 20\%$. Each line shows the mean values of 50 independent runs over 50 generations of one method.

Method	F	Best Program Fitness		Best Progr. Size	Final Progr. Size	Code Growth [nodes/gen.]	CPU Time [s] ⁵
		Train	Test				
Orig. GP		0.038	0.033	114	211	4.1	41
Simpl.	5	0.059	0.043	123	145	2.8	35
Simpl.	2	0.056	0.039	98	128	2.4	36
Simpl.	1	0.067	0.046	55	96	1.8	49

⁵The average execution time of the runs on a Sun Ultra Enterprise 10 Server (440 MHz UltraSPARC-III processor).

We made another comparison among the results of runs with different frequencies of simplification, keeping the probability constant at 20%. Table 4.3 shows the results for the cases: (1) no simplification (original genetic programming), (2) simplification in every fifth generation, (3) every second generation and (4) every generation. As we expected, in the case of genetic programming without simplification the program size is growing at a much higher rate than in the case of using simplification. But the solution accuracy is worse in the latter case. According to the Welch test⁶, the difference in fitness between original genetic programming and simplification in every second generation is not significant (at significance level 2 %). When considering the needed CPU time, one can see that simplification in every generation is more time-consuming than no simplification. Thus, *for better accuracy and less CPU time, it is advisable to apply simplification less often.* For the given problem, the best setting is $P = 20\%$, $F = 2$.

We also compared the results of runs when the frequency of simplification was constant and the probability of simplification varied between 0-100%. While the size of programs was growing fast when no simplification was applied, it was quite stable when the probability of simplification was high. In the case of simplifying each individual program, the accuracy of the best program was worse than for the other cases. Thus, we found again that applying simplification more often leads to much smaller, but slightly worse solutions.

When choosing the probability of simplification, *one has to make a trade-off between accuracy and program size:*

- more accurate programs that grow moderately (less simplifications); or
- less accurate programs that do not grow (more simplifications).

4.3.2 Selection Based on the Pareto Nondomination Criterion

We experimented several parameter settings of the three Pareto selection methods described in Subsection 4.2.2: (A) constant bias; (B) adaptive bias; and (C) thresholded function. The results were compared with those of original genetic programming.

In the first group of experiments, the maximum allowed depth for the random programs in the initial population was 4. There was no depth limit set for the evolving pro-

⁶The Welch test is used to determine whether the means of two random variables (with different estimated standard deviation) are significantly different.

grams during the runs. It was typical that when more code growth was allowed (greater k), the accuracy of the obtained programs was closer to the accuracy of programs obtained by original genetic programming (sometimes the program accuracy of our method was even better). On the other hand, when setting the value of the constant, *a trade-off must be made between code length and program accuracy*. Since we did not know in advance the appropriate constant value for each problem, we experimented with several settings: for constant bias, $k \in \{20, 30, 50\}$, for adaptive bias, $k \in \{50, 100, 500\}$, for the thresholded function, $k \in \{25, 50, 100\}$. For greater values of k the method is close to original GP both in terms of accuracy and program size. These values were chosen within the range of shorter sizes of solutions obtained in a few preliminary GP runs.

Table 4.4: The results of the selection methods based on the Pareto nondomination criterion on the different symbolic regression problems. Each line shows the mean values of 50 independent runs of one method.

#	Method	k	Best Program Fitness		Best Progr. Size	Final Progr. Size ⁷	Nr. Sol.	Effort [1000 indiv.] ⁸	CPU Time [s]
			Train	Test					
1	Orig. GP		0.043	0.053	28	54	25	6.6	11
	Pareto A	20	0.060	0.064	10	8	18	7.4	3
	Pareto B	50	0.070	0.073	9	13	19	5.2	4
	Pareto C	100	0.059	0.062	12	12	16	5.5	4
2	Orig. GP		0.035	0.031	270	427	12	126	117
	Pareto A	50	0.035	0.029	231	382	4	495	82
	Pareto B	500	0.039	0.033	234	229	5	254	54
	Pareto C	100	0.042	0.032	228	273	3	213	69
3	Orig. GP		0.017	0.023	179	639	1	1112	172
	Pareto A	30	0.018	0.021	131	415	1	1203	106
	Pareto B	500	0.017	0.023	189	516	3	614	131
	Pareto C	100	0.017	0.023	188	557	1	1566	150

In Table 4.4 we present the results on 50 independent runs of each method on the 3 selected problems. For each method, the results for the setting that produced best fitness are shown. In order to obtain the fittest programs of shortest size, a fine-tuning of the constant values would be needed. However, the differences in program size at the end

⁷The average size of the programs at the end of the runs.

⁸We measure the computational effort as the number of individuals that must be processed in order to yield a solution with probability 99% [35, pp. 103].

of the runs are remarkable, even if the parameter setting is not the best possible. For the first two problems, convergence is slower in the case of the Pareto selection method (as indicated by the smaller number of solutions found), but the run over the same number of generations consumes much less CPU time. The run over more generations leads to not much larger, but more accurate programs. In the case of the third problem, program accuracy is less affected by the Pareto selection scheme, and the best programs are generally larger than in the case of original genetic programming. But the average programs are much shorter, and less time is needed for their evaluation. When using smaller values of k for the same problem, the accuracy is slightly deteriorated, but programs are much shorter.

In order to see whether the differences in program accuracy are significant, we performed the Welch test for the fitness on test data. The best programs created by original GP were compared to the best programs created by each Pareto method. We found that the differences are statistically not significant (even in the case of problem #1, Pareto method B, the difference is negligible, at a significance level of 5%).

The fact that the computational effort is generally greater for the Pareto method means that there is a greater number of individuals that must be evaluated before getting a solution. But these individuals contain far less nodes than their correspondents in original GP, thus the CPU time is a better indicator for comparing the performance of the different methods.

Table 4.5: The results of the selection methods based on the Pareto nondomination criterion when the maximum program depth in the initial random population is 12. Each line shows the mean values of 50 independent runs of one method.

Method	k	Best Program Fitness		Best Progr. Size	Final Progr. Size	Nr. Sol.	Effort [1000 indiv.]	CPU Time [s]
		Train	Test ⁹					
Orig. GP		0.032	0.029	370	572	5	168	201
Pareto A	50	0.046	0.043	206	262	10	145	104
Pareto B	500	0.035	0.033	255	225	7	276	96
Pareto C	100	0.038	0.037	200	228	3	622	101
Pareto D		0.075	0.061	235	368	7	276	139

⁹With the exception of method Pareto D, all the differences in fitness are negligible, at a significance level of 2%.

Since our method limits code growth, we conducted the second group of experiments on initial populations of individuals of depth up to 12, generated by the same “Ramped Half and Half” method. The results on problem #2 are shown in Table 4.5. By using the same setting for k we could obtain even more solutions than original genetic programming, with less computational effort (method A). In all cases of Pareto selection, the CPU time of one run was much less than the time needed by original GP. In the meantime, according to the Welch test, the differences in program size between original GP and Pareto selection methods are significant (at significance level 1%).

There is only one problem: the determination of the proper value for constant k . We propose here a quite simple strategy. Supposing that the initially allowed program size is sufficiently large for solutions to exist, we could modify our criteria by taking the average program size instead of constant k . The last row of Table 4.5 summarizes the results of method D that uses the average program size instead of k in the expression of the thresholded function. The results are interesting: more successful runs than in the case of constant (7 versus 3) and much more unsuccessful runs¹⁰ (19 versus 6) what is reflected in the slower average convergence. By examining the evolution of program size in the case of successful runs one can define the initial value of the constant. The best setting is determined by fine-tuning.

Usually, when the Pareto nondomination criterion is used, a set of nondominated solutions are obtained. We present in Figure 4.1 the evolution of the nondominated individuals during one run of method B on problem #2. One point (f, s) in the $(fitness, size)$ plane is the projection of all programs from the program space that have fitness f and size s . Each line corresponds to the nondominated individuals of the indicated generation,¹¹ the solid line representing the Pareto solution of the run. It can be seen, how the set of nondominated individuals approaches the optimum (for the objectives of fitness and size).

Next we compared the results of the two groups of experiments. In Figure 4.2 we show the evolution of the fitness of the best individual and the evolution of average program size, respectively (on problem #2). We present the results of original GP versus only one Pareto method for the sake of intelligibility. When using the Pareto selection method, the accuracy of programs converges slower, but when allowing larger programs in the ini-

¹⁰Runs that converge to some local optimum (premature convergence).

¹¹We connected the points corresponding to one generation in order to better differentiate among the individuals from different generations.

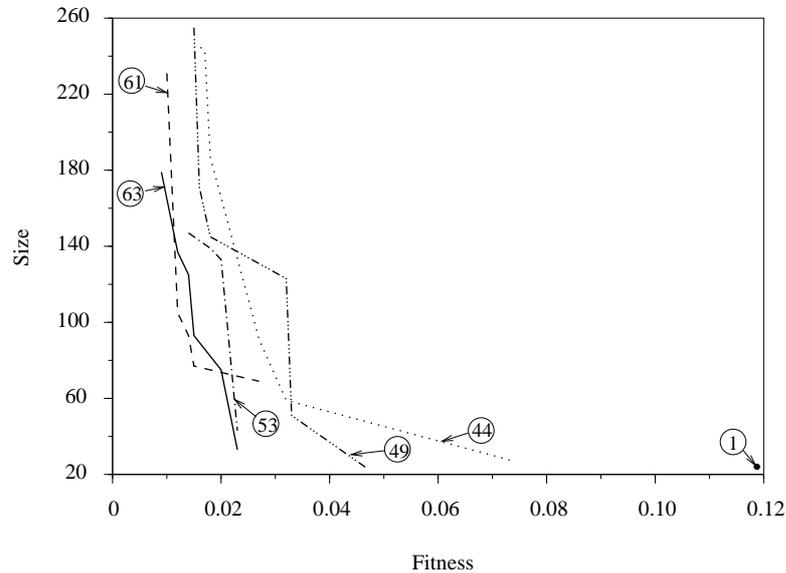


Figure 4.1: The projection of the nondominated individuals on the (fitness,size) plane when using criterion B, $k=500$. Each line shows the nondominated individuals of the population at the indicated generation. In the first generations there is only one nondominated individual in the population.

tial population, better convergence can be obtained. As far as program size is concerned, similar code growth is observable for Pareto selection starting from different size initial programs. At the same time, in original GP, code growth is much more accentuated when larger initial programs are allowed. To sum up, *the Pareto method conducts to more accurate solutions at similar slow increase in program size, if in the initial population larger programs are allowed*, whereas original GP conducts to similarly accurate solutions at much higher increase in program size in the same setting. The solutions of Pareto methods are less accurate than the solutions of original GP, but the size of programs increases at a much lower rate.

4.4 Discussion and Future Work

In this chapter two methods were studied for reducing code growth in genetic programming systems:

- simplification of programs by an additional mutation operator; and

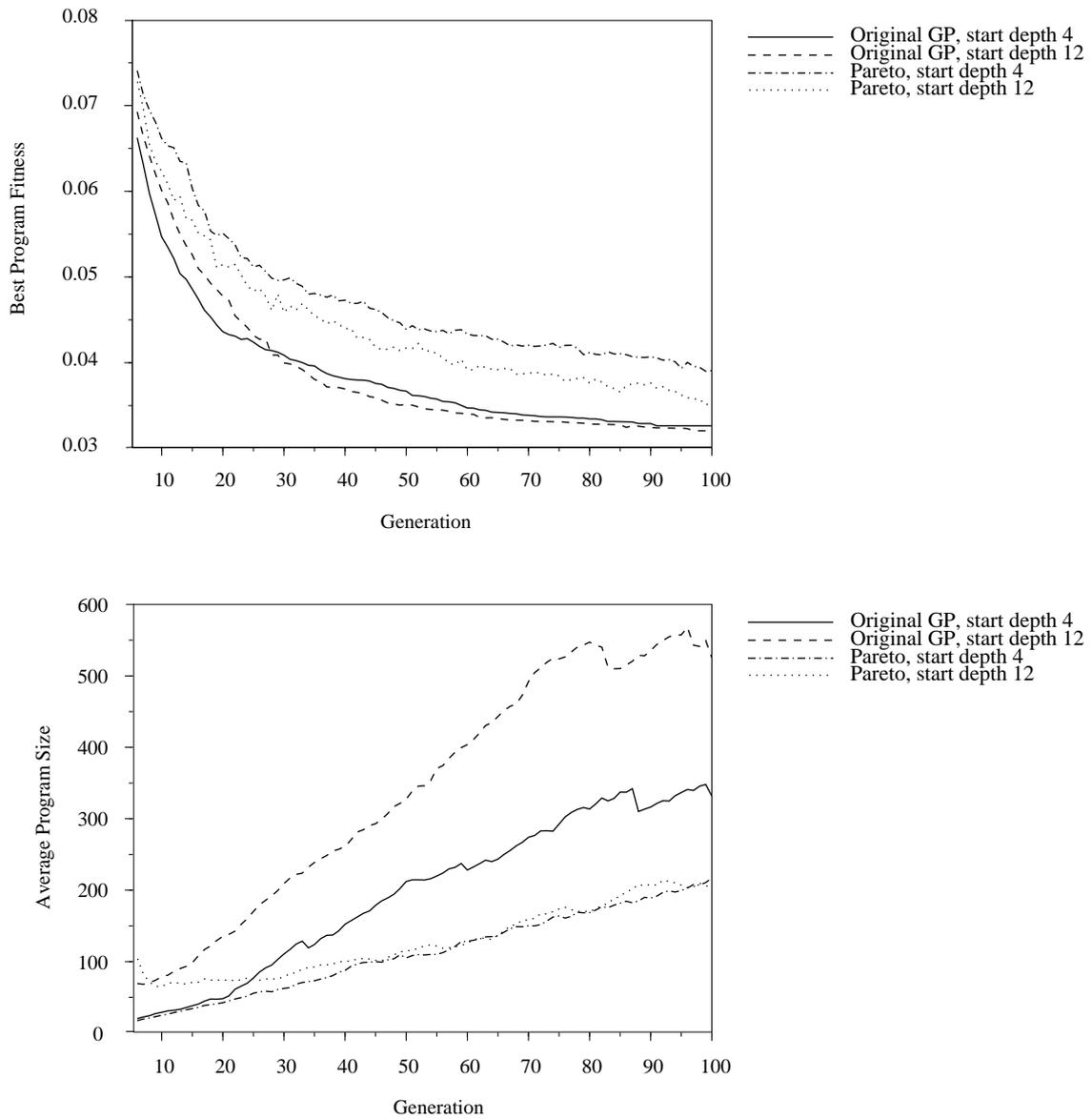


Figure 4.2: The evolution of program fitness and size in original GP and GP using Pareto criterion B, $k=500$. The influence of the size of programs contained in the initial random population.

- multiobjective optimization for fitness (accuracy) and size.

The simplification method We presented a method for limiting code growth in genetic programming where the mechanism for controlling the size of programs was distinct from the fitness-based selection mechanism. The control of code growth was realized by a special mutation operator, inspired by three forms of mutation in biology, namely neutral mutation, frameshift mutation and large DNA sequence rearrangement. This mutation operator consisted in simplifying the expression of the genetic program without changing its fitness value. In symbolic regression problems the special mutation operator consisted of algebraic simplification. The evolution of programs had two alternating phases:

- classical genetic programming - allowing code growth and intron formation (several generations); and
- simplification of programs - eliminating introns and reducing program size by means of the special mutation operator (one generation).

The experimental results show that code growth can be moderated through simplification – with little deterioration of performance – by choosing the right values for P and F . For the studied problems, using $P = 20\%$, $F = 2$ leads to solutions that are (1) significantly shorter than those of original genetic programming and meanwhile (2) not significantly worse in terms of fitness.

The Pareto method We introduced a new selection scheme that helps moderating code growth in genetic programming systems with no significant change in program accuracy. We used variants of the Pareto nondomination criterion for the selection of programs that are more fit and not much larger than the members of a comparison set.

