Reports

Machine Learning and Inference Laboratory

Handling Constrained Optimization Problems and Using Constructive Induction to Improve Representation Spaces in Learnable Evolution Model

Janusz Wojtusiak

MLI 07-3
P 07-6
November 28, 2007

George Mason University

Handling Constrained Optimization Problems and Using Constructive Induction to Improve Representation Spaces in Learnable Evolution Model

A dissertation submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy at George Mason University

By

Janusz Wojtusiak
Master of Science
Jagiellonian University, 2001

Co-Director: Ryszard S. Michalski
PRC Chaired Professor of Computational Sciences and Health Informatics
Co-Director: James E. Gentle
University Professor of Computational Statistics

Fall Semester 2007
George Mason University
Fairfax, VA

DEDICATION

To Professor Ryszard S. Michalski
1937-2007

ACKNOWLEDGEMENTS

I would like to thank Professor Ryszard S. Michalski, who made this dissertation possible. As an advisor and dissertation director, he guided me through the most difficult areas of this research, suggested possible solutions, stimulated thinking and exploration of new areas, and supervised my progress. In addition to sharing his knowledge and expertise, he engaged me in various projects on which I gained experience and learned how to pursue new scientific ideas. He was also always able to arrange financial support for me during my studies.

I would also like to thank my dissertation committee for their support, reviews and excellent comments that helped me to improve this dissertation. In particular, I would like to thank Professor James Gentle, who in the last semester took the responsibility of supervising my work on this dissertation, and was able to provide me with comments that significantly improved this work. Dr. Kenneth Kaufman, one of my closest collaborators at George Mason University and a committee member, helped me at all stages of my research, and carefully reviewed this dissertation. He also shared his knowledge and expertise on various occasions when we collaborated on many different projects. Professor Daniel Carr reviewed this dissertation and gave me many comments that helped me to improve it. Finally, Professor Hugo de Garis provided me with many important comments to the dissertation and in general to this research. We also collaborated on the application of the learnable evolution model to optimization of very large neural networks, which led to the development of methods that allow the application of the learnable evolution model to problems with very many attributes.

I would like to thank my close collaborators with whom I have the privilege to work during my Ph.D. studies. My closest collaborator, Jarek Pietrzykowski, provided me with an opportunity to discuss many issues concerning this research at its different stages. We also collaborated on many projects that are used in this dissertation, including the AQ21 system and the VINLEN system. I collaborated closely with Dr. Bartlomiej Sniezynski during and after his stay at George Mason University. His contributions to development of the AQ21 system helped me in performing the presented research. We also discussed many ideas on AQ learning and the learnable evolution model. My father, Professor Janusz Wojtusiak reviewed an earlier version of this dissertation and gave many useful comments. Jan Gehrke helped me to debug slides for the dissertation defense.

I would like to thank many people who directly or indirectly contributed to this dissertation. I collaborated with Dr. Anna Baranova on the analysis of the metabolic syndrome dataset used in Chapter 8. Dr. Naomi Gerber provided the vitality score dataset

and expertise also used in Chapter 8. Among those who indirectly helped me in this research are Professor Otthein Herzog and Doug Seeman. I also thank the faculty of the College of Science who taught different courses I had the privilege of taking, and my fellow students, with whom I worked on many class projects. Anonymous reviewers of several papers submitted during my Ph.D. studies often provided me with justified criticism of my research and helped to improve this research.

My research and Ph.D. studies would not have been possible without financial support from different sources. These include the National Science Foundation, under grants IIS 9906858 and IIS 0097476, the National Security Agency, under grant LUCITE #32, the George Mason University Provost's Office, and the University of Bremen, Germany. Part of this research was done during my stay at the Hanse Institute for Advanced Study, Germany.

TABLE OF CONTENTS

LIST OF FIGURES ix
LIST OF TABLES xi
ABSTRACT xii
CHAPTER 1 INTRODUCTION 1
CHAPTER 2 BACKGROUND AND DEFINITIONS 4
2.1 Basic Definitions 4
2.2 Evolutionary Computation 8
2.3 Attributional Calculus 11
2.3.1 Attributes and Their Types 11
2.3.2 Attributional Rules 14
2.3.3 Generalized Logic Diagrams 20
2.4 Concept Learning 22
2.4.1 Basic AQ Algorithm 24
2.4.2 Learning Strong Patterns 25
2.4.3 AQ21 Implementation of AQ Learning 27
CHAPTER 3 LEARNABLE EVOLUTION MODEL 31
3.1 The Basic Idea of the Learnable Evolution 31
3.2 Selection of Examples for Hypothesis Generation 33
3.3 Learning Hypothesis Describing High-Performing Candidate Solutions 34
3.4 Instantiation of Learned Hypotheses 37
3.4.1 Instantiation Algorithm 1 37
3.4.2 Instantiation Algorithm 2 40
3.4.3 Instantiation Algorithm 3 41
3.4.4 Instantiation of Alternative Hypotheses 41
3.5 LEM3 Implementation of the Learnable Evolution Model 43
3.5.1 LEM3 Algorithm 44
3.5.2 Innovation Operators in LEM3 45
3.5.3 Action Selection 46
3.6 Example Execution of LEM3 and EA 49
3.7 Other Systems Based on the LEM Methodology 59
3.8 Related Research on Non-Darwinian Evolutionary Computation 61
3.8.1 Estimation of Distribution Algorithms 61
3.8.2 Cultural Algorithms 62
3.8.3 Memetic Algorithms 62
3.8.4 Wise Breeding Genetic Algorithm 63
3.8.5 Other methods related to LEM 64

CHAPTER 4 HANDLING CONSTRAINTS 66
4.1 Introduction and Definitions 66
4.2 Summary of Methods of Handling Constrains 69
4.2.1 Penalty Functions 69
4.2.2 Constraint Preserving Operators 70
4.2.3 Rejection Methods 71
4.2.4 Representation Change 72
4.2.5 Repair Methods 73
4.2.6 Multi-objective Optimization Methods 74
4.3 Classification of Constraints 75
4.4 Instantiable Constraints 78
4.4.1 Instantiation Algorithm 81
4.4.2 An Example Execution of the Instantiation Algorithm 84
4.5 General Constraints 89
4.5.1 Trimming of Rules 93
4.5.2 Learning an Approximation of the Feasible Area 94
4.5.3 Using Infeasible Candidate Solutions as a Contrast Set for Learning 96
4.5.4 Discussion 97
4.6 Flexible Constraints 100
4.7 Starting with no Feasible Solutions 102
4.8 Conclusion 104
CHAPTER 5 REPRESENTATION SPACE 106
5.1 Two Examples Illustrating Modifications of Representation Space 106
5.2 Representation Space in the Learnable Evolution Model 111
5.3 Constructive Induction 113
5.4 Automated Improvement of Representation in the Learnable Evolution Model 119
5.4.1 Transformation Algorithm 120
5.4.2 Construction of Attributes 120
5.4.3 Discretization of Continuous Attributes 124
5.4.4 Selection of Attributes 126
5.4.5 Selection of Representation 127
5.5 Instantiation of Hypotheses Learned in Transformed Representations 129
5.5.1 Instantiation of Discretized Attributes 129
5.5.2 Rejection of Unsatisfied Conditions with Constructed Attributes 130
5.5.3 Instantiation of Conditions with Constructed Attributes 132
5.6 Controlling the Search for the Best Representation Space 133
5.7 Conclusion 133
CHAPTER 6 EXPERIMENTAL EVALUATION 135
6.1 Evaluating the Learnable Evolution Model on Non-Constrained Optimization 136
6.1.1 Optimization Problems 136
6.1.2 Evaluating Results 140
6.1.3 Results 142
6.2 Evaluating the Learnable Evolution Model on Constrained Optimization 145

6.2.1 Constrained Optimization Problems 145
6.2.2 Results of the Experimental Evaluation 147
6.3 Conclusions 150
CHAPTER 7 OPTIMIZATION OF PARAMETERS OF COMPLEX SYSTEMS WITH APPLICATIONS IN MEDICINE 152
7.1 Optimization of AQ21 Parameters on Selected Medical Datasets 152
7.1.1 Representation Space 153
7.1.2 Optimization Objective 157
7.1.3 Medical Datasets 159
7.1.4 Results 161
7.1.5 Conclusions 163
7.2 Application to Finding the Best Discretization of Numeric Attributes 163
7.2.1 Optimization Objective 164
7.2.2 Representation Space 165
7.2.3 Constraints 167
7.2.4 Results 169
7.2.5 Conclusions 171
CHAPTER 8 CONCLUSIONS 172
8.1 Contributions of the Dissertation 173
8.2 Future Work 174