We demonstrated the efficiency of the method on several symbolic regression problems. The differences in final program size (and also best program size) between the original tournament selection scheme and our proposed Pareto selection method are significant for all the examples in favor of the novel method. At the same time, we considerably reduced processing time: practically, in about half time as for the original GP we obtained half size programs of slightly worse accuracy.

When using these Pareto selection methods, if we increased the maximum allowed program depth for the initial population, we obtained more accurate solutions than with shorter initial programs, without any additional increase in program size.

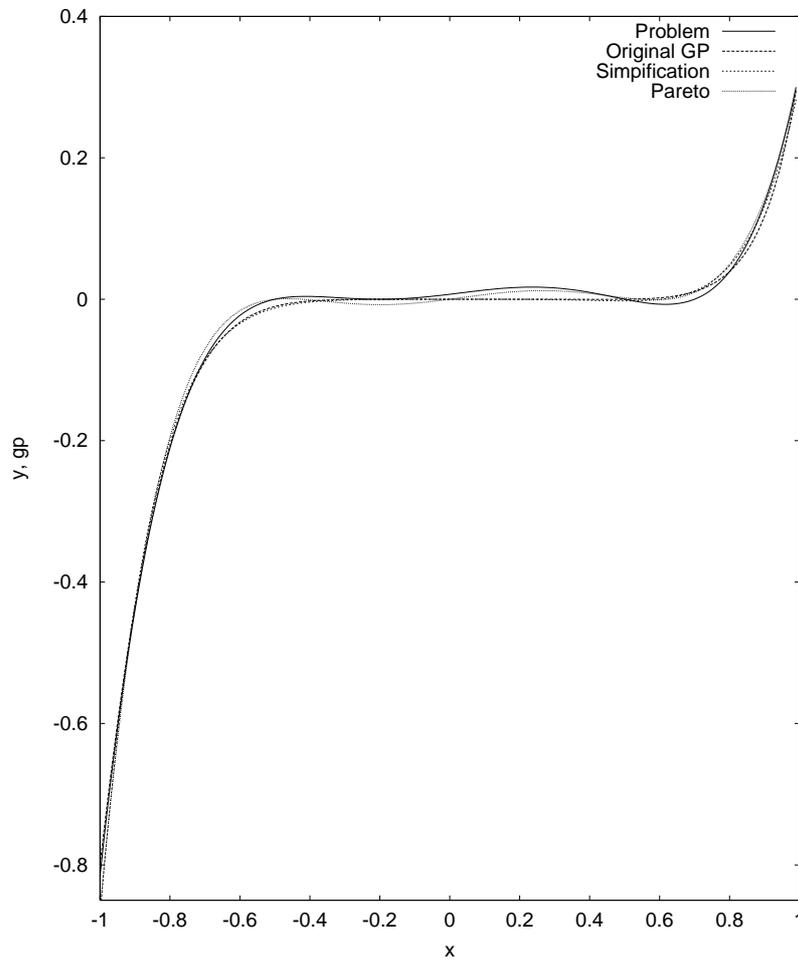
When comparing the results of the best parameter setting of each Pareto selection method on the test problems one by one, we cannot really differentiate. But if we examine the parameter settings of the same method on the different problems, we can say that the thresholded function is the most robust, since its best parameter setting is the same $k = 100$ for all the studied problems.

We have not implemented yet any efficient procedure for finding the best setting for the Pareto selection methods. According to our preliminary experiments, the coevolution of the parameter setting with the genetic programs themselves seems promising.

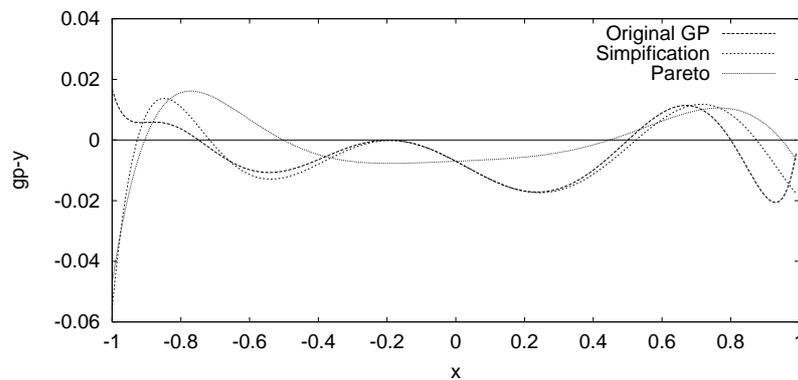
Comparison of the two methods Both studied methods were successful in reducing code growth in genetic programming without significant loss of accuracy. We show in Figure 4.3(a) one problem function and the solutions found by original genetic programming and by our methods. For a better distinction, the errors of the different methods are represented separately in Figure 4.3(b). As it can be seen, the accuracy of the different methods is indeed very similar.

One could choose one or the other method after examining the pros and cons of each one:

1. If there are simplification rules that are easy to formulate, the method using simplification could be advantageous. But the Pareto method is problem-independent and does not need additional programming.
2. The method based on simplification makes selection solely on the basis of fitness. The Pareto method prefers a better performing individual to a worse performing individual, if the first one is shorter or not much longer than the second one.
3. The Pareto method conducts to similar final program size irrespectively of the program size allowed in the initial population.



(a) Problem function and solutions.



(b) Errors of different solutions.

Figure 4.3: Function $y = (x + 0.2)^2(x - 0.5)(x + 0.5)(x - 0.7)$ and the solutions generated by the different GP methods.

Chapter 5

Fitness Sharing in Genetic Programming

An important problem of evolutionary algorithms is that throughout evolution they lose genetic diversity, that is, the individuals in a population resemble more and more each other (see Subsection 3.2.2).

Many techniques have been developed for maintaining diversity in genetic algorithms, but very few investigations have been done for genetic programs. The absence of diversity improving methods in genetic programming is mainly due to the fact that it is difficult to adapt a distance function for trees.

We introduce in this chapter a metric for genetic trees and apply it for fitness sharing¹, relying on our results presented at the Third European Conference on Genetic Programming EUROGP'2000 [20].

For verifying the results of fitness sharing we also define here a diversity measure for genetic programs.

5.1 Fitness Sharing

For genetic algorithms several niching techniques have been developed [41]. *Crowding* is a restricted replacement method, where the offspring of crossover and mutation replaces the most similar individual from a sample of the population. Another approach is the *restricted competition* among dissimilar individuals during selection. Both algorithms restrict their selection method, but in a different way.

Fitness sharing [25, 27] treats fitness as a shared resource of the population, and thus requires that similar individuals share their fitness. The fitness of each individual f_i is worsened by its niche count m_i . The niche count estimates crowding in the neighborhood

¹According to our knowledge, no other researcher applied fitness sharing to genetic programs before.

of individual i and is calculated over the individuals in the current population:

$$m_i = \sum_{j=1}^N S(d(i, j)),$$

N being the population size, $d(i, j)$ the distance of individuals i and j , and S the sharing function. S is a decreasing function, since sharing with a close individual should be greater than sharing with some farther individual. The typical sharing function has the form

$$S(d) = \begin{cases} 1 - (d/\sigma)^\alpha & \text{if } d \leq \sigma, \\ 0 & \text{if } d > \sigma. \end{cases}$$

The niche radius σ is fixed by the user at some minimum distance between peaks, and usually $\alpha = 1$. By using fitness sharing, the population does not converge as a whole, but convergence takes place within the niches.

The main drawback of fitness sharing is that the computation of the shared fitness for the entire population in each generation could be very time-consuming.

By using tournament selection, it is possible to overcome this difficulty. Oei, Goldberg and Chang [49] propose a scheme for combining sharing with binary tournament selection. They calculate the shared fitness by continuously updating the fitness as the new population is filled with individuals.

Yin and Gernay [69] propose the application of a clustering algorithm before sharing. Thus, the population is first divided into niches, and then the shared fitness of any individual is computed only with respect to the individuals that are in its niche.

Horn, Nafpliotis and Goldberg [29] propose a tournament selection scheme, where two individuals are picked randomly from the population, a comparison set is also picked randomly from the population and the two individuals are compared against the individuals from the comparison set. They implement sharing as counting the individuals from the comparison set that are in the niche of the two competing individuals. The individual with the smallest niche count is the best candidate.

The other difficulty of fitness sharing consists in determining the niche radius.

Beyond these difficulties, one has to make the appropriate choice of a genotypic or a phenotypic distance measure for the given problem.

5.2 Distance Measures for Genetic Programs

In the case of tree based genetic programming, a straightforward phenotypic distance measure is the Euclidean distance between the values of the two genetic programs on the fitness cases.

A genotypic distance measure should be a distance function on trees. Several distance-functions on trees have been proposed in different domains of artificial intelligence, such as inductive logic programming, bio-informatics, pattern recognition.

The *edit distance* between labeled trees was defined as the cost of shortest sequence of editing operations that transform one tree to the other [40, 58, 62]. An edit operation consists in inserting or deleting one node (subtree in [58]) or changing the label of one node. The cost of each edit operation could be specified separately [62] or the cost is unique for insertions, deletions, and changes, respectively [40]. The algorithms for computing the edit distance of trees are time-consuming. In the case of ordered labeled trees (where the order of children of a node is significant) T_1 and T_2 , the time complexity of the algorithm for the simplified edit distance [58] is $O(|T_1| * |T_2|)$, $|T|$ being the number of nodes in tree T . In the case of unordered labeled trees, the time complexity is exponential [71]. However, the edit distance has been used in pattern recognition [40] and instance-based learning [7]. But because of time complexity, the edit distance is not widely used in genetic programming [3].

Nienhuys-Cheng [46] defines a metric bounded by 1 that takes into consideration the depth of nodes in the trees:

$$d(p(s_1, s_2, \dots, s_n), q(t_1, t_2, \dots, t_n)) = \begin{cases} 1 & \text{if } p \neq q, \\ \frac{1}{2n} \sum_{i=1}^n d(s_i, t_i) & \text{if } p = q, \end{cases} \quad (5.1)$$

p and q being the root labels of the two trees. This metric is used then in inductive logic programming for measuring the quality of approximations to a correct program. This distance can be computed in $O(\min(|T_1|, |T_2|))$.

5.3 Diversity Measures for Genetic Programs

A diversity measure on any collection of objects must reflect how different are these objects from each other.

In genetic programming, a proper diversity measure reflects the structural difference of the genetic programs in a generation. As pointed out in [52], a good diversity measure could be the percentage of structurally distinct programs. But because of the computational expense of comparing the structure of genetic trees, Rosca [52] uses fitness and program size as measures of diversity. Since at any time there can be members of the population that have similar fitness, but very different structure, and similar size programs with very different structure, fitness and program size are not very reliable diversity measures.

Keller and Banzhaf [33] use an edit distance for determining the structural difference of the genetic trees. They propose a mapping of the space of genetic programs on a two dimensional distance space and then define the diversity of a population in this distance space. However, when applying the above mapping, so-called *collisions* can occur, i.e., more genetic programs could be mapped into the same position in the two dimensional distance space.

5.4 The Proposed Metric

We define here a metric for genetic programs that reflects the structural difference of the genetic programs and can be computed efficiently. We show then the experimental results for the metric on the symbolic regression problems treated in Chapter 4.

The distance of two genetic programs is calculated in three steps:

1. The two genetic programs (trees) to be compared are brought to the same tree-structure (only the contents of the nodes remain different).
2. The distances between any two symbols situated at the same position in the two trees are computed.
3. The distances computed in the previous step are combined in a weighted sum to form the distance of the two trees.

We construct the distance function step by step, as follows.

We make the convention that before comparing two trees of different structure, we complete them by NULL nodes so that the resulting trees have the same structure, as it is shown in Figure 5.1. In the case of commutative operations (unordered trees) we do

not know in advance which pairing of subtrees is better. Trying out each possible pairing makes the distance computation exponential in tree depth. When the trees have different arities, the missing subtree(s) could also be completed by NULL nodes. In this case, it would be difficult to choose where to insert the missing subtree. Therefore, in all of our experiments we use binary ordered trees.

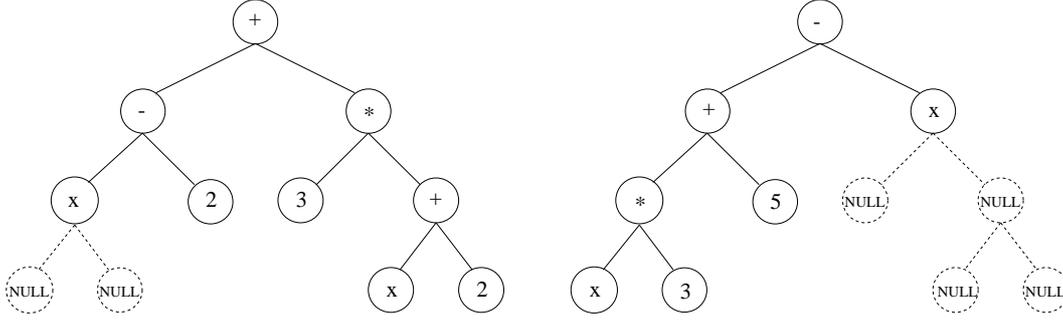


Figure 5.1: The trees are completed by NULL nodes to have the same layout.

The trees that can be constructed by genetic programming contain functions and terminals (variables and constants). The functions may be grouped in several groups according to their similarity (E.g. addition and subtraction, multiplication and division for trees encoding arithmetic expressions). The terminal set is initially differentiated into variables and constants, but one could further partition the variable set.

We partition the set of symbols into n sets A_0, A_1, \dots, A_n ; A_0 being the set containing NULL and A_1 the set containing the constants. We define the set A_0 for the single element NULL in order to treat the comparison between some node and NULL in the same way as a comparison between two elements of different sets.

Let $\delta : \{0, 1, \dots, n\} \times \{0, 1, \dots, n\} \rightarrow (0, +\infty)$ be a function with properties:

$$\delta(i, j) \leq \delta(i, k) + \delta(k, j), \quad (5.2a)$$

$$\delta(i, j) = \delta(j, i), \quad (5.2b)$$

$$\delta(i, i) < \delta(j, k), \quad j \neq k, \quad (5.2c)$$

for all $i, j, k \in \{0, 1, \dots, n\}$.

We define the distance between two different elements $x \in A_i$ and $y \in A_j$ as

follows:

$$d(x, y) = \begin{cases} 0 & \text{if } x = y, \\ C \frac{|x-y|}{\max_{v,w \in A_1} |v-w|} & \text{if } i = j = 1, \\ \delta(i, j) & \text{otherwise,} \end{cases}$$

C being a constant. By choosing $C < \delta(i, j), \forall i, j \in \{0, 1, \dots, n\}$, the distance of two constants becomes less than the distance of two other elements. With this distance definition, we provide that the distance between two elements of the same set is less than the distance between two elements of different sets.

Now, after defining the distance between elements, we can define the distance between trees. We take into consideration the fact that a difference at some node closer to the root could be more significant than a difference at some node farther from the root. First we complete the trees by NULL nodes, so that both trees have the same structure. Let $T_1 = p(s_1, s_2, \dots, s_m)$ be the tree with root p and subtrees s_1, s_2, \dots, s_m , and $T_2 = q(t_1, t_2, \dots, t_n)$ the tree with root q and subtrees t_1, t_2, \dots, t_m , respectively. Then, we calculate the distance between trees T_1 and T_2 as:

$$dist(T_1, T_2) = \begin{cases} d(p, q) & \text{if neither } T_1 \text{ nor } T_2 \text{ have any children,} \\ d(p, q) + \frac{1}{K} \sum_{l=1}^m dist(s_l, t_l) & \text{otherwise,} \end{cases} \quad (5.3)$$

where $K \geq 1$ is a constant, signifying that a difference at any depth r in the compared trees is K times more important than a difference at depth $r + 1$. By choosing different values for K , we could define different shapes for the neighborhoods (see Section 5.6).

It can be easily proved that all conditions for $dist$ to be a **metric** [24] are met:

$$\begin{aligned} dist(T_1, T_2) &\geq 0, \\ dist(T_1, T_2) &= 0 \Leftrightarrow T_1 = T_2, \\ dist(T_1, T_2) &= dist(T_2, T_1) \text{ (symmetry),} \\ dist(T_1, T_2) &\leq dist(T_1, T_3) + dist(T_3, T_2) \text{ (triangle inequality).} \end{aligned}$$

5.5 The Proposed Diversity Measure

Many diversity measures can be defined by using the metric introduced in the previous section. A refined diversity measure would first partition the population into niches,² and then consider the niches both individually and jointly.

However, the most straightforward measure is *the mean distance of two individuals in a population*.

We compute the mean distance of an individual to some given individual i in a population as

$$meandist(i) = \frac{1}{N-1} \sum_{1 \leq j \leq N} dist(gp_i, gp_j)$$

where N denotes the size of population Pop , and gp_i the genetic tree for individual i .

Then our diversity measure can be computed as

$$Diversity(Pop) = \frac{1}{N} \sum_{i=1}^N meandist(i) = \frac{2}{N(N-1)} \sum_{1 \leq i < j \leq N} dist(gp_i, gp_j).$$

This diversity measure does not include niches, but it clearly indicates diversity: the larger is the mean distance of two individuals, the more distributed are the programs in the program space. In addition, our diversity measure is independent of the technique used for maintaining population diversity and can be used for verifying the effects of any such technique.

5.6 Empirical Validation

We used the metric for the symbolic regression problems presented in Chapter 4. The general parameter setting was as shown in Table 4.2.

We define the distance function for two genetic programs in the case of symbolic regression of a function of one variable, with function set $F = \{+, -, *, /\}$. All of the functions take two arguments, thus the resulting trees are binary. We consider ordered

²This requires the application of some cluster analysis method.

trees. We partition the set of symbols as follows:

$$\begin{aligned} A_0 &= \{NULL\}, \\ A_1 &= \{\text{real values} \in [-1, 1]\},^3 \\ A_2 &= F, \\ A_3 &= \{x\}. \end{aligned}$$

Then we choose function δ , such that it satisfies requirements (5.2). A possible δ is given by the matrix

$$\begin{array}{c} 0 \quad 1 \quad 2 \quad 3 \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{bmatrix} 0 & 5 & 5 & 5 \\ 5 & 0 & 2 & 3 \\ 5 & 2 & 1 & 3 \\ 5 & 3 & 3 & 0 \end{bmatrix} \end{array}.$$

We experimented fitness sharing with $K = 1$, $K = 2$ and $K = 10$ in the expression of the distance function (5.3). We set the distance of two symbols $d(x, y) \in [0, 5]$, so that the distance of two nodes at the same position in the trees counts for the final distance with a value depending on this position, as it is shown in Table 5.1.

Table 5.1: The difference of two nodes counted for the distance of the trees containing them.

Position (depth)	Range of added value		
	K=10	K=2	K=1
0 (root)	0-5	0-5	0-5
1	0-0.5	0-2.5	0-5
2	0-0.05	0-1.25	0-5
3	0-0.005	0-0.625	0-5

Thus, if the distance of two trees is a value less than 0.005 in the case of $K = 10$, they only differ at some node at depth 3 (or more) or they differ just at the values of constants.

We experimented with niche sizes corresponding to the values of K , as shown in Table 5.2. By selecting different values for K , we can experiment different niche shapes. For

³We use the set of real values with precision 10^{-3} .

Table 5.2: The ranges for niche size.

K	Niche size
10	$[5 * 10^{-3}, 1.0]$
2	$[5 * 10^{-3}, 0.5]$
1	$[1, 10]$

$K = 1$, for a certain tree, any other tree that differs from it at just one internal node, is in its neighborhood of size $\sigma \geq 1$. A niche size $\sigma < 1$ in this case would restrict only to differences in the values of constants. In the meantime, for $K = 10$, if $\sigma = 1$, for a certain tree, any other tree with the same root is in its neighborhood. But for $K = 10$ and $\sigma = 5 * 10^{-3}$, the neighborhood of a tree contains only the trees with differences at depth 3 or more (and slight differences in the values of constants at depth 1 or 2).

Generally, when fitness sharing was applied, the accuracy of the solution was better than in the case of original genetic programming. For instance, the error of the final solution (averaged over 50 runs with the same parameter setting) was 0.035 for original genetic programming and 0.017 for fitness sharing with $K = 10$ and niche radius $\sigma = 0.1$.

For larger niche radii, the error increases as an effect of fitness sharing with too many neighbors (E.g., for $K = 10$ and $\sigma = 0.5$, sharing is performed among individuals that can differ at one node at the first depth or any number of nodes at depth more than 1). But the scope of fitness sharing is to maintain diversity in the population. We show in Figure 5.2 the evolution of diversity according to our diversity measure in the case of original genetic programming and fitness sharing with several settings.

Original genetic programming converges to populations of programs with very similar structure, but not necessarily similar fitness. In the meantime, when applying fitness sharing, the populations (in any generation) contain individuals that are different from each other. This difference depends on the predefined niche size.

The main characteristics of fitness sharing in genetic programming can be summarized as follows:

1. By the application of fitness sharing, fitness is shared among not highly fit individuals containing the same unfit code. Their shared fitness is even worse, and they have less chance of being selected. *Thus, the accuracy of solutions found by fitness sharing is better than the accuracy of solutions found by original genetic programming.*

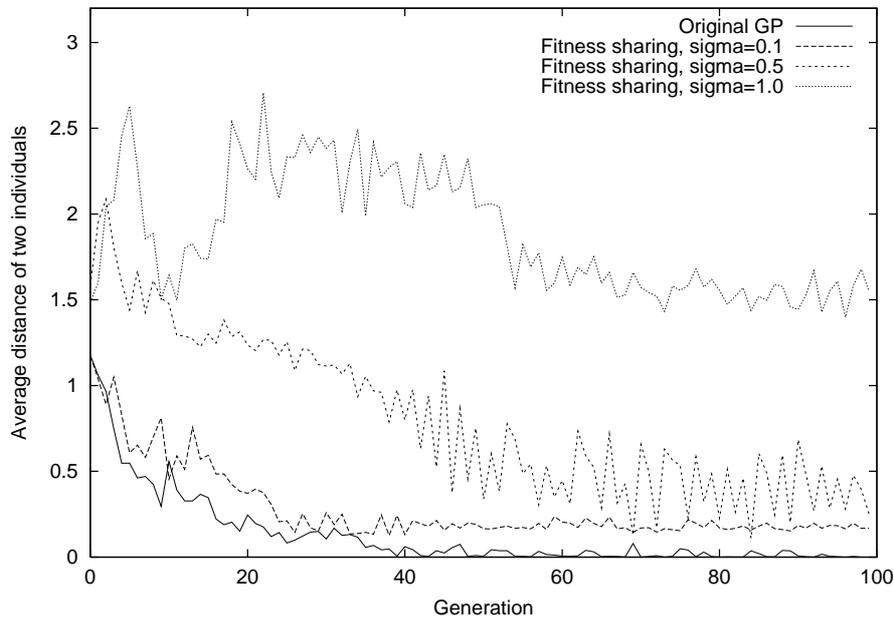


Figure 5.2: Population diversity for original genetic programming and fitness sharing ($K = 10$).

2. By using tournament selection and fitness sharing, if two competing individuals are of similar (unshared) fitness, the individual with less neighbors or farther ones is preferred to the individual with more or closer neighbors. In other words, the individual that has less common parts with the other individuals from the tournament is selected. In this way, *the diversity of the population is maintained at a certain level depending on niche size.*
3. *The diversity of a population is an increasing function of niche size.* Practically, since few individuals are allowed in a niche, when niche size is large, the evolved individuals are generally far from each other. When niche size is small, the evolved individuals need only to be at some distance larger than this niche size, thus not very far from each other.
4. When niche size is large, and the individuals are generally far from each other, it is more difficult to obtain an accurate solution: the search for solution becomes one-directional within each niche. But when niche size is very small, diversity is low and the population converges to very similar individuals, almost as if no fitness sharing were used. Thus, *a medium niche size should be chosen so as to allow at the same time the evolution of different programs and the convergence to accurate solutions.*

5.7 Discussion

In this chapter we introduced a new metric for genetic programs that reflects their tree-structure and can be efficiently computed.

We used the metric for applying fitness sharing to genetic programming. By the application of fitness sharing, we obtained genetic programs of better fitness. We applied fitness sharing (as it is usual in genetic algorithms) for maintaining population diversity. Thus, for interpreting its result, we defined a new measure of population diversity using our metric based on the structural difference of programs. Our experiments prove that fitness sharing maintains population diversity in genetic programming at a level depending on niche size.

Thus, we successfully adapted the technique of fitness sharing to genetic programming: we obtained more accurate solutions while population diversity was maintained. Since we do not know in advance, what is the appropriate niche size for a given problem, we plan to study the effects of adaptively changing niche size throughout evolution. This adaptation will be driven by population diversity.

Chapter 6

Genetic Programming and Decision Trees in Synthesis of Four Bar Mechanisms

Four bar mechanisms are basic components of many important mechanical device. The kinematic synthesis of four bar mechanisms is a difficult design problem.

We present here a novel method that combines the genetic programming and decision tree learning methods. We give a structural description for the class of mechanisms that produce desired coupler curves. For finding and characterizing feasible regions of the design space constructive induction is used. Decision trees constitute the learning engine and the new features are created by genetic programming.

The results in this chapter are based on our papers [14, 16, 18].

6.1 Introduction

A mechanism is defined as an arrangement of machine elements that produce a specified motion [55]. The synthesis of a mechanism is the process of combining parametric elements into a mechanism that shows complex behavior. We investigate here a simple, but practically important class of mechanisms: four bar mechanisms. The utilization of four bar mechanisms ranges from simple device, such as the windshield-wiping mechanism and the door-closing mechanism to complicated ones, such as the rock crusher, the sewing machine, the round baler and the suspension system of automobiles [48, 65]. Figure 6.1 shows the structure of the four bar mechanism, where a represents the fixed link, b the input link, c the coupler link, d the follower link, and point P the tracer point.

From the kinematic point of view four bar mechanisms can be designed for: (1) path generation, (2) rigid body guidance, and (3) function generation. The current application is related to the path generation problem: *given certain path fragments, find the mechanism (i.e., its structural parameters) whose coupler curve contains these fragments.* A path fragment is given as an ordered set of points. In the case of path generation with prescribed timing, there is associated with each point an input angle (α in Figure 6.1) as

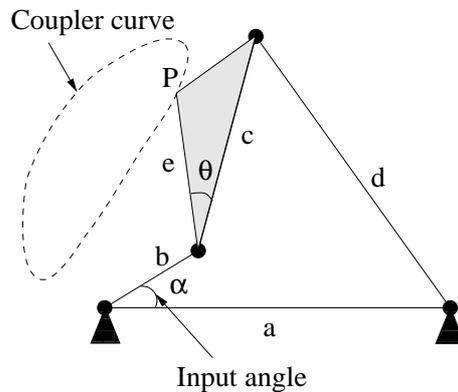


Figure 6.1: The four bar mechanism layout.

well. Since for points more than five the path generation problem is overconstrained, [55] the acceptable tolerance between the input path fragments and the coupler curve is also specified.

For path generation with prescribed timing, the classical analytical approach [55] is limited to the case of five specified points. Otherwise, there is no general recipe for choosing the structural parameters for the mechanism that approximates a given path. Recent work has been done for finding numerical methods for the general case: e.g., mechanism synthesis is considered as a nonlinear programming problem and an exact gradient method is used for the dimensional synthesis of mechanisms [42].

For the situation when more than five points are specified there have been developed some variant based methods, such as case-based reasoning [8]: after a multi-level case retrieval process, adaptation is accomplished by simple transformation rules (increasing or decreasing the length of one link by a small percent). However, the method is applicable only in cases when the coupler curve contains no crossings or there is a close-enough curve in the case-base. In another work [26] neural networks are used for learning and synthesizing mechanisms for coupler curves similar to the ones stored in the knowledge base. Since the coupler curves are stored as bitmaps, pattern matching is used for finding similar curves. In [63] genetic algorithms are used for synthesizing a mechanism that generates a given coupler curve in a constrained environment: the fixed link must be within a given area and the link lengths are also limited.

Unlike the previous approaches, we do not generate a single mechanism, but give structural constraints for mechanisms whose coupler curves contain the desired path fragments. Any mechanism satisfying these constraints will meet the input requirements.

If there are further restrictions for the acceptance of a mechanism (such as dimensional limits of the links), the structural description should be refined accordingly.

6.2 The Design Method

The goal of this work is to provide such a description of the feasible mechanisms that can be exploited by the designers of mechanisms.

Particularly, we create the structural description of the four bar mechanisms that satisfy the input requirements. That is, given some curve fragments and a corresponding tolerance, the coupler curve generated by any acceptable mechanism must be within the given tolerance limit from these curve fragments. The admissible mechanisms are given by constraints on their structural parameters. Any mechanism satisfying these constraints can be designated as a solution for the given family of path generation problems.

The method consists of the following steps (where steps 2-4 represent the definition of the actual design problem):

1. creation of a catalog of four bar mechanisms;
2. specification of input requirements;
3. classification of the elements of the catalog;
4. generation of the structural description.

The main result is that the design space can be described by constraints on the structural parameters (a, b, c, d, e, θ) of the mechanisms. For any desired path fragment, we give the structural description of the mechanisms that produce coupler curves similar to it. By this structural description the class of mechanisms that produce similar coupler curves is given. Hoeltzel and Chieng [26] make a classification of four bar mechanisms based on the form of coupler curve, but they provide no structural description for the members of a class.

Our catalog of four bar mechanisms contains 7276 elements and we created it as a computer version of the classical catalog of Hrones and Nelson [30]. Each element consists of the structural and functional description of a mechanism. The structural description contains the parameters of the mechanism, as shown in Figure 6.1. The functional description is the coupler curve generated by the mechanism and it is recorded as an

ordered list of points. We call mechanism space the universe of four bar mechanisms. The dimensions of this space are the structural parameters of mechanisms. The elements of the catalog are points of the mechanism space, with the structural parameters $a \in \{1.5, 2, 2.5, \dots, 6.5\}$, $b = 1$, $c \in \{1.5, 2, 2.5, 3, 3.5, 4\}$, $d \in \{1.5, 2, 2.5, 3, 3.5, 4\}$, e and θ taken for 50 sampled points on a rectangle attached to the coupler link of the mechanism (see Figure 6.2). This rectangle extends the coupler link c in directions parallel and perpendicular to it with a distance equal to the length of the input link b .

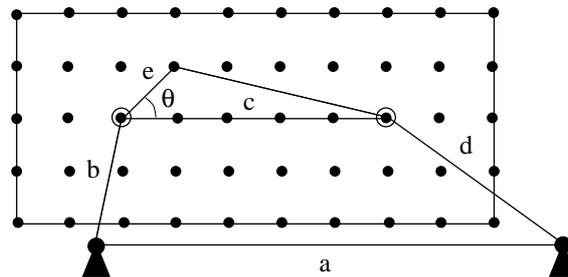


Figure 6.2: The sampled points on the coupler link.

In the second step the desired curve fragments are specified as a list of points and corresponding input angles (i.e., our problem is path generation with prescribed timing) and the tolerance is also given.