LIST OF FIGURES

Figure 2-1: Illustration of Lexicographical Evaluation Functional. 6
Figure 2-2: A pseudocode of general schema of evolutionary computation programs. 8
Figure 2-3: Example hierarchy of attribute Vehicle. 12
Figure 2-4: Examples of attributional conditions. 15
Figure 2-5: Examples of attributional rules. 18
Figure 2-6: An example of Generalized Logic Diagram. 21
Figure 2-7: Example of concept learning..... 22
Figure 2-8: Decision tree learned from the input examples..... 23
Figure 2-9: Basic AQ algorithm. 24
Figure 2-10: AQ algorithm for learning strong patterns. 26
Figure 3-1: Pseudocode of a general LEM algorithm. 32
Figure 3-2: Generating new candidate solutions by machine learning. 33
Figure 3-3: A rule learned during optimization of the Rosenbrock function. 36
Figure 3-4: Instantiation algorithm 1. 38
Figure 3-5: Instantiation algorithm 2. 40
Figure 3-6: Instantiation algorithm 3. 41
Figure 3-7: Pseudocode of LEM3 Algorithm. 44
Figure 3-8: Flowchart of LEM3 Algorithm. 45
Figure 3-9: Pseudocode of no-progress condition. 48
Figure 3-10: A plot of the function f(x0, x1). 50
Figure 3-11: Randomly generated initial population (the same for both programs). 51
Figure 3-12: Randomly generated initial population (the same for both programs). 51
Figure 3-13: Learned hypothesis and H- and L-group candidate solutions in LEM3. 51
Figure 3-14: LEM3 Population in the second generation (100 fitness evaluations)..... 53
Figure 3-15: EA Population in the second generation (85 fitness evaluations)..... 53
Figure 3-16: Learned hypothesis and H- and L-group candidate solutions in LEM3. 53
Figure 3-17: LEM3 Population in the third generation (150 fitness evaluations). 54
Figure 3-18: EA Population in the third generation (130 fitness evaluations). 54
Figure 3-19: Learned hypothesis and H- and L-group candidate solutions in LEM3. 54
Figure 3-20: LEM3 population in the fourth generation (200 fitness evaluations). 56
Figure 3-21: EA Population in the fourth generation (168 fitness evaluations). 56
Figure 3-22: Learned hypothesis and H- and L-group candidate solutions in LEM3. 56
Figure 3-23: Two global optima found by LEM3 in the last, fifth generation (250 fitness evaluations)..... 57
Figure 3-24: EA population in the fifth generation (214 fitness evaluations). 57
Figure 3-25: One of the two optima found by EA in the eighth generation (346 fitness evaluations)..... 58

Figure 4-1: Pseudocode of rejection method for handling constraints. 71
Figure 4-2: Steps of LEM's learning mode with instantiable constraints. 79
Figure 4-3: Top level instantiation algorithm for constrained rules. 82
Figure 4-4: Backtracking-like method for resolving constraints of Types 2-3..... 84
Figure 4-5: Feasible and infeasible candidate solutions in the example problem. 92
Figure 4-6: Example rules discriminating high- and low-performing candidate solutions. 92
Figure 4-7: Trimmed rules for the example problem..... 93
Figure 4-8: Feasible space approximation. 95
Figure 4-9: Intersection of the learned hypothesis and the feasible space approximation where candidate solutions are created and tested against constraints..... 96
Figure 4-10: Hypothesis learned with set of infeasible solutions used as negative examples. 97
Figure 4-11: An example of a missed optimum for a constrained optimization problem. 98
Figure 4-12: Infeasibility of the best individuals in different generations when optimizing the G1 function. Each line represents average of ten executions for a given tolerance t. 104
Figure 5-1: Rules learned during optimization of the Rosenbrock function of 5 attributes..... 107
Figure 5-2: Rule learned during optimization of the Rosenbrock function of 5 attributes with discovered ridge. 108
Figure 5-3: Characteristic rule learned during optimization of the Rosenbrock function of 5 attributes with discovered ridge. 109
Figure 5-4: An illustration of function (5-1) with marked high- and low-performing candidate solutions..... 110
Figure 5-5: Rules leaned by LEM when optimizing the function (5-1). 110
Figure 5-6: A general schema of dependencies on representation in LEM. 112
Figure 5-7: General diagram of constructive induction in AQ learning. 116
Figure 5-8: Algorithm for constructing general form of attributes. 121
Figure 5-9: Algorithm for constructing instantiable attributes. 122
Figure 5-10: An illustration of improved simplicity when transforming the representation space. 128
Figure 5-11: Instantiation algorithm for rules with constructed attributes. 131
Figure 6-12: The Rastrigin function of 2 variables. 139
Figure 6-13: The Griewangk function of 2 variables..... 139
Figure 6-26: The Rosenbrock function of 2 variables. 139
Figure 6-15: The Sphere function of 2 variables. 139
Figure 6-16: The Step function of 2 variables. 139
Figure 6-17: Illustration of a δ-close solution..... 141
Figure 6-18: The average normalized numbers of infeasible candidate solutions generated during 100 generations of LEM3 execution on the G1, G12, G19, and G24 functions. 148
Figure 7-19: Increasing value of the fitness function when optimizing AQ21's parameters for the vitality score dataset..... 162

LIST OF TABLES

Table 5-1: The best fitness value and time of LEM3 execution with no constructive induction, and constructive induction with max parameter set to 5 and 10 respectively. 123
Table 6-2: List of parameters used in experimental evaluation. 143
Table 6-3: Average δ(v) values after 100 generations for different numbers of attributes for Griewangk, Rastrigin, Rosenbrock, and Sphere functions..... 144
Table 6-4: Average δ(v) values after 100 generations for the Griewangk, Rastrigin, Rosenbrock, Sphere, and Step functions with 2, 4, 10, 50 and 100 attributes..... 144
Table 6-5: Average δ(v) values after 100 generations for the Griewangk, Rastrigin, Rosenbrock, Sphere, and Step functions with 2 attributes..... 145
Table 6-6: Comparison of errors (E) for εDE, DMS-PSO + SQP, and LEM3 on G1, G12, G19, and G24 functions after 5000 fitness evaluations. For LEM3, the number of infeasible solutions (I) and for DMS-PSO + SQP the number of violated constraints (C), are also reported. 150
Table 7-7: Representation space for the problem of optimizing AQ21 parameters. 154
Table 7-8: Constraints in the problem of optimizing AQ21s parameters. 156
Table 7-9: Constraints in the problem of finding optimal discretization. 168
Table 7-10: Results of finding the best discretization by LEM3 and ChiMerge on the metabolic syndrome dataset. 170
Table 7-11: Results of finding the best discretization by LEM3 and ChiMerge on the vitality score dataset. 170

ABSTRACT

HANDLING CONSTRAINED OPTIMIZATION PROBLEMS AND USING CONSTRUCTIVE INDUCTION TO IMPROVE REPRESENTATION SPACES IN LEARNABLE EVOLUTION MODEL

Janusz Wojtusiak, Ph.D.

George Mason University, 2007

Dissertation co-Director: Dr. Ryszard S. Michalski
Dissertation co-Director: Dr. James E. Gentle

This dissertation investigates two closely related problems in the learnable evolution model: the automatic improvement of the representation space using constructive induction, and the handling of constraints in optimization tasks. The former includes an investigation of the theoretical and implementational aspects of representation space transformations in the context of complex optimization problems, the development of algorithms that perform these transformations, and algorithms for creating new candidate solutions (via instantiation) in the improved representation spaces. Handling specific types of constraints is closely related to the equation instantiation task in the modified representation spaces; therefore, the same methods can be used for solving both problems. Moreover, transformations of representation spaces may help in handling constraints of other types, that is, constraints that cannot be handled directly during the instantiation process.

The developed algorithms are implemented in the LEM3 and AQ21 systems and tested on a set of constrained and non-constrained benchmark optimization problems. Two exemplary applications to optimization of complex systems in the context of selected medical datasets are also presented. These applications are the optimization of AQ21 parameters, and automatic discretization of numeric attributes.