In the classification step the mechanisms are grouped into two classes: the **positive** and the **negative** class. A mechanism is positive, if its coupler curve fulfills the input requirements. The mechanism is negative in the opposite case. *A coupler curve corresponds to the input requirements when it can be moved (translated and rotated) so that its similarity to the desired curve is within the tolerance limit.* Classical methods only compare the coupler curve to the desired curve without translation and rotation, meaning that the position of the mechanism in the plane is predefined. We remove this restriction and compute the similarity of two curves after bringing them as close as possible by translating and rotating one of the curves. We compute the similarity value of a coupler curve to the desired curve (fragments) in the following way (as shown in Figure 6.3):

1. To each point of the desired curve we associate a point of the coupler curve according to the input angle (timing) corresponding to that point of the desired curve, that is, we associate a point list from the coupler curve to the given point list of the desired curve. Generally, there are several possible matchings. (In Figure 6.3 we

- show only the best matching for the given example.)
2. We calculate the similarity value for each such possible matching:
 - (a) We translate and rotate the point list obtained in step 1 so that the associated points of the desired curve and the coupler curve get as close as possible (this is equivalent to translating and rotating the entire coupler curve, but we consider only the point list since for the comparison these points are needed).
 - (b) We compute the distances of the associated point pairs.
 - (c) We assign the maximum of these distances as similarity value for the given matching.
 3. We select the matching with the smallest similarity value and designate this value as similarity of the coupler curve and the desired curve. If this similarity value is within the tolerance limit, then all the points of the coupler curve selected in step 1 are close enough to the desired curve, and thus, the mechanism belongs to the positive class. Otherwise, the mechanism is classified as negative, like the example shown in Figure 6.3.

The key issue is the generation of the structural description of the positive class. The structural description consists of a set of constraints for the structural parameters. The description can be written as a DNF (disjunctive normal form) formula. Two machine learning methods have been applied: decision tree induction (by the C4.5 program) and genetic programming.

Decision tree induction was effective in the cases when the elements of the positive class were situated in a convex region of the mechanism space and the mechanism space could be partitioned by planes parallel with the axes. However, it produced too many errors in the other cases. We needed a method that could create descriptions corresponding to as many elements of the positive class as possible, regardless whether they were situated in a convex region of the space and allowing other partitions. Genetic programming was our next choice. Unfortunately, in most of the cases the descriptions produced by genetic programming were not accurate enough. Two types of errors were encountered: (1) too many members of the negative class were classified as positive and (2) some members of the positive class were classified as negative. In this way the frontiers of the feasible regions of the mechanism space were misrepresented. Since decision tree induction is

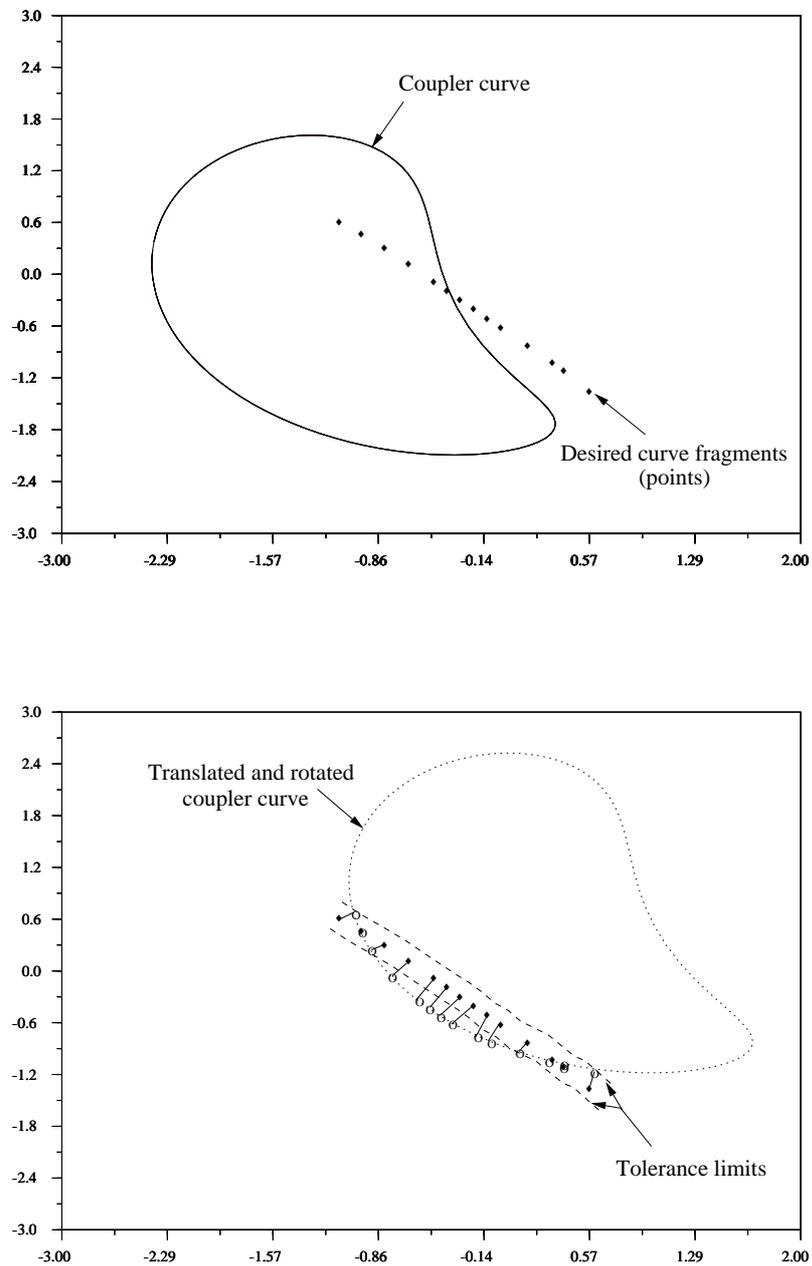


Figure 6.3: Computing the similarity of the coupler curve (for mechanism with $a = 1.5$, $b = 1$, $c = 1.5$, $d = 1.5$, $e = 1.41$ and $\theta = 2.35$) to the desired path fragments. For better intelligibility, timing is not indicated. This mechanism is classified as **negative**.

very fast and accurate when presented with the appropriate attributes, and, on the other hand, genetic programming is a plausible tool for creating new attributes, we decided to apply constructive induction [68]. This step is discussed in detail in the next section.

6.3 Creating the Structural Description

Generating the description of one class can be seen as partitioning the space into two regions: positive and negative. The generated description corresponds to the positive region, and no element situated in the negative region satisfies this description.

6.3.1 Decision Tree Learning

Decision tree induction is a commonly used method for concept learning [51]. The input data is a set of examples of the concept to be learned. An example consists of a set of attributes and belongs to a class. When building up the tree and selecting the next attribute to be tested at a certain node, the information gain¹ is computed for each candidate attribute. Then, the attribute with the best gain is selected.

For studying the power of decision trees in our problem, we used the C4.5 system [51]. Each example is a mechanism, having its structural parameters as attributes and the classification computed in the previous step (i.e., positive or negative). Since the attributes have continuous numeric values, at each decision node the cases are separated into two groups according to whether the value of the tested attribute is less or greater than a threshold value.² Case studies are presented in Section 6.4.

A typical example, where C4.5 produced many errors, is shown in Figure 6.4. Attributes $b = 1$, $d = 3$, $e = 1.41$, $\theta = 2.36$ and the distribution of positive and negative examples in the mechanism space over coordinates a and c is drawn. For this cross-section, the decision tree produced by C4.5 misclassifies 3 of the 30 examples. The explanation is that this plane could not be partitioned by straight lines parallel to the axes of the coordinate system. Thus, using the original attributes C4.5 could not produce an exact classification. If the attributes were transformed, a better partition of the plane could be

¹The information gain is defined as the increase in information content when the set of examples is partitioned in subsets according to some criterion, such as a value of an attribute.

²In this case the information gain is computed for several candidate threshold values.

found. An exact description (as it is drawn in Figure 6.4) of the positive region of the plane found by our program is

$$(a - 1.55) * (2.31/c + 1.56 - a) > 0.$$

Overcoming this difficulty by creating new attributes is the subject of Section 6.3.3.

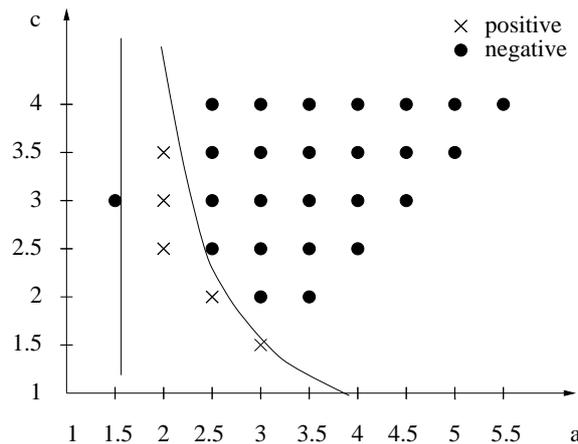


Figure 6.4: The cross-section of the mechanism space.

6.3.2 Genetic Programming

As an alternative, we also generated structural descriptions of mechanism classes by using genetic programming. We used the genetic programming paradigm with the setting shown in Table 6.1.

The search space is a six-dimensional cube corresponding to the parameters of the four bar mechanisms. Here, the goal of genetic programming is to find descriptions for the positive region of the space. Usually, the positive examples are not situated in a convex region of the search space and often there exist no linear delimiters between positive and negative regions, as shown in the two-dimensional cross-section of Figure 6.4.

Initially we defined a genetic program as an inequality on the structural parameters to be satisfied by some feasible mechanisms and possibly no infeasible mechanism. For instance, a possible inequality is $a/b \sin(\theta) - 1.2 > 0$. This condition is satisfied by all the mechanisms with $a \geq 2, b = 1, 0.79 \leq \theta \leq 2.35$, but only some of these mechanisms are feasible. In fact, a mechanism should rather satisfy a set of such inequalities for being a feasible mechanism. Thus, we represented a genetic program as a tree with a

fixed number of branches where each branch represented an inequality on the structural parameters of the mechanisms. The genetic program was then evaluated as the union of the inequalities contained in its branches.

Table 6.1: Genetic programming parameter setting.

Objective	Evolve the structural description of the four bar mechanisms contained in the positive class
Terminal set	$a, c, d, e, \sin(\theta), \cos(\theta)$, ³ real numbers $\in [-10, 10]$
Function set	$+, -, *, /$
Fitness cases	The mechanisms of the catalog
Population size	50
Crossover probability	90%
Mutation probability	10%
Selection method	Tournament selection, size 10
Termination criterion	none
Maximum number of generations	50
Maximum depth of tree after crossover	20
Initialization method	Ramped Half and Half

We used both crossover and mutation as recombination operators. The offspring of crossover were obtained from the two parents by selecting one subtree of each parent and exchanging these subtrees. We used point mutation, i.e., one node of the tree was randomly selected and mutated. In the case of internal nodes, mutation consisted in randomly changing the function represented by the node. In the case of leaf nodes, mutation was different for terminals representing the structural parameters of mechanisms and real constants. A structural parameter was mutated to another structural parameter, and a constant was mutated by changing its value.

The fitness cases were the mechanisms of the catalog given by their structural parameters and class rating (positive or negative). The genetic program was run on the fitness

³Since $b = 1$ for all the examples, we use here a for the ratio of the fixed link to the input link, and similarly, for any other link we use the ratio of that link to the input link. In this way, the genetic programs are valid dimensionless expressions.

cases, and its fitness value was computed as follows:

$$\begin{aligned} err(i) &= \begin{cases} 1 & \text{if } class(i) = 0 \text{ and } gp_class(i) = 1 \\ 0 & \text{otherwise} \end{cases} \\ fitness &= \begin{cases} 0 & \text{if } \forall i \ gp_class(i) = 0 \\ 1 - \frac{1}{N} \sum_{i=1}^N err(i) & \text{otherwise,} \end{cases} \end{aligned}$$

where $err(i)$ = the error of the genetic program for fitness case i , $class(i)$ = the class rating of the mechanism contained in the fitness case (0 for negative and 1 for positive), $gp_class(i)$ = the class rating given by the genetic program, and N = the number of fitness cases.

Thus, a 100 % fit individual is an expression corresponding to some positive examples and no negative example. An expression satisfied by some positive and some negative examples cannot make the distinction between the positive and negative examples, and therefore, it is penalized by a worse fitness value.

We used a fixed number of inequalities keeping the representation simple. When fewer inequalities were needed, some branches of the genetic program represented inequalities that were always true.

6.3.3 Constructive Induction

Both previous methods – when applied alone – had some shortcomings, hence the idea to apply them together. Decision tree learning is not always able to separate the two classes because the structural attributes presented to it are not the most relevant ones. There are several notions of relevance, let us consider **incremental usefulness** of [6]:

Given a sample of data S , a learning algorithm L , and a feature set A , feature x_i is incrementally useful to L with respect to A if the accuracy of the hypothesis that L produces using the feature set $\{x_i\} \cup A$ is better than the accuracy achieved using just the feature set A .

In other words, decision trees (or any learning algorithm) can perform better, if presented with more useful attributes. New attributes could be generated as *logical expressions* over the original attributes [43, 50]. Since the original attributes (the structural pa-

rameters of mechanisms) are numerical values, a straightforward method is the generation of *arithmetic expressions* over the original attributes [5].

Having in mind the definition for usefulness, we used C4.5 as the learning engine for the constructive induction and we applied genetic programming for attribute (feature) generation. The new scheme for generating the description is shown in Figure 6.5.

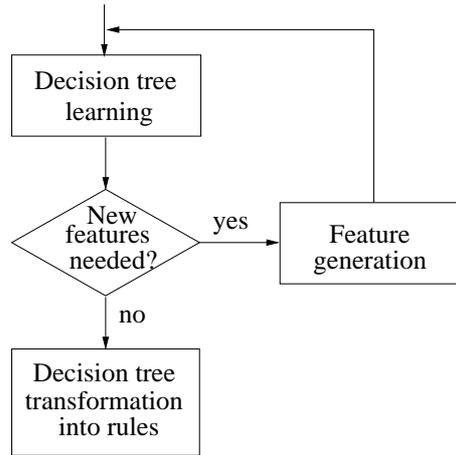


Figure 6.5: Constructive induction.

Thus, genetic programming proposes new attributes, and C4.5 uses them in addition to the original ones. The genetic programs are arithmetic expressions on the original attributes. The fitness measure is modified in order to include both kinds of errors: (1) misclassification of positive cases and (2) misclassification of negative cases. The error of the genetic program was computed as:

$$err(i) = \begin{cases} n & \text{if } class(i) = 0 \text{ and } gp_class(i) = 1 \\ p & \text{if } class(i) = 1 \text{ and } gp_class(i) = 0 \\ 0 & \text{otherwise} \end{cases}$$

We experimented with different values for n and p ($p \gg n$ or $n \gg p$), since we wanted to find good descriptions for both the positive and the negative examples. By obtaining a good description of the negative examples and presenting the attribute to C4.5, we expected C4.5 to eliminate many negative examples at the beginning. The rest of parameter settings for genetic programming are the same as described in Section 6.3.2.

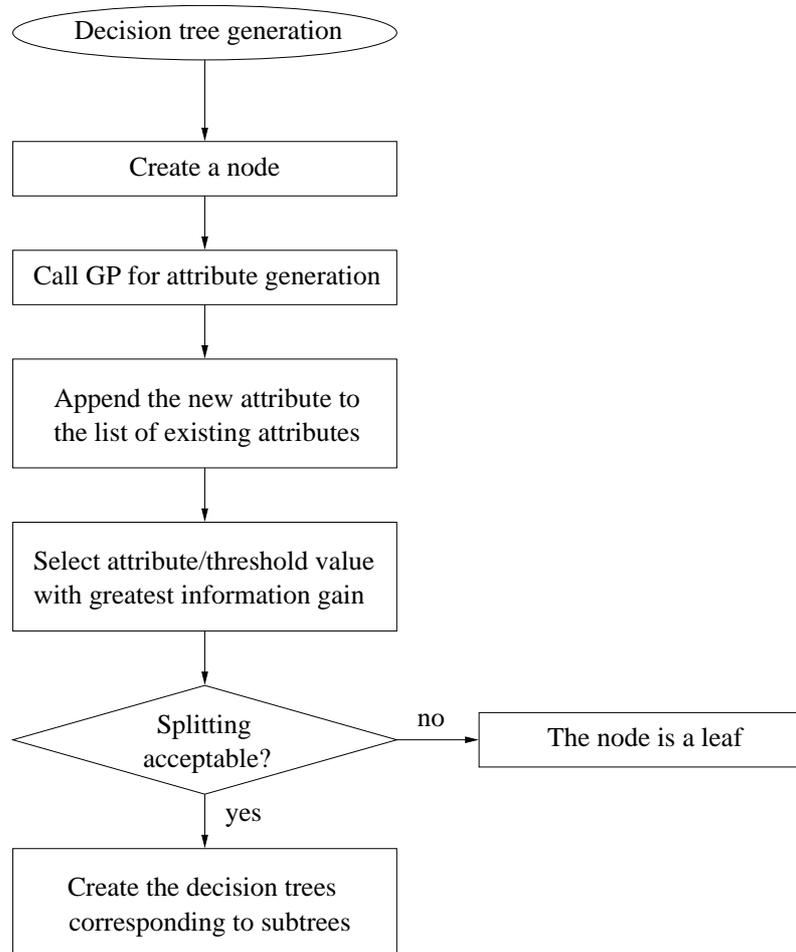


Figure 6.6: Decision tree construction.