CHAPTER 1 INTRODUCTION

This research investigates and extends a novel optimization method, called the *learnable evolution model* (LEM), that applies advanced machine learning to search very complex problem solution spaces. By applying machine learning, LEM hypothesizes why some candidate solutions (e.g. designs, complex parameter settings) perform better than others, and uses that knowledge to create new candidate solutions (e.g., Michalski, 1998; 2000). In particular, this research adds solutions of two interrelated problems to the body of work in that area. One is how to handle complex constraints in LEM, and the second is how to automatically improve the representation space in which solutions are sought in order to improve and speed-up the optimization process.

Most real world optimization problems have constraints which that limit the space of feasible solutions. Such constraints are used to prohibit search space solutions that are physically impossible, make no sense based on expert's knowledge, or imply undesirable properties of the optimized systems. Because many methods of constrained optimization have been developed in the past, this research concentrates on novel methods that specifically apply to LEM. The main observation that led to the development of methods for handling constraints in LEM is that different types of constraints require different methods. A general distinction is made between *instantiable* constraints that can be

handled directly in LEM's instantiation process and *general* constraints for which there is no simple algorithmic solution. Different methods for handling both types of constraints are presented in Chapter 4 and experimentally evaluated in Chapter 6.

The original representation of candidate solutions may not be adequate for the optimization problem, thus its improvement may lead to finding better solutions or finding solutions more efficiently. The process of designing representation spaces is often complicated and requires substantial domain knowledge. Moreover, different representation may be needed at different stages of the optimization process. Because of that, there is a need for automated methods that improve the representation of solutions. This research adopts selected *constructive induction* (CI) methods used in machine learning for improving representation spaces and applies them in the learnable evolution model. The key issue concerns creating new candidate solutions from hypotheses learned in modified representation spaces. The solution proposed in Chapter 5 of this dissertation uses methods for handling constraints to resolve this issue. Experimental evaluation presented in Chapter 6 confirms that for selected types of problems, constructive induction consistently improves the program's performance, as measured by the fitness of the best achieved solution after a given number of the fitness evaluations.

This dissertation is organized in the following way: Chapter 2 presents definitions and background knowledge from the fields of machine learning and evolutionary computation needed for further discussion. Chapter 3 presents in detail the learnable evolution model and its relevant features. It also introduces LEM3, the newest LEM implementation,

which is the basis for development of the research described here. Chapter 4 presents the problem of constrained optimization, proposes a classification of constraints, and presents methods of handling constrained optimization problems in LEM. Chapter 5 discusses issues of representation (optimization) space in LEM and methods of its automated improvement. Chapter 6 presents experimental results of the application of LEM to several well known test problems, and analysis of its performance. Chapter 7 presents an application of LEM to optimization of complex systems. In particular, it is applied to optimizing parameters of the AQ21 machine learning program for selected medical datasets, and finding the best discretizations of numeric attributes in medical datasets. Finally, Chapter 8 presents conclusions and discusses future research directions in LEM. In addition, each chapter discusses related research.

CHAPTER 2 BACKGROUND AND DEFINITIONS

This chapter presents definitions of basic concepts and background information needed for further discussion of the learnable evolution model, methods of handling constraints and automated improvement of representation spaces.

2.1 Basic Definitions

The problem *representation space,* also known as *search space,* is the set of all possible problem solutions. In this work it is the Cartesian product of the domains of attributes used to define candidate solutions or hypotheses. Formally, $E=D_1 \times D_2 \times \ldots \times D_n$, where E is the representation space and $D_1, \ldots, D_n$ are domains of the attributes used to represent possible solutions and hypotheses. The presented method distinguishes between the *original representation space* provided to the system in which the fitness function is defined and a *modified representation space* in which innovation operators are executed. The original representation space is denoted E and the modified representation spaces are denoted E_C.

The *fitness function*, also known as the *objective function*, defines a criterion for evaluating candidate solutions. The goal of the optimization process is to find the function's optima (minima or maxima). Formally, the fitness function f is given by (2-1).

$$f: E \rightarrow \mathcal{R} \qquad (2\text{-}1)$$

E is the representation space (search space) and $\mathcal{R}$ is the set of real numbers.

Candidate solutions are members of the set of possible solutions. Each candidate solution is represented by a single point in the representation space. In evolutionary computation, candidate solutions are commonly referred to as *individuals*.

Concept learning is a specific type of *supervised learning* that concerns methods of learning from labeled data. Given a set of examples (e_i, C_i) belonging to classes $C_1..C_n$, where e_i are elements of E, the goal of supervised learning is to induce general descriptions of classes C_1 .. C_n based on the provided examples. In the case of concept learning, there are usually only examples of one class (the concept) and non-examples of the concept. In the presented study, learned hypotheses are represented in form of rules in *attributional calculus* ($\mathcal{AC}$), and the AQ method is used for learning (see Section 2.4).

Constructive induction is a process of improving the representation space, E, by modifying domains of attributes, removing irrelevant attributes, and/or creating new attributes. The original representation space E is transformed into a modified representation space E_C which is more suitable for the process of learning or optimization. Chapter 5 describes in detail different methods of constructive induction,

especially those applicable to improvement of representation spaces in the learnable evolution model, and other representation-related issues.

Lexicographical evaluation functional (LEF) is a method of evaluating solutions using multiple criteria (e.g., Michalski, 1972). The concept of LEF is used in all chapters, so its understanding is important. Objects such as rules or candidate solutions go through a series of tests on predefined criteria, each with a given tolerance (2-2). For each criterion, in sequence, LEF selects for consideration remaining solutions whose evaluation is within a given tolerance from the best solution.

$$<(\text{Criterion}_1, \tau_1); (\text{Criterion}_2, \tau_2); \ldots (\text{Criterion}_n, \tau_n)> \quad (2\text{-}2)$$

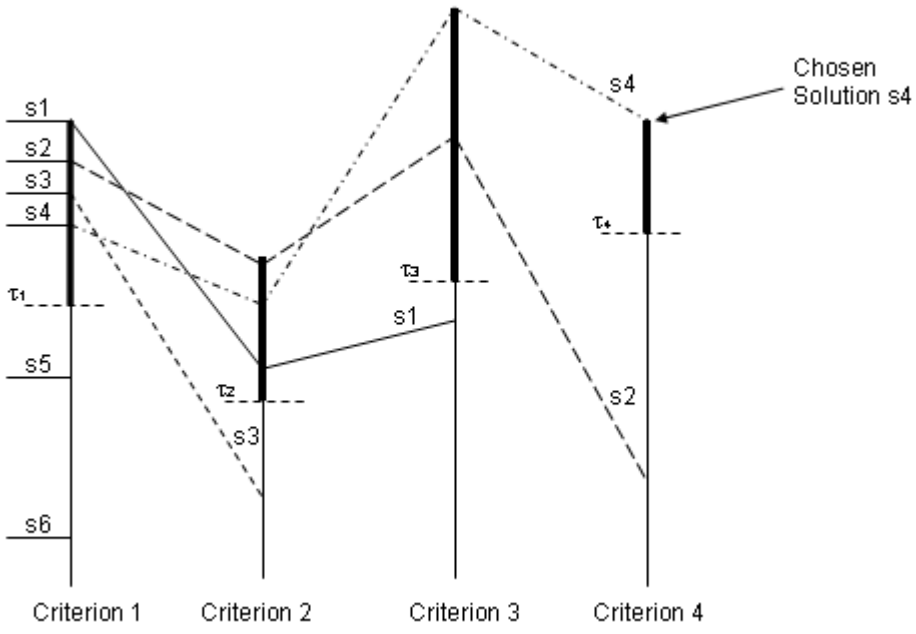


Figure 2-1: Illustration of Lexicographical Evaluation Functional.

The usage of LEF is illustrated in Figure 2-1, in which six solutions (s1, …,s6) are being evaluated by a LEF with four criteria. The solution s1 scores the best for the first criterion and the solutions s2, s3, and s4 are within the given tolerance τ1 of s1, thus s1, s2, s3, and s4 are all considered acceptable according to the first criterion. The solutions s5 and s6 are outside the tolerance and are rejected. Among the solutions that passed through the first criterion, s2 scores the best on the second criterion, and s1 and s4 are still within the given tolerance for the second criterion. The solution s3 is rejected. On the third criterion the solution s4 scores the best, with s2 within its tolerance. Finally, on the fourth criterion, the solution s4 scores the best, and is selected according to the LEF. Please note that the solution s4 is not the best based on the first and second criteria, but because it is close enough to the best solution (within given tolerances), it is considered equivalent. Finally, when all the other solutions are rejected the solution s4 is selected by the LEF.