Initially, we generated the new features separately from decision tree construction, as it is shown in Figure 6.5. The new features created by genetic programming tended to cover as many positive examples as possible. But when constructing the decision tree, at each node it is sufficient that the chosen feature describe the examples to be classified at that node. *There is no need for a general feature.* If we had found such a general feature, we would not have needed decision tree induction at all. Thus, in order to get new features corresponding to each node of the decision tree, we introduced the attribute generator at the level of node creation in the decision tree builder. We modified the fitness measure for the genetic programs to be the information gain (as defined in [51]) of the feature represented by a genetic program. The decision tree induction step is modified as it is shown in Figure 6.6.

By creating the new features at the construction of nodes in the decision tree, we obtained more accurate descriptions.

6.4 Experimentation

In this section two design cases are discussed in detail. We chose a circular and a linear path fragment, because these are very important fragments of the paths needed in practical mechanism design. So far approximate circle tracing mechanisms have been synthesized through laborious computation algorithms. Only the recent results of Ceccarelli and Vinciguerra allow their efficient synthesis [9]. Approximate straight lines are required in the case of level luffing cranes used on many docks to load and unload cargo [65]. Another application of straight line generating mechanisms is in strip chart recorders. In addition, a linear path fragment followed by a circular one is required in the film-advance mechanism of any movie camera [10, 48, 55], as shown in Figure 6.7.

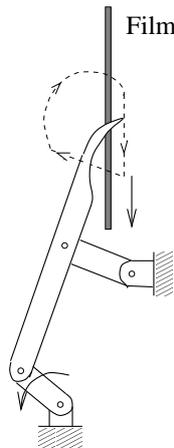


Figure 6.7: The film-advance mechanism of a movie camera.

For the circular path fragment (Figure 6.8) we found a 100% correct description by using just decision trees, so that no other method could perform better. In the second case the description given by decision trees had errors, thus constructive induction was needed.

The desired curve fragments for the two case studies are given in Figures 6.8 and 6.9. The path fragment is given as an ordered list of points (the ordering is reflected in the numbering of points).

In the first case (see Figure 6.8), the mechanism space was divided to the positive

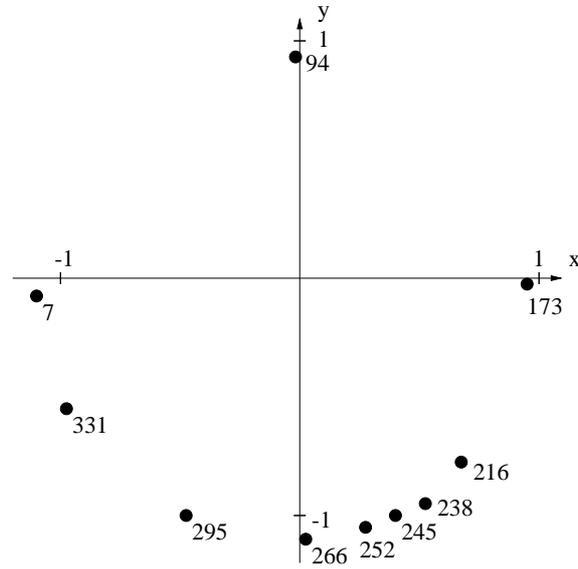


Figure 6.8: The first desired path fragment.

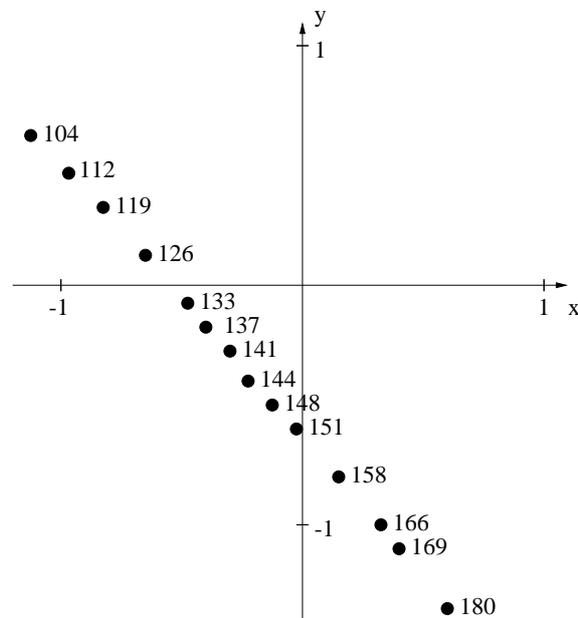


Figure 6.9: The second desired path fragment.

and negative classes and then C4.5 was applied. The produced decision tree is shown in Figure 6.10 (a). At each leaf-node the predicted class of the corresponding cases is given as “-” for the negative and “+” for the positive class.

The tree classifies all the cases correctly, so that for every example presented to it the real class is the same as the predicted class. The production rules can be derived from it by reading the paths from the root to the leaves. We show in Figure 6.10 (b) only the rules for the positive class, since any example that satisfies neither of these rules is negative.

```

e > 0.33 : -
e <= 0.33 :
|   theta <= 1.46 : -
|   theta > 1.46 :
|       |   c <= 1.5 : -
|       |   c > 1.5 :
|       |       |   a > 2.5 : +
|       |       |   a <= 2.5 :
|       |       |       |   c <= 3 : +
|       |       |       |   c > 3 : -

```

(a) The output of C4.5.

- 1: $e \leq 0.33$ and $\theta > 1.46$ and $c > 1.5$ and $a > 2.5$
- 2: $e \leq 0.33$ and $\theta > 1.46$ and $c > 1.5$ and $a \leq 2.5$ and $c \leq 3$

(b) The rules for the feasible mechanisms.

Figure 6.10: The decision tree produced by C4.5 and the corresponding rules.

For the same problem the best description (of the positive examples) produced by genetic programming is shown in Figure 6.11.

```

( AND
  ( > ( - ( / sin(theta) 6.4 )
        ( * ( * e sin(theta) ) sin(theta) ) ) 0 )
  ( > ( - sin(theta) cos(theta) ) 0 )
  ( > ( / ( / c sin(theta) ) ( - a c ) ) 0 )
  ( > ( - ( / 0.3 sin(theta) ) ( + a e ) ) 0 ) )

```

Figure 6.11: The best description created by GP.

We tried out different numbers of branches for the genetic trees. According to our observations, with fewer branches more accurate descriptions were found in fewer generations. Hence the idea of applying genetic programming for attribute generation in constructive induction.

The second desired path is a straight line approximated by the ordered point list of Figure 6.9. In this case, C4.5 produced a decision tree of size 47 containing 11 errors (i.e. misclassified cases). Our next step was the separate generation of new attributes by genetic programming, as it is shown in Figure 6.5.

Some of the attributes proposed by genetic programming are presented in Table 6.2. The third column (Type) specifies whether the attribute corresponds to the positive or the negative class. The last column (G) represents the generation when the attribute emerged. The best decision tree had 41 nodes and misclassified 6 cases.

Table 6.2: The new attributes proposed by GP.

Att.	Expression	Type	Fitness	G
1	$\sin \theta - 0.7$	p	87.2	6
2	$e - a + 1$	p	61.3	1
3	$d / \cos \theta$	n	76.4	2
4	$c - a - e - (1 + e) \cos \theta$	p	89.0	5
5	$a / 2.3 - 5.2 / \sin \theta$	n	97.0	2
6	$8.7 \sin \theta - 4.48$	n	84.7	1
7	$\cos \theta$	n	76.4	9
8	$e - a$	n	80.5	3

The next step was the introduction of the attribute generator at the level of node creation in the decision tree (see Figure 6.6). Table 6.3 summarizes the results of 10 runs. The size of a decision tree is given as the number of nodes, and the errors are the misclassified cases. The number of new features actually used and their average complexity are shown. The complexity of a feature is the number of nodes in its tree representation (E.g., the complexity of the second attribute from Table 6.2 is 5). By new feature usage we mean the percentage of decision nodes in the tree where a new feature was selected.

So far learning was performed on the whole catalog. In order to reduce CPU time requirements, we introduce an additional training data selection step in our algorithm presented in Section 6.2 (between steps 3 and 4).

Since the number of positive examples is much less than the number of negative examples, and omitting some of them could result in losing important data that are not sampled elsewhere, we include all the positive examples in the training set. We select the relevant negative examples as the neighbors of positive examples in the mechanism space. The rest of the negative examples are put in the test data set. By this preselection we reduce the size of the training set to 983 elements and we also reduce the CPU time requirements by 85 % at the cost of worse solution accuracy. The best tree has 21 nodes, misclassifies 8 training examples, but classifies correctly all the test data. This structural description can be seen in Figure 6.12. A mechanism that satisfies one of the three constraint sets in Figure 6.12(b) is a feasible straight line generating mechanism.

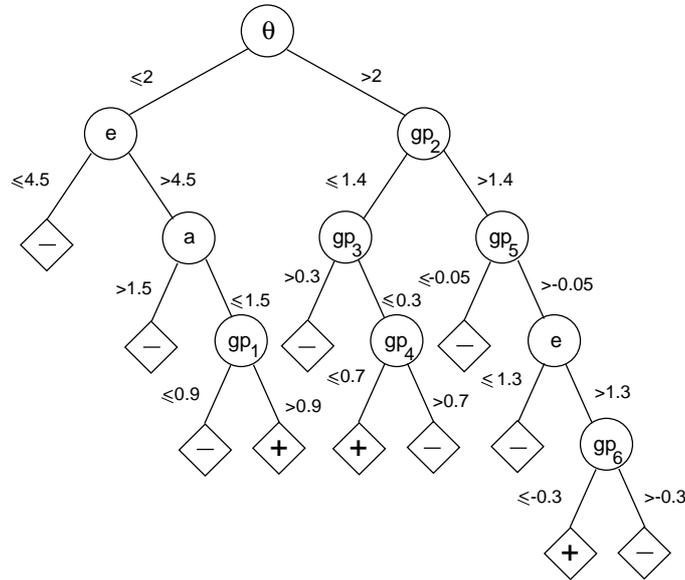
Table 6.4 summarizes the best results for the straight line generating mechanisms.

Table 6.3: The results of 10 runs of the program.

#	Decision tree		New features		
	size	errors	nr.	avg. usage	usage
				compl.	[%]
1	37	4	10	28	66.7
2	39	4	9	58	78.9
3	41	5	9	37	45
4	41	5	10	39	55
5	45	6	6	86	34.8
6	33	5	8	45	68.75
7	41	6	10	33	50
8	31	6	4	11	33.3
9	33	5	11	36	75
10	45	7	9	21	50
Avg.	39	5	9	37	55.7

Table 6.4: The best decision trees for the different methods.

Method	New features		Decision tree	
	nr.	usage[%]	size	errors
C4.5	0	0	47	11
Separate GP	8	60	41	6
GP at node level	10	66.7	37	4
Train data presel.	6	60	21	8



(a) The decision tree.

- 1: $\theta \leq 2$ and $e > 4.5$ and $a \leq 1.5$ and $gp_1 > 0.9$
- 2: $\theta > 2$ and $gp_2 \leq 1.4$ and $gp_3 \leq 0.3$ and $gp_4 \leq 0.7$
- 3: $\theta > 2$ and $gp_2 > 1.4$ and $gp_5 > -0.05$ and $e > 1.3$ and $gp_6 \leq -0.3$

(b) The structural description.

$$gp_1 = e \cos \theta + \cos \theta - 5.1$$

$$gp_2 = \frac{1.5+a}{2.8 \sin \theta + 0.53}$$

$$gp_3 = e \cos \theta - a + 2.9$$

$$gp_4 = \frac{c}{a(a + \sin \theta - ce)}$$

$$gp_5 = \frac{\sin \theta}{c} + \sin \theta - a - \frac{\sin \theta + c}{\sin \theta - c}$$

$$gp_6 = c - d + \frac{\sin \theta}{1.4}$$

(c) The GP attributes.

Figure 6.12: The description of the straight line generating mechanisms.

6.5 Discussion

Let us see again the example problem presented in Figure 6.9. The coupler curves of the corresponding mechanisms (that belong to the catalog) are presented in Figure 6.13. There are several different types of curves that all contain the straight line (within the given tolerance). As a matter of fact, the mechanisms that generate these curves differ from each other in many ways: the length of links, the position and orientation of the fixed link, the input angle corresponding to any selected point of the straight line. For example, let us consider two distinct curves and the corresponding mechanisms that generate these curves, as shown in Figures 6.14(a) and 6.14(b). The graphics are drawn at the same scale. We represented the mechanisms (together with their coupler curve) in two positions: the start (dashed lines) and the terminal position (solid lines) of the generated straight line. The input link is the emphasized link and it is rotated in the direction indicated by the arrow. For better distinction, instead of drawing the fixed link, we show just its joints by the black discs.