Predictive accuracy of a hypothesis on a testing dataset is defined as a ratio of the number of correctly classified examples by the hypothesis to the total number of examples in the testing dataset. It can be expressed either as a number in the range [0, 1] or as a percentage.

Precision of a hypothesis H on a testing dataset Ts is given by (2-3). A testing program may assign a testing example to more than one class, thus its answer is imprecise.

$$precision(H, Ts) = \frac{|Ts| * |classes| - |assignments|}{|assignments| * (|classes| - 1)} \quad (2\text{-}3)$$

Here |classes| denotes the number of classes, |assignments| denotes the number of assignments of training examples to classes, and |Ts| denotes the total number of testing examples.

2.2 Evolutionary Computation

Evolutionary computation (EC) is a class of stochastic optimization methods initially inspired by Darwin's theory of evolution (Darwin, 1859). Its terminology is also borrowed from biology; terms such as population, individual, and selection of the fittest are commonly used. A typical goal of evolutionary computation, also used in this dissertation, is optimization, that is, finding the best solution in the representation space according to a specified criterion (fitness). The general schema of evolutionary computation is presented in pseudocode in Figure 2-2.

```
Create an initial population of candidate solutions
Evaluate candidate solutions in the initial population
Loop while stopping criteria are not satisfied
      Create new candidate solutions
      Evaluate fitness of the new candidate solutions
      Select a new population
```

Figure 2-2: A pseudocode of general schema of evolutionary computation programs.

These methods start with an initial population, which is either generated randomly or is provided to the system (for example, designs/individuals known to have high performance). Depending on the application and the particular EC method used, several different methods for generating initial populations have been proposed and discussed in the literature. In most cases, researchers agree that initial populations should be

distributed uniformly in the search space if no prior knowledge about possible solutions' locations is available.

Evaluation of candidate solutions/designs involves computing their *fitness* values. Evolutionary computation methods require fitness values that can be compared. Given two or more candidate solutions, a program is able to compute their fitnesses and use the result to decide which perform better and which perform worse. The evaluation of solutions may be a very time consuming and costly operation. It may involve running simulators, running various experiments, and sometimes may even require interaction with human experts. The time needed to evaluate fitness of a single candidate solution varies from a small fraction of a second (e.g. in function optimization problems) to hours of supercomputer CPU time for problems that require advanced simulation. Examples of applications that require running simulators to evaluate candidate solutions are designing heat exchangers (Kaufman and Michalski, 2000a; Domanski et al. 2004), optimal non-linear filters (Coletti et al., 1999), aircraft wing shapes (Oyama, 2000), and various other applications (see Bentley and Corne, 2002; Gen and Cheng, 2000; Rothlauf et al., 2006; Giacobini et al., 2007).

Stopping criteria depend on implementation of an evolutionary computation method, its parameters, and a general optimization task. Example stopping criteria include finding a solution whose fitness reached a given value (a satisfactory solution is found), exceeding maximum computational resources, such as time of evolution or maximum number of fitness evaluations; and attaining insufficient progress for a given number of generations.

Creation of new candidate solutions, the so called *innovation* process, is the most important step of evolutionary computation. Methods for creating new candidate solutions vary from purely random generation of points in the search space, through semi-random operators such as different types of mutations and recombinations, to advanced methods that involve reasoning or machine learning and are guided by an "intelligent agent." The advanced methods are realized in the form of recently introduced *non-Darwinian evolutionary computation* methods such as cultural algorithms (CAs), estimation of distribution algorithms (EDAs), the learnable evolution model (LEM), and memetic algorithms (MAs) that are further described in Chapter 3.

Selection of new populations initially followed the Darwinian rule of the selection of the fittest. Although it is generally the case in all selection methods that candidate solutions with better fitness have higher chances of survival, recent methods are probabilistic and/or include advanced reasoning. For example, one method is to select a group of lower performing candidate solutions in the population in order to maintain diversity and allow for better search of the optimization space.

Classifications of evolutionary computation methods take into consideration their different aspects, such as representation of candidate solutions, operators used, and applications (Back, Fogel and Michalewicz, 2000). The main subfields of evolutionary computation are *genetic algorithms* (Holland, 1962; 1975) and *evolution strategies* developed independently (Rechenberg, 1965; Schwefel, 1965), *evolutionary programming* (Fogel, Owens, and Walsh, 1966; Fogel, 1999), *genetic programming*

(Koza, 1992), and several new approaches such as *estimation of distribution algorithms* (Mühlenbein and Paaß, 1996; Larrañaga and Lozano, 2002), *memetic algorithms* (Moscato, 1989; Hart, Krasnogor and Smith, 1994), and *differential evolution* (Storn and Price, 1997; Price, Storn and Lampien, 2005).

2.3 Attributional Calculus

Attributional calculus ($\mathcal{AC}$) is a logic system, introduced by Michalski (2004a), which combines elements of first order predicate logic, propositional logic, and multi valued logic. Its purpose is to provide a formal representation language for *natural induction*, an inductive learning process whose goal is to hypothesize knowledge in human oriented forms, using easy to interpret rules, graphical representations etc. The following sections present selected aspects of attributional calculus that are important for the learnable evolution model and methods discussed in this dissertation.

2.3.1 Attributes and Their Types

Objects being considered in learning or optimization are described in terms of their properties formally represented by attributes. Attributional calculus recognizes several types of attributes that correspond to ways in which humans represent object's properties. These attribute types are employed in the learnable evolution model, in particular in its LEM3 implementation, and in AQ learning used for hypothesis formulation in LEM. An attribute is a function that for each entity in the space of possible entities assigns a value from the attribute domain. In this research attributes are denoted using their names with

capitalized first letters, e.g., Weather or Color, and symbolic values of their domains are usually denoted using all small letters, e.g., windy or red. Attributional calculus recognizes the following attribute types:

nominal – whose domain is an unordered, finite set of values, for example, an attribute Color with domain {red, green, blue, yellow}

structured – whose domain is a partially ordered (with a given hierarchy), finite set of values, for example, an attribute Vehicle with domain {car, suv, sports car, van, bicycle, mountain bicycle, cross country, downhill, sport bicycle, train} where the hierarchy is shown in Figure 2-3.

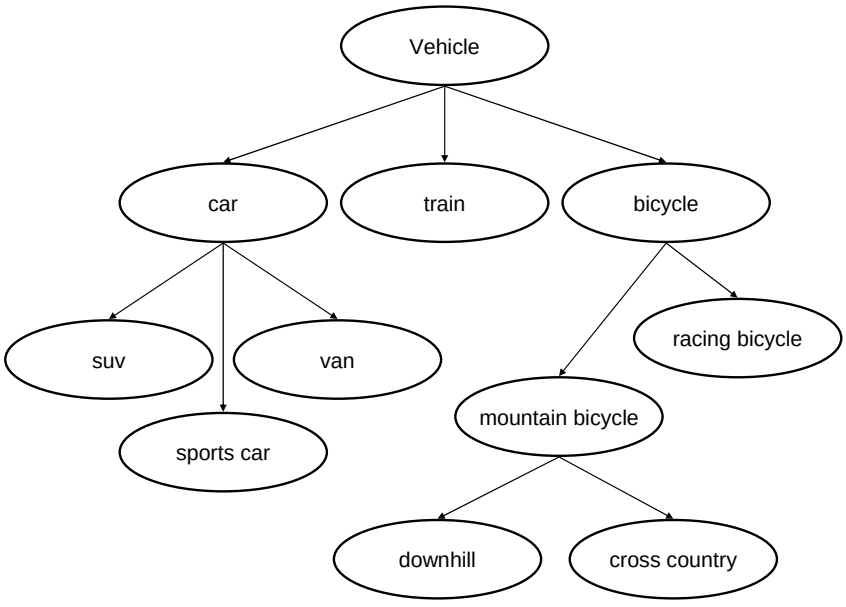


Figure 2-3: Example hierarchy of attribute Vehicle.

ordinal – whose domain is an ordered set of values, for example an attribute "grade" whose domain is {A, B, C, D, E} where "A" is the highest and "E" is the lowest grade.

interval – whose domain is an ordered set of values. The transformations $y' = a + y$ apply to the interval attributes. For example, Temperature (in degrees Fahrenheit) is an interval attribute – it makes sense to say "temperature outside is 20 degrees lower than inside," but it is not correct to say that "temperature outside is twice as low as inside," because zero is not well defined for the attribute (unless it is measured in the Kelvin scale).

ratio – whose domain is an ordered set of values, and ratio transformations $y' = ay$ apply to the attributes. For example, Length is a ratio attribute, because one may say "that table is twice as long as the sofa."

absolute – whose domain is an ordered set of values, but no transformations apply to the attributes. For example, the Social security number (SSN) is an absolute attribute.

set-valued – whose values are sets, and its domain is a power set (a se of all subsets) of a base set. For example, Items in a customer's cart is a set-valued attribute.

compound – whose domain is a Cartesian product of domains of its constituent attributes. For example, an attribute Weather can be built of constituent attributes such as Wind, Rain, Temperature, etc.

The ordinal, interval, ratio, and absolute attribute types are commonly referred to as *linear*, because their values are linearly ordered. A detailed description of attribute types and their use in AQ learning are presented by Michalski and Wojtusiak (2007).

2.3.2 Attributional Rules

Natural induction requires that knowledge is learned in forms easy to understand and interpret by people who may not be experts in machine learning, knowledge mining, or have a technical background. Thus, medical doctors, engineers, economists, security officers, or architects should be able to understand, interpret, modify and apply knowledge learned by computer systems. Such a goal requires that knowledge discovery programs use a rich language that can be either automatically translated to natural language (e.g. English) or easy to understand itself. Learned knowledge is represented in attributional calculus in the form of *attributional rules* which consist of *attributional conditions*. An attributional condition takes the form:

$$[L \text{ rel } R: A], \tag{2-4}$$

where L is an attribute, an internal conjunction or disjunction of attributes, a compound attribute, or an expression; rel is one of =, >, <, ≤, ≥, :, or ≠; R is an attribute value, an internal disjunction of attribute values, an attribute, an internal conjunction of values of attributes that are constituents of a compound attribute, or an expression, and A is an optional annotation that may list |p| and |n| values for the condition, defined as the numbers of positive and negative examples, respectively, that satisfy the condition, and the condition's consistency defined as $|p|/(|p|+|n|)$. If the condition is presented in conjunction with other conditions, the annotation may also consist of $|p_c|$ and $|n_c|$ which are cumulative numbers of positive and negative examples, respectively (numbers of positive and negative examples that satisfy the condition and all previous conditions in the conjunction), and cumulative consistency $|p_c|/(|p_c|+|n_c|)$. Figure 2-4 presents examples of attributional conditions and their explanations.

Condition	Explanation
[Length>7.3]	The length of an entity is greater than 7.3 units (as defined in the attribute's domain).
[Color=red ∨ blue: 40,2]	The color of an entity is red or blue. The condition is satisfied by forty positive and two negative examples.
[Width=short..medium]	The width of an entity is between short and medium (inclusive).
[Head color = Body color]	The color of the head and the color of the body are the same.
[Length & Height≤12]	An entity's length and height are both smaller or equal to 12 units. The units are defined in the attributes' domains.
[Weather: sunny & windy]	The weather is sunny and windy. This is an example of a condition that includes a compound attribute Weather.
[Length + Width≤24 inches]	The sum of Length and Width is smaller or equal to 24 inches.

Figure 2-4: Examples of attributional conditions.

There are several different forms of attributional rules allowed by attributional calculus. Three important forms of attributional rules are presented below (2-5) - (2-7).

$$\text{CONSEQUENT} \Leftarrow \text{PREMISE} \tag{2-5}$$

$$\text{CONSEQUENT} \Leftarrow \text{PREMISE} \llcorner \text{EXCEPTION} \tag{2-6}$$

$$\text{CONSEQUENT} \Leftarrow \text{PREMISE} \lceil \text{PRECONDITION} \tag{2-7}$$

where PREMISE, CONSEQUENT, EXCEPTION, and PRECONDITION are complexes, that is, conjunctions of attributional conditions. An EXCEPTION can also be an explicit list of examples that constitute exceptions to the rule. The rules (2-5) are interpreted that the CONSEQUENT is true whenever the PREMISE is true. The rules (2-6) are interpreted that the CONSEQUENT is true whenever the PREMISE is true, except for when the EXCEPTION is true. The rules (2-7) are interpreted that the CONSEQUENT is true whenever the PREMISE is true, provided that the PRECONDITION is true. The signs $\llcorner$ and $\lceil$ are used to denote exception and precondition, respectively. Each rule may be optionally annotated with several parameters such as numbers of covered examples (positive and negative), the rule complexity etc. Examples of attributional rules are presented in Figure 2-5.

Rule	Explanation
[Part=acceptable] ⇐ [Width=7..12] & [Length<3] & [Material=steel ∨ plastic]	A part is acceptable if its width is between 7 and 12, its length is less than 3 and its material is steel or plastic.
[Activity=play ∨ hike] & [Dress=casual] ⇐ [Weather: sunny & high temperature] & [Day of week=weekend] & [Homework=completed]	The activity will be to play or to hike and dress will be casual if weather is sunny and high-temperature and day of week is weekend and homework is completed. This is a *multi-head rule* with two conditions in its consequent. The premise consists of three conditions. The attribute Weather is compound, the attribute Day of week is structured (here Saturday and Sunday are generalized into a higher level concept of weekend), and Homework is a nominal attribute.
[Design=high-performing] ⇐ [X1=1..3: 20,10] & [X5=1.5: 7, 3] : p=5, n=1	A design is high-performing if X1 takes value between 1 and 3, and X5 takes value 1.5. The first condition is satisfied by 20 positive and 10 negative training examples and the

	second condition is satisfied by 7 positive and 3 negative raining examples. The entire rule is satisfied by 5 positive and 1 negative training examples.
[Activity=play] ⇐ [Condition=cloudy ∨ sunny: 7,8] & [Temp= medium ∨ high: 7,7] ∟ [Condition=cloudy] & [Wind=yes] & [Temp= high] : p=7,n=0,q=1	An activity is play if condition is cloudy or sunny and temperature is medium or high, except for when condition is cloudy, there is wind and temperature is high. The rule covers 7 positive and no negative examples. Its quality measure (see Section 2.4.2) is 1.
[User=user3] ⇐ [Program=word ∨ excel] & [#Open windows=2..5] ⌜ [Day=monday]	The user is user3 if Program is word or excel, and the Number of open windows is between 2 and 5, provided that day is Monday.

Figure 2-5: Examples of attributional rules.

Output attributes (a.k.a. decision attributes or dependent attributes/variables) are attributes used in the CONSEQUENT part of rules. They are used to define classes or concepts. For example the output attribute in the first rule demonstrated in Figure 2-5 is "Part." In the second rule demonstrated in the same Figure, the output attributes are "Activity" and "Dress."

Input attributes (a.k.a. independent attributes/variables) are those used in the premise, exception, and/or precondition, that is, on the right side of rules. For example in the first rule demonstrated in Figure 2-5 the output attributes are "Width," "Length," and "Material." Please note that not all input attributes from the representation space are necessarily included in rules. Some input attributes may not be needed to describe positive examples, or are irrelevant to a given learning task.

A *ruleset* is a set of rules with the same consequent that represents a hypothesis describing one particular class. Rulesets may be complete and consistent, meaning that constituent rules cover all examples of a concept and the rules do not cover any examples of other concepts (negative examples), or rulesets may be partially incomplete or inconsistent.