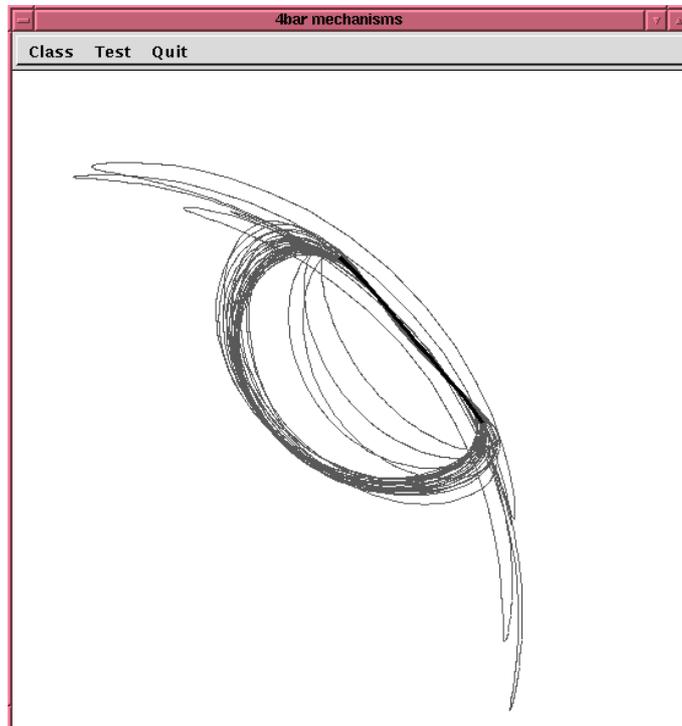
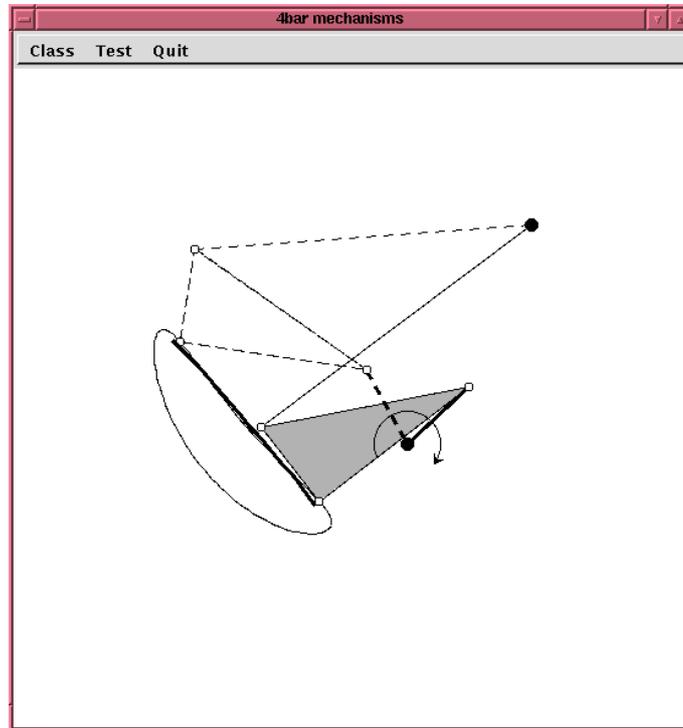
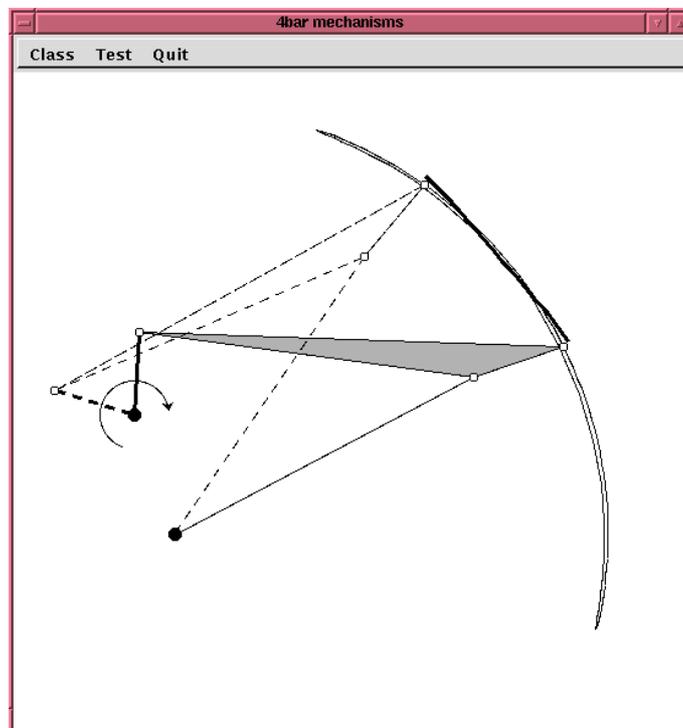


Figure 6.13: The coupler curves of the straight line generating mechanisms.



(a) $a = 3$, $b = 1$, $c = 2.5$, $d = 4$, $e = 2.24$, $\theta = 0.46$.



(b) $a = 1.5$, $b = 1$, $c = 4$, $d = 4$, $e = 5.02$, $\theta = 0.1$.

Figure 6.14: Straight line generating mechanisms.

This represents clear evidence for the fact that the common features of such different mechanisms (belonging to the same class) are not obvious, and, therefore, discovering these common features is not an easy task. Then again, the knowledge of such features (the structural description in our terminology) could be of much help for the engineer involved in mechanism design.

In many cases the mechanism is a part of a complex system and it has to fit in a predefined area. That is, the mechanism must remain within the predefined area during path generation. In addition, the position and orientation of the fixed link might also be constrained. The mechanism of Figure 6.14(a) could be preferred to the mechanism of Figure 6.14(b) from the point of view of fitting in a predefined area.

The choice of one or another mechanism from a class is left to the designer. The class is defined by the structural description and not the enumeration of the acceptable elements of the catalog. The catalog consists of points of the mechanism space and therefore, the found structural description applies not only to its elements, but to the mechanism space.

Chapter 7

Evolving Decision Principles for Multicriteria Decision Problems

In multicriteria decision problems many decisions must be made on subjective matters, such as the importance of the different criteria and the values of the alternatives with respect to subjective criteria. Since subjective decisions are not perfectly accurate, it is very important to analyze the sensitivity of results when small modifications of the subjective values are made. When solving a multicriteria decision problem, it is desirable to choose a decision principle that leads to a solution as stable as possible. We propose here a method based on genetic programming that produces better decision principles than the commonly used ones. The theoretical expectations are validated by two case studies.

The results presented in this chapter are based on our papers [19, 21].

7.1 Introduction

In most real life problems, if we want to make a decision, then generally we have to take into consideration more than one criterion. For example, if we want to buy a TV set, our criteria could be the cost, the quality of image, the physical aspect, etc. In general, the criteria have different importance. In the case of the TV set we would probably prefer the quality of the image to the physical aspect. If a possible alternative has values not worse in every criteria and better value in at least one criterion than another alternative (i.e., the alternatives are Pareto ordered), then it is easy to make a decision. We would definitely choose the first alternative. Unfortunately, in most of the cases the criteria are conflicting. In particular, a TV set with better image could have higher cost.

We use some decision principle for ranking the alternatives. A decision principle is a function of the values of an alternative with respect to the different criteria. The decision principle takes into consideration the value on each criterion with the corresponding importance.

In real life problems we must make many subjective decisions. The assignment of

importance to criteria is always a subjective decision. Subjective decisions must be also made when we assign values to alternatives with respect to a subjective criterion. For example, in the case of the TV set the quality of the image is a subjective criterion. It is difficult to assign an exact value to image quality. When slightly perturbing the subjective values (i.e., weights and values of alternatives with respect to subjective criteria, see the definitions in the next section) it is possible to obtain a different order of the alternatives. Since the assignment of subjective values is approximate, it would be preferable either to get the same order (i.e., to have a stable solution), or to obtain the same order with the smallest possible decrease of the perturbations (i.e., to have a solution with reasonable *stability*).

We shall introduce an index called the *stability index* which takes values between 0 and 1 and measures the stability of a solution. If the stability index is 1, then the solution is stable. The closer is the index to 1, the more stable is a solution. Our goal is to find a decision principle such that – for a fixed problem – the solution be as stable as possible. It is almost a hopeless task to search for a solution on the whole function space of possible decision principles. Therefore, we shall restrict the search to a subclass of these functions. We shall use genetic programming for finding decision principles as stable as possible for certain multicriteria decision problems.

7.2 The Multicriteria Decision Problem

First let us define a multicriteria decision problem.

In a *multicriteria decision problem* n alternatives A_1, \dots, A_n must be *ranked* by using m criteria C_1, \dots, C_m (generally) of different importance expressed by using the positive numbers w_1, \dots, w_m called the *weights* of criteria C_1, \dots, C_m , respectively. The values of the alternatives on each criterion can be organized in the following table:

$$\begin{array}{cc}
 & \begin{array}{ccc} x_1 & \cdot & \cdot & x_n \end{array} \\
 & \begin{array}{ccc} \mathbf{A}_1 & \cdot & \cdot & \mathbf{A}_n \end{array} \\
 \begin{array}{cc} w_1 & \mathbf{C}_1 \\ \cdot & \cdot \\ \cdot & \cdot \\ w_m & \mathbf{C}_m \end{array} & \left[\begin{array}{ccc} a_{11} & \cdot & \cdot & a_{1n} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ a_{m1} & \cdot & \cdot & a_{mn} \end{array} \right]
 \end{array}$$

In this table $a_{ij} > 0$ denotes the value of the j -th alternative in the i -th criterion and x_j denotes the aggregated value of the j -th alternative taking into account all the criteria. $\{x_1, \dots, x_n\}$ is called *solution* of the multicriteria decision problem. The alternatives are ranked by the values x_j , i.e., the larger is x_j , the better is A_j .

For calculating x_j we shall make the replacement

$$a_{ij} := 8 \frac{a_{ij} - \min \{a_{ik} : k = 1, \dots, n\}}{\max \{a_{ik} : k = 1, \dots, n\} - \min \{a_{ik} : k = 1, \dots, n\}} + 1,$$

if the larger values and

$$a_{ij} := 8 \frac{\max \{a_{ik} : k = 1, \dots, n\} - a_{ij}}{\max \{a_{ik} : k = 1, \dots, n\} - \min \{a_{ik} : k = 1, \dots, n\}} + 1$$

if the smaller values are preferred with respect to the i -th criterion, respectively. In this way the values of the alternatives with respect to different criteria are scaled in the same interval $[1, 9]$. If the larger values are preferred, the smallest value with respect to a criterion C_i takes the value 1 and the largest value with respect to C_i takes value 9. If the smaller values are preferred the situation is the opposite. All the other values of the alternatives with respect to C_i are disposed proportionally in the $[1, 9]$ interval. The scaling is important for comparing the values of the alternatives with respect to different criteria. With this replacement, the larger is a_{ij} , the better is the alternative A_j with respect to criterion C_i . The aggregated value x_j will be a function of the weights w_i and the values a_{ij} , where $i = 1, \dots, m$. The same function must be taken for every alternative. In order to have a feasible solution, this function should satisfy some natural conditions. This function will be called decision principle.

A function $f : \mathbf{R}_+^m \times \mathbf{R}_+^m \rightarrow \mathbf{R}_+$ is called a *decision principle* if the following axioms are satisfied:

- A1.** $f = f(v, z)$ satisfies the following homogeneity conditions: $f(\lambda v, z) = f(v, z)$ and $f(v, \lambda z) = \lambda f(v, z)$ for all $v, z \in \mathbf{R}_+^m$ and $\lambda \in \mathbf{R}_+$.
- A2.** $f(v_{\sigma(1)}, \dots, v_{\sigma(m)}; z_{\sigma(1)}, \dots, z_{\sigma(m)}) = f(v_1, \dots, v_m; z_1, \dots, z_m)$, for all $v_1, \dots, v_m, z_1, \dots, z_m \in \mathbf{R}_+$ and permutation σ of the set $\{1, \dots, m\}$.
- A3.** f is strictly increasing in each variable $z_i, i = 1, \dots, n$
- A4.** $\min_{i=1, \dots, m} z_i \leq f(v_1, \dots, v_m; z_1, \dots, z_m) \leq \max_{i=1, \dots, m} z_i$.

Hence, x_j will be calculated by the following formula:

$$x_j = f(w_1, \dots, w_m; a_{1j}, \dots, a_{mj}),$$

where f is a decision principle.

Generally, the α power mean

$$m_\alpha(w_1, \dots, w_m; y_1, \dots, y_m) = \left(\frac{\sum_{i=1}^m w_i y_i^\alpha}{\sum_{i=1}^m w_i} \right)^{1/\alpha}, \quad \alpha \geq 0$$

is used as decision principle. Most commonly, the arithmetic mean ($\alpha = 1$) is employed [53].

When subjective values are present in a multicriteria decision problem, it is very important to choose a decision principle that leads to a fairly stable solution. For a fixed decision table (i.e., fixed m , n , w_i and a_{ij}) *the stability of a solution depends only on the choice of the decision principle f .*

By the *sensitivity analysis* of a multicriteria decision problem we mean the analysis of the results, when we make small perturbations to the weights w_i and values a_{ij} of a decision table. We call a solution $\{x_1, \dots, x_n\}$ *stable* with respect to these perturbations if the decreasing order of the new solution $\{x'_1, \dots, x'_n\}$ coincide with the decreasing order of $\{x_1, \dots, x_n\}$.¹

For simplicity, let us suppose that all the criteria are subjective. A similar analysis can be made if we have both objective and subjective criteria: the values for the objective criteria are fixed, the sensitivity analysis for the changing subjective criteria can follow.

The decreasing order of the solution is not altered by the perturbations if the pairwise order of alternatives remains unchanged. The allowable perturbations may differ for the different pairs of consecutive alternatives. The stability of the order of two consecutive alternatives is a measure of the allowable perturbation.²

¹There have been developed several methods for sensitivity analysis, such as the method implemented in the *Expert Choice* software based on the *Analytic Hierarchy Process* multicriteria decision methodology [53, 54].

²The notion of stability of order was introduced in the multicriteria group decision software WINGDSS 4.1 [12, 13, 45] developed at the Laboratory of Operations Research and Decision Systems, Computer and Automation Research Institute.

For each value a_{ij} of an alternative let $a_{ij}^-(\varepsilon)$ and $a_{ij}^+(\varepsilon)$ denote the perturbed values for which the aggregated value of the alternative is minimal and, respectively, maximal, when perturbations of at most ε of the alternative value are permitted and the weights are constant.³ Then for each alternative j

$$x_j^-(\varepsilon) = f(w_1, \dots, w_m; a_{1j}^-(\varepsilon), \dots, a_{mj}^-(\varepsilon))$$

and

$$x_j^+(\varepsilon) = f(w_1, \dots, w_m; a_{1j}^+(\varepsilon), \dots, a_{mj}^+(\varepsilon))$$

are the minimum and, respectively, the maximum aggregated value for the allowed perturbations. Let us order the alternatives in decreasing order of the aggregated values x_j , given by the permutation σ such that $x_{\sigma(1)} \geq \dots \geq x_{\sigma(n)}$.

The *stability* of the order $x_{\sigma(j)} \geq x_{\sigma(j+1)}$ is the greatest $\delta_j \in [0, 1]$ for which $x_{\sigma(j)}^-(\delta_j \varepsilon) \geq x_{\sigma(j+1)}^+(\delta_j \varepsilon)$.

That is, if we allow perturbations of at most $\delta_j \varepsilon$ of the alternative values, the order of alternatives $\mathbf{A}_{\sigma(j)}$ and $\mathbf{A}_{\sigma(j+1)}$ remains unchanged. Hence, a stability value of 1 means that the ordering is stable with respect to perturbations of at most ε .

It is easy to see⁴ that for any decision principle the minimal (maximal) value $x_j^-(\varepsilon)$ ($x_j^+(\varepsilon)$) is attained for $a_{ij}^-(\varepsilon) = a_{ij} - \varepsilon a_{ij}$ ($a_{ij}^+(\varepsilon) = a_{ij} + \varepsilon a_{ij}$).⁵

With the help of the previously defined stability values we can measure the stability of a solution (decreasing order of the alternatives).

Consider δ_j the stability of order $x_{\sigma(j)} \geq x_{\sigma(j+1)}$, when using decision principle f .

We shall call *stability index* of decision principle f (for the fixed decision table) the value

$$S(\varepsilon) = \left(\prod_{j=1}^{n-1} \delta_j \right)^{\frac{1}{n-1}}.$$

³Further on, we plan to make a sensitivity analysis with respect to weights, too.

⁴The proof follows directly from the strict monotonicity of decision principles (axiom **A3**)

⁵If any value falls outside the [1,9] interval, it is rounded to the closest endpoint.

A stability index of 1 means that the solution is stable with respect to perturbations of at most ε , and a stability index of 0 means the solution is not stable at all. The closer is the stability index to 1, the more stable is the solution.

7.3 Generating the Decision Principle

Any decision principle is a function satisfying axioms **A1-A4** listed in the previous section. Thus, a straightforward method is the construction of new decision principles as combinations of elementary decision principles (α power means), so that the new functions satisfies these axioms.

First we studied the *arithmetic-geometric mean* of Gauss $M(a, b)$, defined as the common limit of the iteration

$$\begin{aligned} a_0 &= a > 0, & b_0 &= b > 0 \\ a_{n+1} &= \frac{a_n + b_n}{2}, & b_{n+1} &= \sqrt{a_n b_n}. \end{aligned}$$

We gave in [21] a set of interconnected noncontinuous recursions which are convergent to the same value. In order to accomplish this, we introduced the notions of prickly set, scalar and vectorial mean.

A set $C \subset \mathbb{R}_+^m$ is called *prickly* if for every $\mathbf{x} = (x^1, \dots, x^m) \in C$ the m -dimensional rectangle $\underbrace{\left[\min_{1 \leq i \leq m} x^i, \max_{1 \leq i \leq m} x^i \right] \times \dots \times \left[\min_{1 \leq i \leq m} x^i, \max_{1 \leq i \leq m} x^i \right]}_{m \text{ times}}$ is contained in C .