A formula is in *disjunctive normal form* (DNF) if it is a disjunction of clauses that are either a single attributional condition or a conjunction of attributional conditions. For example (2-8) is in DNF, where A, B, C, D, E, and F are attributional conditions.

$$A \,\&\, B \vee C \vee D \,\&\, E \,\&\, F \qquad (2\text{-}8)$$

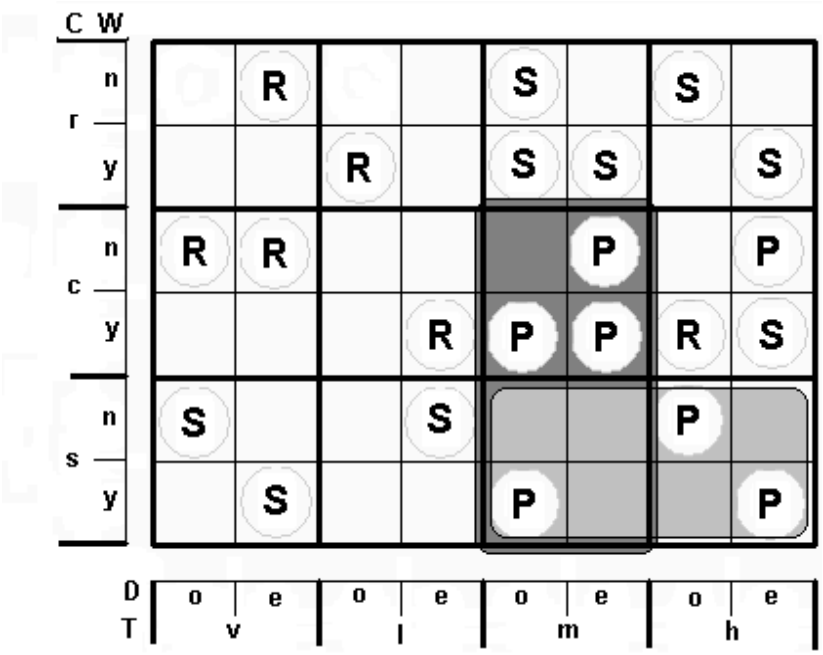
2.3.3 Generalized Logic Diagrams

Visualization of data and knowledge is a very important aspect of natural induction. One of many methods for visualizing representation spaces, examples and different forms of attributional rules is the *generalized logic diagram* (GLD; Michalski, 1972; Michalski 1978; Wnek, 1995; Sniezynski, Szymacha and Michalski, 2005).

Generalized Logic Diagrams provide a planar representation of the multidimensional representation space spanned over multiple-valued discrete attributes. Continuous attributes need to be discretized prior to being displayed. By properly ordering attributes assigned as axes of the diagram, patterns can be displayed in an easy to understand form. In GLDs each cell represents exactly one point in the discrete representation space, thus a set of examples can be easily represented as a set of points in the diagram. Each complex (conjunction of conditions) is represented by a rectangle or a set or rectangles.

An example of GLD is presented in Figure 2-6. The representation space is a Cartesian product of four attributes (Condition, Wind, Temperature, and Daytype). The diagram shows 22 examples (6 of Activity Read, 9 of Activity Shop, and 8 of Activity Play), and two complexes (shaded areas) belonging to rules describing activity “Play”.

```
[Activity=play]
⇐ [Condition=cloudy v sunny] & [Temperature=medium]
⇐ [Condition=sunny] &  [Temperature=medium v high]
```



C – Condition: r – rain, c – cloudy, s – sunny
W – Wind: n – no, y – yes
T – Temperature: v – very low, l – low, m – medium, h – high
D – Daytype: o – workday, e – weekend,
P – play, R – read, S - shop

Figure 2-6: An example of Generalized Logic Diagram.

In practice GLDs are practically applicable to about 5-10 attributes, depending on their domain sizes. For larger numbers of attributes the GLD's planar representation becomes too large, and projections onto smaller spaces are needed. In this dissertation, most examples of learned rules and individuals are presented using GLDs.

2.4 Concept Learning

Concept learning, a special case of supervised learning, is one of the most important and the best explored type of learning from examples. It focuses on creating general descriptions of concepts given set of labeled examples belonging to these concepts. Formally, the task of concept learning is defined as: given a set of examples (e_i, C_i) belonging to classes C_1 .. C_n, where e_i are elements of representation space E, the goal is to induce general descriptions of classes C_1 .. C_n based on the provided examples, using a given language L. Concept learning is often also described as learning a description of one class given examples belonging to that class and examples not belonging to it. An example of concept learning with an output hypothesis in the form of attributional rules is presented in Figure 2-7.

Attributes:

```
Shape nominal {square, oval, triangle}
Color nominal {red, green}
Class nominal {positive, negative}
```

Input examples:

```
square, red, positive
oval, red, positive
square, green, negative
triangle, red, negative
```

Output rule:

```
[Class=positive] ⇐ [Shape=square v oval] & [Color=red]
```

Figure 2-7: Example of concept learning.

The same concept can be learned and represented, for example, by a decision tree, as shown in Figure 2-8.

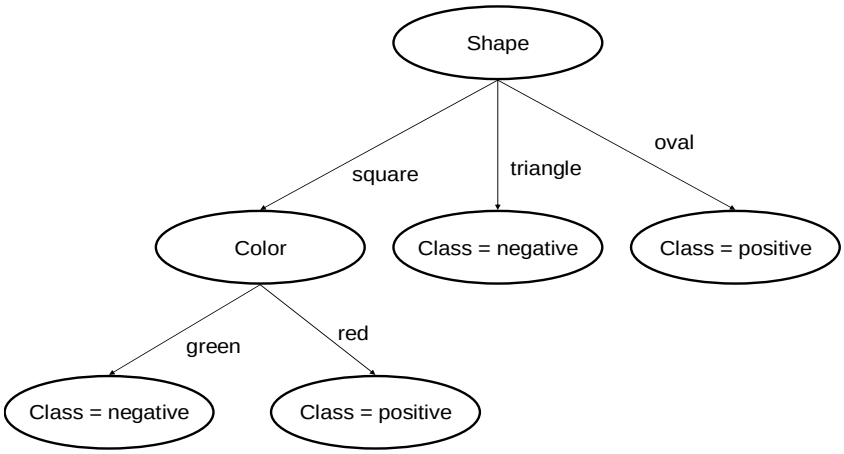


Figure 2-8: Decision tree learned from the input examples.

As shown in the above example, even in the very simple case, the decision tree and the attributional rule generalized examples in very different ways, although both generalizations are complete and consistent with regard to the provided examples (training data). This issue is investigated by Wnek and Michalski (1994), who show how different models (e.g. neural nets, attributional rules, decision trees) generalize.

The following Sections briefly present the AQ learning methodology and its features and methods that are used or can be used in the learnable evolution model. It also presents a summary of the most important features of the AQ21 learning system, to date the most advanced implementation of AQ learning, which is used for hypothesis formulation in the LEM3 system.

2.4.1 Basic AQ Algorithm

The well-known family of AQ programs originated with the A^q algorithm for solving the general covering problem (e.g. Michalski, 1969). Numerous implementations and extensions of the method were developed over the years. Among the best known AQ implementations are AQ7 (Michalski and Larson, 1975), AQ11 (Michalski and Larson, 1983), AQ15c (Wnek et al., 1995), AQ17 (Bloedorn et al., 1993), AQ19 (Michalski and Kaufman, 2001a) and most recently AQ21 (Wojtusiak, 2004a; Wojtusiak et al., 2006a,b).

A basic version of the AQ learning algorithm is presented in Figure 2-9. It takes as input a set of positive examples of a concept, P, and set of negative examples, N, belonging to all other classes (examples not belonging to the learned concept), and a multicriterion quality measure (lexicographical evaluation functional, LEF). It returns a complete and consistent hypothesis in the form of an attributional ruleset optimized according to the given lexicographical evaluation functional.

```
Hypothesis = null
While P is not empty
   Select a seed example p from P
   Generate star G(p, N)
   Select the best rule R from G according to LEF, and
     include it in Hypothesis
   Remove from P all examples covered by the selected rule
Return learned Hypothesis
```

Figure 2-9: Basic AQ algorithm.

The key part of the algorithm is the generation of star G(p, N) given a seed p and set of negative examples N. The star is a set of maximally general rules covering the seed p, but not covering any negative example from N. A star is constructed by intersecting

partial stars which are generated using the *extension-against* operator (e.g. Michalski, 1983). The *extension-against* that takes two data points and creates a set of maximal generalizations of one data point (a positive example) that does not cover the second data point (a negative example). The result of such an operation is a set of local stars. An intersection of local stars creates a star of the given seed. To narrow down a possibly very large number of intermediate generalizations, AQ uses beam search that at each step of star generation keeps no more than a predefined number of best rules, as determined by the given pattern quality measure.

To select a rule from a star, the algorithm uses a lexicographical evaluation functional (LEF), a user-defined multicriterion measure of rules' quality (see Section 2.1). The default criteria for selecting a rule from a star are to (1) maximize the number of positive examples covered by the rule and (2) minimize the number of conditions in the rule. A complete list of LEF criteria available in the AQ21 system is presented by Wojtusiak (2004a).

2.4.2 Learning Strong Patterns

The basic AQ algorithm presented in the previous section is designed to learn complete and consistent hypotheses with regard to the training data, and is known as *theory formation* (TF) mode. A modification of the method, called *pattern discovery* (PD) mode, is designed to search for strong patterns that maximize an assumed pattern quality measure. Similarly to the basic AQ algorithm shown in Figure 2-9, the method takes as input a set of positive examples P, a set of negative examples N, and a pattern quality

measure, called LEF, defined by the user. It returns a hypotheses consisting of strong patterns that characterize examples from the set P. The method is presented in Figure 2-10.

```
Hypothesis = null
While P is not empty
   Select a seed p from P
   Generate approximate star G(p, N)
   Select the best k rules from G according to LEF, and
     include in Hypothesis
   Remove from P all examples covered by the selected rules
Optimize final rules
Select the final hypothesis from all selected rules
```

Figure 2-10: AQ algorithm for learning strong patterns.

In PD mode the program starts by focusing attention on one data point, called the seed, and then creates a set of alternative approximate generalizations of the seed, called an *approximate star*. The generalizations are approximate, because while they do not have to be consistent with the data; they must optimize a pattern quality criterion.

The pattern quality measure, q(*w*), is defined by:

$$q(w) = cov^{w} * config^{1-w} \tag{2-9}$$

where

$$cov=|p| / |P| \tag{2-10}$$

and

$$config=((|p| / (|p| + |n|)) - (|P| /(|P| + |N|))) * (|P| + |N|) / |N| \tag{2-11}$$

are measures of pattern (here, attributional rule) coverage and confidence gain, respectively, and w is a user-defined parameter. Here, |p| and |n| are the numbers of

positive and negative examples covered by the rule, and |P| and |N| are the numbers of positive and negative examples in the training dataset, respectively (Michalski and Kaufman, 2001b). The q(w) definition (2-9) assumes that config ≥ 0 which means that a rule's prediction is better than a random guess. I practice the formula implemented in AQ21 distinguishes two cases to also incorporate config < 0.

Optimization of rules consists of several possible operations which may lead to improvement of rules' quality. The operations are: abstraction of conditions, specialization of conditions, removing conditions, and removing entire rules.

Well-known rule learning methods, such as RIPPER (Cohen, 1995) or CN2 (Clark and Niblett, 1989) also seek strong patterns, but AQ21 can determine both strong patterns and complete and consistent theories (hypotheses), depending on the setting of its parameters. Also, the patterns learned by AQ21 can be richer and of different types.

2.4.3 AQ21 Implementation of AQ Learning

Over the past three and half decades, the different versions of AQ programs implemented different methods extending the basic AQ algorithm. These programs were developed in different programming languages (PL/1, Lisp, Pascal, C/C++) and on different platforms. AQ21, which is the newest implementation of the AQ methodology, is an attempt to integrate and implement the most important features of the previous programs along with several new additions, which are the result of the newest research in the field. It also implements several features which are unique and not present in any other program.

Although AQ21 is still under development and many of the desired features are not yet implemented, to the best of author's knowledge it is the most powerful symbolic machine learning program ever developed. This implementation reuses the source code of the previous AQ20 system (Cervone, Panait and Michalski, 2001). AQ21 is described in this dissertation because it is used as a learning module in the LEM3 implementation of the learnable evolution model that is a basis for implementing presented methods. The following paragraphs briefly describe the most important features of AQ21.

AQ21 induces hypotheses that are represented as rules in attributional calculus. Depending on its settings, it learns rules in the forms (2-5) - (2-7) discussed in Section 2.3.2. It operates in three modes, namely *theory formation* (TF), *approximate theory formation* (ATF), and *pattern discovery* (PD). In the TF mode, it learns theories that are complete and consistent with regard to the training data using a modification of the algorithm presented in Section 2.4.1. In ATF mode, it first learns complete and consistent theories and then modifies them in order to improve the rules' quality. This operation may result in partial inconsistency and/or incompleteness of the learned hypotheses. In PD mode, AQ21 seeks strong patterns in data. The patterns may be (and usually are) neither complete nor consistent with regard to the input data. To do so, the program implements a version of the algorithm briefly presented in Section 2.4.2.

In order to deal with large and/or noisy datasets, AQ21 implements several features such as multiple seed selection, ordering of attributes before extension against, evaluation and selection of the most relevant attributes, and ordering of negative examples. AQ21 also

implements several unique features such as learning alternative hypotheses (Michalski, 2004b; Wojtusiak et al., 2006a,b), learning rules with exceptions in the form (2-6), improving representation spaces (see Chapter 5), and using meta-values (Michalski and Wojtusiak, 2005).

For testing and application of attributional rulesets AQ21 implements two methods: ATEST and EPIC. The ATEST method (e.g., Reinke, 1984) is used for the testing and application of attributional rulesets to individual testing/application examples. EPIC (e.g., Wojtusiak, 2004a; Michalski et al., 2005) similarly applies attributional rulesets to sequences of examples to be classified as a whole. For example, in the application to computer user profiling, we are not interested in identifying the user responsible for each individual command in the database; rather, the program is provided with a sequence of commands bundled together, for which it establishes the responsible user.

The general methodology of ATEST and EPIC is as follows:

1) For each individual testing example, determine a degree of match between it and each decision rule.

2) For each decision class, aggregate the degrees of match determined in step 1, in order to determine degrees of match between each testing example and each decision class.

3) If using EPIC, aggregate the degrees of match determined in step 2 for the examples in each sequence, in order to determine degrees of match between the sequence and each decision class.

4) Based on the calculated degrees of match and threshold and tolerance parameters, output a classification for each testing example (ATEST) or sequence (EPIC). There are several matching and aggregation methods available in steps 1-3. Ones appropriate to the task may be selected by the user (Wojtusiak, 2004a).

Both methods, in addition to predictive accuracy (see Section 2.1) based on the best match, report also precision, because AQ21 is able to classify an event (sequence) to more than one class. In many applications, it is more appropriate to give an imprecise classification rather than to give a wrong answer. For example, when diagnosing diseases, the program may give the answer with list of possible diseases.

CHAPTER 3 LEARNABLE EVOLUTION MODEL

This chapter describes the learnable evolution model (LEM), a novel non-Darwinian evolutionary computation method. It also presents selected features of LEM3, the most recent LEM implementation, which is the basis for methods of handling constraints and improving representation spaces presented in Chapters 4 and 5, respectively. This chapter concludes with a brief discussion of other non-Darwinian evolutionary computation methods and their relation to LEM and LEM3.

3.1 The Basic Idea of the Learnable Evolution

Research on non-Darwinian evolutionary computation is concerned with developing algorithms in which the creation of new candidate solutions in the population is guided by an "intelligent agent," rather than done merely by random or semi-random change operators, such as mutation and/or crossover, employed in the "Darwinian-type" evolutionary methods. The selection of candidate solutions for the new generation from those generated by the intelligent agent is done according to standard methods of selection, or can also be done by employing advanced reasoning. The learnable evolution model (LEM) employs a learning program to direct the evolutionary process (e.g., Michalski 1998; 2000; Wojtusiak and Michalski, 2006). Specifically, the program

creates general hypotheses indicating regions in the search space that likely contain optimal solutions and then instantiates these hypotheses to generate new candidate solutions.

The learnable evolution model follows a general evolutionary computation schema presented in Figure 2-2. Specifically, LEM follows the algorithm presented in the Figure 3-1, and creation of new candidate solutions is done by applying hypothesis learning and instantiation as illustrated in Figure 3-2.

```
Create an initial population of candidate solutions
Evaluate candidate solutions in the initial population
Loop while stop criteria are not satisfied
      Create new candidate solutions by machine learning:
         Identify groups of high- and low-performing candidate
           solutions
         Apply machine learning to distinguish between the groups
         Instantiate the learned hypothesis
      Evaluate fitness of the new candidate solutions
      Select a new population
```

Figure 3-1: Pseudocode of a general LEM algorithm.