If $C \subset \mathbb{R}_+^m$ is a prickly set, a function $f : C \rightarrow \mathbb{R}_+$ is called a *scalar mean* if $\min_{1 \leq i \leq m} x^i \leq f(\mathbf{x}) \leq \max_{1 \leq i \leq m} x^i$ and the equality on the left hand side (right hand side) holds if and only if $x^1 = \dots = x^m$.

A map $F = (F_1, \dots, F_m) : C \rightarrow C$ is called a *vectorial mean* if its components F_1, \dots, F_m are scalar means and there exist $i_1, i_2 \in \{1, \dots, m\}$ such that for every $i \in \{1, \dots, m\}$ we have $F_{i_1} \leq F_i \leq F_{i_2}$.

We proved in [21] that if F is a vectorial mean, then the recursion $\mathbf{x}_{n+1} = F(\mathbf{x}_n)$ is always convergent. Moreover, F can be identified with a scalar mean. If F is continuous, the limit F^∞ of the recursion is a continuous function of the initial value $\mathbf{x} = \mathbf{x}_0$.

Many examples can be given by using the inequality between the power means.

We want to define stable decision principles, and in the case of this *infinite recursion*

it is very difficult, if not impossible, to analyze stability. Therefore, we restrict ourselves to the more simple *finite recursions*. Consequently, *we will search for stable decision principles that are finite recursive combinations of elementary decision principles (i.e., α power means)*.

7.4 Using the Genetic Programming Paradigm

The goal is to find the most stable decision principles corresponding to any given multicriteria decision problem. This is in fact a special function regression problem: we are looking for some function that (1) satisfies the axioms of a decision principle and (2) leads to a stable order of the alternatives for the given multicriteria decision problem.

Since we do not know in advance the exact structure of the function we are looking for, the genetic programming paradigm [3, 34] is especially well suited for this task:

- as an evolutionary method, it conducts a multidimensional search in the program space;
- the structure of individuals evolves together with their content;
- if the proper representation and evaluation methods are chosen, it is likely to discover solutions that are better than the manually created solutions [36].

Before starting the implementation of any genetic programming system, one should decide what representation to use and how to evaluate the genetic programs [34].

When choosing the *representation*, we have to take into consideration the fact that only those functions of the function space constitute feasible solutions, which satisfy the axioms of a decision principle. Generally, in the cases when only a fraction of the search space represent feasible solutions, one can select

1. a general representation – where additional correcting operators are needed for the transformation of any evolved non-feasible individual into the closest feasible one;
or
2. a representation encoding only feasible solutions – that excludes the possibility of evolving any non-feasible individual.

We decided for the second alternative. As outlined in Section 7.3, we construct valid decision principles starting from elementary decision principles, i.e., α power means ($\alpha \geq$

0, for $\alpha = 0$ we obtain in limit the weighted geometric mean). We use tree based genetic programming, where the rules for constructing a tree are the following:

- Any leaf of a tree is an α power mean ($\alpha \geq 0$) applied to an alternative's values (for the given multicriteria decision problem).
- Any internal node of a tree is an α power mean ($\alpha \geq 0$) with equal weights applied to its descendants.

All the individuals of our tree based genetic programming system are decision principles. On the other hand, by using this representation, we do not cover all possible decision principles. Even so we can obtain decision principles that are more general than the used elementary decision principles.

The *evaluation method* follows directly from the sensitivity analysis of multicriteria decision problems. For computing the fitness measure of the evolved decision principles we apply the following algorithm:

1. for each alternative \mathbf{A}_j , compute the aggregated value x_j , the minimum and the maximum aggregated value for the allowed perturbation $x_j^-(\varepsilon)$ and ,respectively, $x_j^+(\varepsilon)$;
2. order the alternatives in decreasing order of the aggregated values:

$$x_{\sigma(1)} \geq x_{\sigma(2)} \geq \dots \geq x_{\sigma(n)};$$
3. for each pair of neighboring alternatives $\mathbf{A}_{\sigma(j)}$ and $\mathbf{A}_{\sigma(j+1)}$ compute the stability value δ_j ;⁶
4. compute fitness as the stability index $Fitness = S(\varepsilon)$.

Having defined the representation and the fitness evaluation method, we can proceed and make experiments with our system.

7.5 Experimental Results

We conducted experiments on two multicriteria decision problems. In both cases we allowed variations of up to $\varepsilon = 10\%$ of the alternative values (a_{ij}). The experimental

⁶The stability value is found by binary search in the $[0, 1]$ interval with a predefined precision.

parameter setting including control parameters is summarized in Table 7.1. We present the results of 10 runs on each problem with each population size.

Table 7.1: The genetic programming parameter setting

Objective	Evolve the decision principle with greatest stability
Terminal set	Random reals $\alpha \in [0, 40]$, being evaluated as the α power mean of an alternative's values
Function set	α power means with equal weights ⁷ where $\alpha \in [0, 40]$
Fitness cases	The alternatives of the multicriteria decision problem
Fitness	The stability index computed for the given multicriteria decision problem
Population size	500, 1000, 2000
Crossover probability	90%
Probability of Point Mutation	10%
Selection method	Tournament selection, size 5
Termination criterion	None
Maximum number of generations	40
Maximum depth of tree after crossover	10
Initialization method	Grow

Example 1 Let us consider the problem of buying a TV set. We would like to get the best TV set of certain dimensions for a given price, but there are ten different offers. We base our decision on the following subjective criteria (in increasing order of importance):

C_1 *aesthetics* – The physical aspect influences us: we might like a box with rounded forms, with the speakers on the sides, or a brown, almost cubic box with the speakers at the bottom. We might prefer a colorful or artistically designed remote controller to a simple one.

C_2 *type of offered warranty* – It is a subjective decision whether a two years full war-

⁷We used equal weights for the simplicity of representation. As we will see in the case of the second example, different weights can evolve in the form of multiple occurrence of the same subtree (or leaf).

ranty, a one year full warranty followed by two years free service, or a one year full warranty followed by two years warranty on components is more preferable.

C_3 *confidence in trade-name* – The preference for the product of one company over the product of other company is also subjective.

C_4 *quality of image* – The perceived image (brightness, sharpness, colors, etc.) represents the most important criterion.

This is a multicriteria decision problem with four criteria and ten alternatives with the following decision table (after scaling):

		A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}
$w_1 = 0.1$	$C_1 = \text{aesthetics}$	1	2	9	3	1	4	8	9	5	6
$w_2 = 0.2$	$C_2 = \text{warranty}$	3	9	1	6	9	7	8	2	3	1
$w_3 = 0.3$	$C_3 = \text{trade-name}$	9	1	7	1	8	8	3	6	2	9
$w_4 = 0.4$	$C_4 = \text{image}$	5	6	2	9	1	2	4	3	7	4

In Table 7.2 we show several results produced by our genetic programming system. Each line shows the characteristics of one decision principle created by genetic programming in terms of (1) the stability index, (2) the size given as the number of nodes in the genetic tree, (3) the generation at which the function emerged and (4) the population size used for the run. With larger populations we obtained slightly more stable solutions. We made several runs with population size 4000, but the results were not better than for population size 2000. The best decision principle has stability index $S_{gp}(0.1) = 0.216$, meaning that the solution can be stable if the perturbations are reduced on average to 21.6 % of the initially allowed $\varepsilon = 0.1$ perturbation.

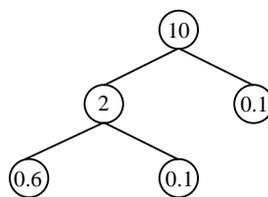


Figure 7.1: The most stable decision principle found by GP for Example 1.

The structure of the best decision principle we found is shown in Figure 7.1. Each internal node is evaluated to the corresponding (equal-weight) α power mean of its de-

Table 7.2: Some results given by GP for Example 1

#	Stability	Size	Gen.	Pop. Size
1	0.163	3	7	500
2	0.164	7	12	500
3	0.167	11	9	500
4	0.168	15	22	500
5	0.167	23	17	1000
6	0.167	19	8	1000
7	0.169	19	17	1000
8	0.166	19	19	2000
9	0.169	11	19	2000
10	0.216	5	12	2000

Table 7.3: Comparison between the solution given by GP and the solution given by the arithmetic mean for Example 1

best GP	order	A_1	A_7	A_6	A_{10}	A_4	A_9	A_8	A_2	A_5	A_3
	x_j	4.796	4.602	4.377	4.312	4.277	4.06	3.914	3.754	3.385	3.204
	δ_j	0.213	0.252	0.092	0.06	0.287	0.197	0.24	0.678	0.357	
arith- metic mean	order	A_1	A_4	A_{10}	A_6	A_7	A_2	A_5	A_9	A_8	A_3
	x_j	5.4	5.4	5.1	5	4.9	4.7	4.7	4.5	4.3	4
	δ_j	0	0.389	0.1	0.1	0.256	0	0.225	0.252	0.404	

scendants. Each leaf is evaluated to the corresponding α power mean of the alternative's values. Thus, the encoded decision principle is

$$f = \left(\frac{\left(\left(\frac{m_{0.6}^2 + m_{0.1}^2}{2} \right)^{\frac{1}{2}} \right)^{10} + m_{0.1}^{10}}{2} \right)^{\frac{1}{10}}.$$

The solution given by this decision principle is not very stable (stability would have been achieved for $S(\varepsilon) = 1$). At the same time, for the same problem the arithmetic mean leads to an unstable solution, $S_{arith}(0.1) = 0$. In Table 7.3 the two solutions are shown. The alternatives are listed in decreasing order of the aggregated values x_j . For each solution, we also show the stability value δ_j for each pair of consecutive alternatives. In the solution given by the arithmetic mean there are two pairs of alternatives whose ordering is totally unstable: A_1, A_4 and A_2, A_5 , respectively.

Thus, the evolved decision principle suggests buying TV set A_1 : not aesthetic, but best trade-name and above average image quality. The arithmetic mean could not differentiate between TV sets A_1 and A_4 . From practical considerations, nobody would buy a no-name TV set with seemingly better image quality (A_4): over the years the technical parameters of the TV set may drastically get worse, the unknown manufacturer may disappear, etc. Instead, the product of the preferred reliable company (A_1) would be chosen even if image quality is not so good at first glance.

Example 2 The second example is a multicriteria decision problem with three criteria and six alternatives:

$$\begin{array}{l} w_1 = 0.3 \quad C_1 \\ w_2 = 0.2 \quad C_2 \\ w_3 = 0.5 \quad C_3 \end{array} \begin{array}{c} A_1 \quad A_2 \quad A_3 \quad A_4 \quad A_5 \quad A_6 \\ \left[\begin{array}{cccccc} 1 & 8 & 9 & 9 & 5 & 9 \\ 7 & 9 & 1 & 6 & 9 & 7 \\ 9 & 1 & 7 & 1 & 8 & 8 \end{array} \right] \end{array}$$

The best decision principle produced by genetic programming (in generation 38 of a run with population size 2000) has stability index $S_{gp}(0.1) = 0.81$, meaning that the solution can be stable if the perturbations are reduced on average to 81 % of the initially allowed $\varepsilon = 0.1$. The evolved tree contains 21 nodes, its structure is shown in Figure 7.2. The multiple occurrence of the same leaf indicates the formation of different weights for the inner nodes of the tree. In the future we plan to introduce the different weights in the representation of inner nodes of trees. In this way we expect genetic programming to produce shorter (and more comprehensible) trees with similar performance in earlier generations.

For this problem, the stability index of the solution produced by the arithmetic mean is $S_{arith}(0.1) = 0.574$. Thus, the decision principle given by genetic programming leads to a considerably more stable solution. In Table 7.4 the two solutions are shown. The alternatives are listed in decreasing order of the aggregated values x_j . Although the order of the alternatives is the same in the two solutions, the stability values are higher for our solution. The evolved decision principle relies strongly on the geometric mean, but its stability is higher, namely $S_{gp}(0.1) = 0.81 > S_{geom}(0.1) = 0.768$.

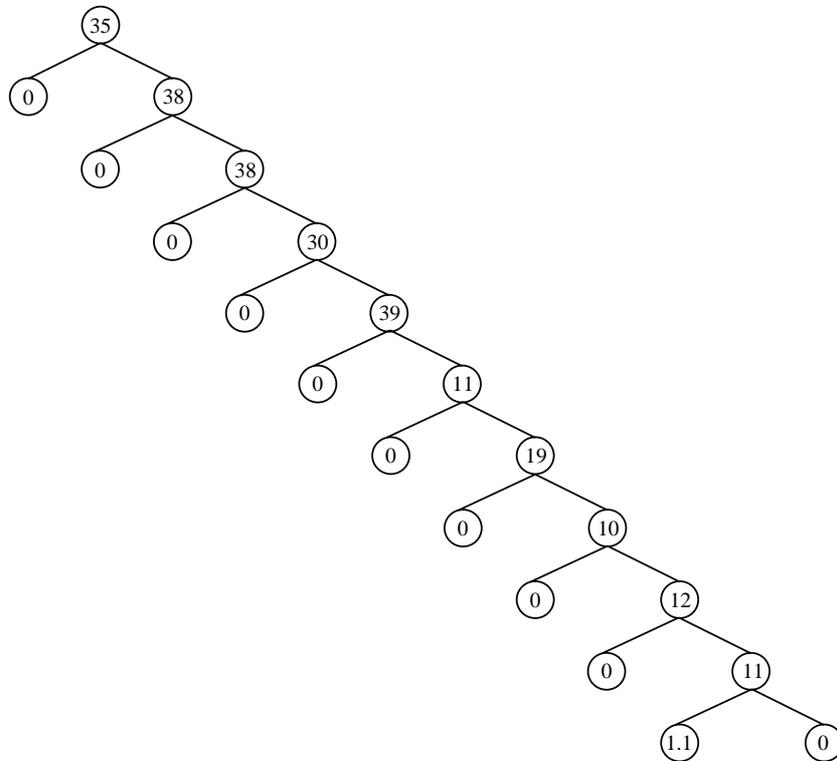


Figure 7.2: The most stable decision principle found by GP for Example 2.

Table 7.4: Comparison between the solution given by GP and the solution given by the arithmetic mean for Example 2

best GP	order	A_6	A_5	A_3	A_1	A_2	A_4
	x_j	8.069	7.114	5.125	4.519	3.33	3.123
	δ_j	0.697	1	1	1	0.498	
arithmetic mean	order	A_6	A_5	A_3	A_1	A_2	A_4
	x_j	8.1	7.3	6.4	6.2	4.7	4.4
	δ_j	0.588	0.818	0.252	1	0.51	

7.6 Discussion

We presented a novel approach to finding stable solutions for multicriteria decision problems. We used genetic programming for evolving decision principles that produce solutions more stable than those generated by commonly used decision principles. Two case studies confirmed that departing from traditional decision principles (i.e., α power means) even better decision principles can be constructed by genetic programming.

In genetic programming, there are several possibilities for the implementation of code reuse. Our next step will be the introduction of automatically defined functions [35]. We also plan to introduce different weights at the inner nodes of the genetic trees (i.e., weighted α power means in the function set). In this way we expect genetic programming to produce shorter (and more comprehensible) trees with similar performance in earlier generations.

In this work we supposed that the weights of the different criteria are constant. In the future we plan to make a sensitivity analysis with respect to weights.

Chapter 8

Conclusions

Genetic programming is a relatively new and rapidly developing field of evolutionary computation. In this work we have addressed several questions related to genetic programming and we have tried to provide answers.

The contributions of this work to the advance of genetic programming are two-fold: we have presented here both theoretical and application results. Our results can be grouped in four distinct categories, the first two categories summing up theoretical developments and the last two categories showing application results.

- I. Two new methods for *moderating code growth* in genetic programming have been introduced: the simplifying mutation method and the Pareto selection method.
 - (a) The first method has added a new simplification step to genetic programming systems. Simplification has been applied to probabilistically selected individuals in certain generations. Thus, the control of code growth has been independent of the fitness-based selection mechanism of genetic programming. It has been shown that code growth can be controlled without significant loss in performance.
 - (b) The second method has been based on *multiobjective optimization* for the objectives of *fitness* and *size*. We have used variants of the Pareto nondomination criterion in the selection step. We have shown that selection based on the Pareto nondomination criterion reduces code growth and processing time without significant loss of solution accuracy. Furthermore, the size of programs in the final population is independent of the maximum program size allowed in the initial population.
 - (c) The new methods have been compared with original genetic programming. Both new methods have reduced code growth and produce solutions of similar accuracy as those of original genetic programming. In the simplification method the mechanism for controlling code growth has been independent of

the fitness-based selection mechanism. The Pareto method has been very simple and it has not included program size in fitness computation.

II. A *distance function* reflecting the structural difference between genetic programs has been defined. The metric has allowed the application of fitness sharing between genetic programs and thus the control of the programs' diversity.

- (a) We have defined a new *distance metric* for genetic programs that can be efficiently computed. By using this metric, we have applied *fitness sharing* to genetic programming. The accuracy of solutions found by fitness sharing has been better than the accuracy of solutions found by original genetic programming.
- (b) We have applied fitness sharing for maintaining the *diversity* of the evolving population of genetic programs. For monitoring the diversity of the population throughout evolution, we have defined a new *diversity measure* for genetic programs. The new diversity measure has been based only on the structural differences of programs within the population, since the diversity of a population does not depend on the performance of its programs on a given data set.
- (c) By using tournament selection and fitness sharing, we have maintained the diversity of the population at a certain level depending on the size of program neighborhood.

III. Genetic programming has been combined with decision trees in a novel *hybrid machine learning method* for efficient learning of rules containing algebraic expressions. Decision trees have constituted the inductive learning engine and genetic programming has created new attributes.

- (a) This method has been successfully applied for finding meaningful descriptions of classes of four bar mechanisms. The data set has contained more than 7000 examples, from which less than 1% have belonged to the preferred class.
- (b) We have developed a preprocessing step for the best selection of the examples in the training set, and we have reduced the training set to 14% of the given

data set with little loss in accuracy. This preprocessing step can also be applied in other machine learning problems where the data set contains very few examples of one class and many examples of other class(es).

IV. The genetic programming paradigm has been applied for generating *stable decision principles* in multicriteria decision problems. The new decision principles have evolved as genetic programs and our best decision principle has been more stable than the classical ones.

- (a) We have defined new decision principles as combinations of elementary decision principles. The evaluation of the new decision principles has consisted in the sensitivity analysis of the multicriteria decision problem.
- (b) We have analyzed the situation when the values of alternatives with respect to subjective criteria are allowed to vary. By using genetic programming, we have obtained decision principles with much better stability when compared to the widely used decision principles.

In this work we have dealt with both theoretical and application aspects of genetic programming. In the future we plan to further extend our results related to code growth control to forms of representation other than trees (linear or graph-based). We also intend to continue the work on population diversity. We would like to devise other methods for controlling diversity and analyze their effects by using the presented diversity measure. Beyond the two presented real world problems, there are many other prospective problems that could be solved by genetic programming and where our findings could be applied.

Bibliography

- [1] Peter J. Angeline, Genetic programming and emergent intelligence, in *Advances in Genetic Programming*, ed., Kenneth E. Kinnear, 75–97, MIT Press, (1994).
- [2] Thomas Bäck, Ulrich Hammel, and Hans-Paul Schwefel, Evolutionary computation: comments on the history and current state, *IEEE Transactions on Evolutionary Computation*, **1**(1), (1997).
- [3] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone, *Genetic Programming: An Introduction*, Morgan Kaufmann, 1998.
- [4] Peter Bentley, An introduction to evolutionary design by computers, in *Evolutionary Design by Computers*, Morgan Kaufmann, (1999).
- [5] Eric Bloedorn and Ryszard S. Michalski, Data-driven constructive induction, *IEEE Intelligent Systems*, 30–37, (March/April 1998).
- [6] Avrim L. Blum and Pat Langley, Selection of relevant features and examples in machine learning, *Artificial Intelligence*, **97**, 245–271, (1997).
- [7] Uta Bohnebeck, Tamás Horváth, and Stefan Wrobel, Term comparisons in first-order similarity measures, in *Proceedings of the 8th International Workshop on Inductive Logic Programming*, ed., D. Page, volume 1446 of *LNAI*, pp. 65–79. Springer-Verlag, (1998).
- [8] Ashim Bose, Maria Gini, and Donald Riley, A case-based approach to planar linkage design, *Artificial Intelligence in Engineering*, **11**, 107–119, (1997).
- [9] Marco Ceccarelli and Adalberto Vinciguerra, Approximate four-bar circle-tracing mechanisms: classical and new synthesis, *Mechanism and Machine Theory*, **35**, 1579–1599, (2000).

-
- [10] Nicholas P. Chironis, *Mechanisms & Mechanical Devices Sourcebook*, McGraw-Hill, 1991.
- [11] Carlos A. Coello Coello, An updated survey of evolutionary multiobjective optimization techniques: state of the art and future trends, in *Proceedings of the Congress on Evolutionary Computation*, pp. 3–13, (1999).
- [12] Péter Csáki, Ferenc Fölsz, Tamás Rapcsák, and Zoltán Sági, On tender evaluations, *Journal of Decision Systems*, **7**, 179–194, (1998).
- [13] Péter Csáki, Tamás Rapcsák, Piroska Turcsányi, and Mátyás Vermes, R and D for group decision aid in Hungary by WINGDSS, a Microsoft Windows based group decision support system, *Decision Support Systems*, **14**, 205–217, (1995).
- [14] Anikó Ekárt, Generating class descriptions of four bar linkages, in *Late Breaking Papers at the Genetic Programming 1998 Conference*, ed., John R. Koza, pp. 42–47, (1998).
- [15] Anikó Ekárt, Controlling code growth in genetic programming by mutation, in *Late Breaking Papers of EUROGP'99*, eds., William B. Langdon, Riccardo Poli, Peter Nordin, and Terence Fogarty, pp. 3–12, (1999).
- [16] Anikó Ekárt, A general framework for obtaining useful design features of mechanisms, in *Annals of DAAAM for 1999, Proceedings of the 10th International DAAAM Symposium*, ed., B. Katalinic, pp. 149–150, (1999).
- [17] Anikó Ekárt, Shorter fitness preserving genetic programs, in *Artificial Evolution, 4th European Conference AE'99, Selected Papers*, eds., Cyril Fonlupt, Jin-Kao Hao, Evelyne Lutton, Edmund Ronald, and Marc Schoenauer, volume 1829 of *LNCS*, pp. 73–83. Springer-Verlag, (2000).
- [18] Anikó Ekárt and András Márkus, Decision trees and genetic programming in synthesis of four bar mechanisms, in *Life Cycle Approaches to Production Systems, Proceedings of the Advanced Summer Institute-ASI'99*, pp. 201–208, (1999).
- [19] Anikó Ekárt and S. Z. Németh, Evolving decision principles for multicriteria decision problems, Working paper, Laboratory of Operation Research and Decision Systems, Computer and Automation Research Institute, (2000).

-
- [20] Anikó Ekárt and S. Z. Németh, A metric for genetic programs and fitness sharing, in *Genetic Programming. Proceedings of EUROGP'2000*, eds., Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian Miller, Peter Nordin, and Terence Fogarty, volume 1802 of *LNCS*, pp. 259–270. Springer-Verlag, (2000).
- [21] Anikó Ekárt and S. Z. Németh, A noncontinuous generalization of the arithmetic-geometric mean, *Applied Mathematics and Computation*, (2001). In print.
- [22] Anikó Ekárt and S. Z. Németh, Selection based on the Pareto nondomination criterion for controlling code growth in genetic programming, *Genetic Programming and Evolvable Machines*, **2**, 61–73, (2001).
- [23] Walter Gilbert, Genes-in-pieces revisited, *Science*, **228**, 823–824, (1985).
- [24] J. R. Giles, *Introduction to the Analysis of Metric Spaces*, Australian Mathematical Society Lecture Series, 1987.
- [25] David E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.
- [26] D. A. Hoeltzel and Wei-Hua Chieng, Pattern matching synthesis as an approach to mechanism design, *ASME Journal of Mechanical Design*, **112**, 190–199, (1990).
- [27] John H. Holland, *Adaptation in Natural and Artificial Systems*, Ann Arbor: The University of Michigan Press, 1975.
- [28] Dale C. Hooper and Nicholas S. Flann, Improving the accuracy and robustness of genetic programming through expression simplification, in *Genetic Programming 1996: Proceedings of the First Annual Conference*, eds., John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, p. 428, (1996).
- [29] Jeffrey Horn, Nicholas Nafpliotis, and David E. Goldberg, A niched Pareto genetic algorithm for multiobjective optimization, in *Proceedings of the First IEEE Conference on Evolutionary Computation*, pp. 82–87, (1994).
- [30] John A. Hrones and George L. Nelson, *Analysis of the four bar linkage*, Wiley, 1951.

-
- [31] Hitoshi Iba, Hugo de Garis, and Taisuke Sato, Genetic programming using a minimum description length principle, in *Advances in Genetic Programming*, ed., Kenneth E. Kinnear, 265–284, MIT Press, (1994).
- [32] Forrest H. Bennett III, John R. Koza, Martin A. Keane, and David Andre, Darwinian programming and engineering design using genetic programming, in *Proceedings of the First International Workshop on Soft Computing Applied to Software Engineering*, eds., Conor Ryan and Jim Buckley, pp. 31–40, (1999).
- [33] Robert E. Keller and Wolfgang Banzhaf, Explicit maintenance of genetic diversity on genospaces, Technical report, Dortmund University, (1994).
- [34] John R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.
- [35] John R. Koza, *Genetic Programming: Automatic Discovery of Reusable Programs*, MIT Press, 1994.
- [36] John R. Koza, Human-competitive machine intelligence by means of genetic programming, *IEEE Intelligent Systems*, **15**(3), 76–78, (2000).
- [37] William B. Langdon, Size fair and homologous tree genetic programming crossovers, in *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, eds., W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, (1999).
- [38] William B. Langdon, Size fair and homologous tree crossovers for tree genetic programming, *Genetic Programming and Evolvable Machines*, **1**, 95–119, (2000).
- [39] William B. Langdon and Riccardo Poli, Fitness causes bloat, in *Soft Computing in Engineering Design and Manufacturing*, eds., P. K. Chawdhry, R. Roy, and R. K. Pant, 13–22, (1997).
- [40] Shin-Yee Lu, The tree-to-tree distance and its application to cluster analysis, *IEEE Transactions on PAMI*, **1**(2), 219–224, (1979).
- [41] Samir Mahfoud, Niching methods for genetic algorithms, Illigal report 95001, University of Illinois at Urbana-Champaign, (1995).

-
- [42] Jawaharlal Mariappan and Sundar Krishnamurty, A generalized exact gradient method for mechanism synthesis, *Mechanisms and Machine Theory*, **31**, 413–421, (1996).
- [43] Christopher J. Matheus and Larry Rendell, Constructive induction on decision trees, in *Proceedings of the IJCAI*, pp. 645–650, (1989).
- [44] Kurt Melhorn, Stefan Näher, and Christian Uhrig, *The LEDA user manual*, Max-Planck Institut für Informatik, Saarbrücken, 1997.
- [45] Csaba Mészáros and Tamás Rapcsák, On sensitivity analysis for a class of decision systems, *Decision Support Systems*, **16**(3), 231–240, (1996).
- [46] Shan-Hwei Nienhuys-Cheng, Distance between Herbrand interpretations: a measure for approximations to a target concept, in *Proceedings of the 7th International Workshop on Inductive Logic Programming*, eds., N. Lavrač and S. Džeroski, volume 1297 of *LNAI*, pp. 213–226. Springer-Verlag, (1997).
- [47] Peter Nordin, Frank D. Francone, and Wolfgang Banzhaf, Explicitly defined introns and destructive crossover in genetic programming, in *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, ed., Justinian P. Rosca, pp. 6–22, (1995).
- [48] Robert L. Norton, *Design of Machinery*, McGraw-Hill, 1992.
- [49] C. K. Oei, David E. Goldberg, and S. J. Chang, Tournament selection, niching and the preservation of diversity, Illigal report 91011, University of Illinois at Urbana-Champaign, (1991).
- [50] Giulia Pagallo, Learning DNF by decision trees, in *Proceedings of the IJCAI*, pp. 639–644, (1989).
- [51] J. Ross Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1993.
- [52] Justinian P. Rosca, Genetic programming exploratory power and the discovery of functions, in *Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pp. 719–736, (1995).

-
- [53] Thomas L. Saaty, *The analytic hierarchy process. Planning, priority setting, resource allocation*, McGraw-Hill, 1980.
- [54] Thomas L. Saaty and Luis G. Vargas, Uncertainty and rank order in the analytic hierarchy process, *European J. Oper. Res.*, **32**(1), 107–117, (1987).
- [55] George N. Sandor and Arthur G. Erdman, *Advanced mechanism design: Analysis and synthesis*, volume 2, Prentice Hall, 1984.
- [56] Joshikazu Sawaragi, Hirotaka Nakayama, and Tetsuo Tanino, *Theory of Multiobjective Optimization*, Academic Press, 1985.
- [57] J. David Schaffer, Multiple objective optimization with vector evaluated genetic algorithms, in *Genetic Algorithms and their Applications: Proceedings of the First International Conference on Genetic Algorithms*, pp. 93–100, (1985).
- [58] Stanley M. Selkow, The tree-to-tree editing problem, *Information Processing Letters*, **6**(6), 184–186, (1977).
- [59] Terence Soule, Code growth in genetic programming, Phd dissertation, University of Idaho, (1998).
- [60] Terence Soule, James A. Foster, and John Dickinson, Code growth in genetic programming, in *Genetic Programming 1996: Proceedings of the First Annual Conference*, eds., John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, pp. 215–223, (1996).
- [61] Lubert Stryer, *Biochemistry*, Freeman, 1995.
- [62] Kuo-Chung Tai, The tree-to-tree correction problem, *Journal of the ACM*, **26**(3), 422–433, (1979).
- [63] György Vancsay, *The application of genetic algorithms in mechanical design (in Hungarian)*, Ph.D. dissertation, Technical University of Budapest, 2000.
- [64] David A. Van Veldhuizen and Gary B. Lamont, Multiobjective evolutionary algorithms: analyzing the state-of-the-art, *Evolutionary Computation*, **8**(2), 125–147, (2000).

-
- [65] Kenneth J. Waldron and Gary L. Kinzel, *Kinematics, Dynamics and Design of Machinery*, Wiley, 1999.
- [66] J. Watson, N. H. Hopkins, J. W. Roberts, J. Argetsinger-Steitz, and A. M. Weiner, *Molecular Biology of the Gene*, Benjamin-Cummings, 1987.
- [67] Allan C. Wilson, The molecular basis of evolution, *Scientific American*, **253(4)**, 148–157, (1985).
- [68] Janusz Wnek and Ryszard S. Michalski, Hypotesis-driven constructive induction in AQ17-HCI: a method and experiments, *Machine Learning*, **14**, 139–168, (1994).
- [69] Xiaodong Yin and Noël Gerday, A fast genetic algorithm with sharing scheme using cluster analysis methods in multimodal function optimization, in *Artificial Neural Nets and Genetic Algorithms*, eds., Rudolf F. Albrecht, Nigel C. Steele, and Colin R. Reeves, pp. 450–457, (1993).
- [70] Byoung-Tak Zhang and Heinz Mühlenbein, Balancing accuracy and parsimony in genetic programming, *Evolutionary Computation*, **3(1)**, 17–38, (1995).
- [71] Kaizhong Zhang, Rick Statman, and Dennis Shasha, On the editing distance between unordered labeled trees, *Information Processing Letters*, **42**, 133–139, (1992).
- [72] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele, Comparison of multiobjective evolutionary algorithms: empirical results, *Evolutionary Computation*, **8(2)**, 173–195, (2000).