The assignment of high- and low-performing candidate solutions into the *H-group* and *L-group* allows these sets to be used as examples for a concept learning program. The concept learning program generates a hypothesis determining why candidate solutions in the H-group perform better than these in L-group. Such a hypothesis is then instantiated to generate new candidate solutions which are likely to be high-performing because they satisfy the hypothesis description. These three steps for generating new candidate solutions by machine learning are depicted in Figure 3-2 and are described in detail in the following sections.

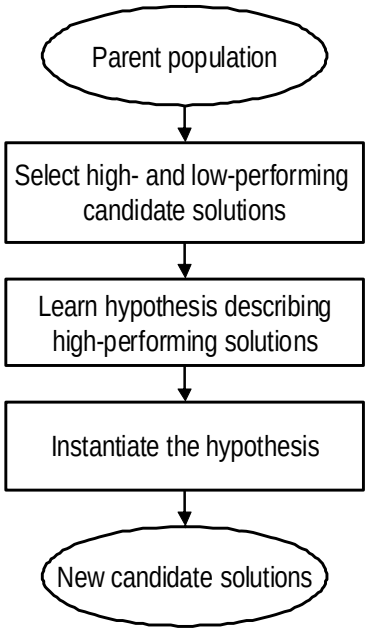


Figure 3-2: Generating new candidate solutions by machine learning.

3.2 Selection of Examples for Hypothesis Generation

Before concept learning is applied, examples of high-performing and low-performing candidate solutions need to be selected. The examples are selected from the current population and optionally also from previous populations, depending on the lookback parameter. The set of selected high-performing candidate solutions is called the H-group and the set of selected low-performing candidate solutions is called the L-group.

Michalski (2000) proposed two methods of creating the groups of high-performing and low-performing candidate solutions from the current population. One, *fitness-based selection*, defines high and low fitness thresholds in the range from the highest to the lowest fitness value observed in the current population. For example, if high and low fitness thresholds (HFT and LFT) are both 25%, then candidate solutions whose fitnesses

are in the highest 25% of the range and the lowest 25% of the range are included in the H-group and L-group, respectively. The second method, *population-based selection*, selects a specified percentage of candidate solutions from the population for each group, regardless of the distribution of fitness values. These percentages are defined by the high population threshold (HPT) and low population threshold (LPT). For example, if HPT and LPT are both 30%, then the 30% of the candidate solutions with the highest fitness and the 30% with the lowest fitness are included in the H- and L-group, respectively.

Selection of examples provided to a learning program depends not only on the selection method and its threshold, but also on a population size. The larger the population size, the more candidate solutions are selected as training examples, assuming the same values of the thresholds. Initial experimental study has shown that LEM is not sensitive to population sizes if the sizes are sufficiently large, as results similar in terms of accuracy and evolution length were achieved with different population sizes. Population sizes in LEM need to be larger than in Darwinian-type evolutionary computation methods such as genetic algorithms, because the learning module in LEM requires a sufficient number of examples to perform learning.

3.3 Learning Hypothesis Describing High-Performing Candidate Solutions

Once the H-group and L-group are selected, they are provided to the learning module in order to generate general hypotheses that characterize the high-performing candidate solutions in contrast to the low performing ones. This process reflects the behavior of human experts, who compare better and worse candidate solutions in order to understand

reasons for differences in their performance. Equipped with that knowledge, the expert is able to design new candidate solutions that are likely to perform well.

The learnable evolution model is a general methodology that allows virtually any concept learning program to be employed for hypotheses formulation. The only requirement is the existence of a method for instantiating hypotheses learned by that program. Different implementations of LEM used different learning methods, most of which were based on AQ rule learning. There are also implementations that use different learning methods, for example, the c4.5 decision tree learning method (Quinlan, 1993) is used in the LEMMO system (Jourdain et al., 2005). The reason AQ attributional rule learning is suitable for LEM systems is that generated hypotheses are represented using rules in a highly expressive language, attributional calculus. Such rules are efficiently instantiable and are easy for human experts to understand (see Chapter 2).

Given positive and negative examples of a concept, AQ induces a general concept description in the form of an attributional *ruleset*, a set of rules with the same consequent. The simplest form of an attributional rule is (2-5) CONSEQUENT $\Leftarrow$ PREMISE, where CONSEQUENT and PREMISE are conjunctions of attributional conditions (Chapter 2). In LEM the CONSEQUENT always defines high-performing candidate solutions.

Figure 3-3 shows an example of a rule learned by AQ21 (in LEM3) during the optimization of the Rosenbrock function of 10 variables. The function is described and graphically illustrated in Chapter 6.

```
[Group=H] ⇐ [x0=-0.5..1.5: 28,19] &
            [x4=-0.5..2.0: 15,16] &
            [x5=-1.5..1.5: 18,12] &
            [x8=-0.5..1.5: 30,28] &
            [x9=-0.5..1.5: 27,22]: p=12,n=0
```

Figure 3-3: A rule learned during optimization of the Rosenbrock function.

The rule sates that if attribute x0 takes a value between 0.5 and 1.5, x4 takes a value between -0.5 and 2.0, and so forth, then the candidate solution belongs to a (generalized) H-group. The pairs of numbers after ":" in each condition indicate the positive and negative coverage (support) for this condition. For example, the condition specifying the value of x0 is itself satisfied by 28 candidate solutions in the H-group and 19 candidate solutions in the L-group. The numbers p and n indicate the coverage of the entire rule (12 in the H-group and 0 in the L-group).

An important problem is to determine the optimal parameter settings of a learning program used in LEM that give the best results. Learning programs from the AQ family provide the ability to control types of learned descriptions. AQ21 allows control of many parameters such as the generality of rules, types of rules (complete and consistent, approximate, patterns, with and without exceptions etc.), types of descriptions (characteristic, discriminant, simplicity-based), etc. Although all of these possibilities are present, it is unclear for what types of optimization problems they should be used in LEM. Different problems may require different settings, and a detailed study is required to find the appropriate AQ21 parameter settings (see Chapter 7). Although this matter requires detailed study, LEM3 uses experimentally found default values and adds the possibility of full control of the algorithm by the user (Wojtusiak, 2004b).

3.4 Instantiation of Learned Hypotheses

The learned hypotheses are used to generate new candidate solutions by the instantiation process. The next sections describe methods for instantiating attributional rules learned by AQ systems. Basic attributional rules are conjunctions of conditions that define ranges (or sets) of attribute values, thus the instantiation of such rules is a relatively easy process.

When instantiating a rule for a member of the new population, the program faces two problems: what values to assign to attributes that are cited in the rule, and what values to assign to attributes not present in the rule. The latter is exemplified in the rule illustrated in Figure 3-3, which does not include attributes x1, x2, x3, x6 and x7. Three algorithms for instantiating attributional rules used in LEM3 are described in the following sections.

3.4.1 Instantiation Algorithm 1

This section describes the simplest algorithm for generating new candidate solutions by instantiating attributional rules. In the first step the algorithm takes all rules from a learned ruleset and for each rule computes the number of candidate solutions to be generated. The total number of candidate solutions that are created can be either constant during the evolution or may vary over time; it is defined as a parameter by the user. The number of candidate solutions can be the same for all rules, or can be computed proportionally according to a measure of the rules' significance that is calculated as the sum of the fitness values of the high-performing candidate solutions covered by the rule.

For each newly created candidate solution, the program has to assign values for all attributes, both to those included in the rule being applied, and to those not included in the rule. Depending on the attribute type and user-defined parameters, different distributions can be used to select random values for the attributes specified in the rule; it can be done using a uniform distribution, using a normal distribution for numerical attributes with mean equal to the middle of the range and user-defined variance, using the maximum distance from negative examples, or using projections of positive and negative examples.

```
For each rule in a ruleset (hypothesis) to be instantiated:
  Compute the number of candidate solutions to be created
  For each candidate solution to be created:
    For each attribute:
      If the attribute is specified in the rule
        Select a random value satisfying the rule
      Else Select a random candidate solution from the previous
        population and use its value
```

Figure 3-4: Instantiation algorithm 1.

Selection of values of attributes not specified in the rule is a more intricate problem, for which there are many potential solutions. One possibility is to select a random value from the entire attribute domain. This will result in candidate solutions consistent with the rule; however, it is easy to show some cases in which this approach will result in poor performance. For instance, let us assume that the goal is to optimize a function with two attributes x and y. Both attributes are continuous and defined on the range -5 to 5, and the function optimum is at the point (0, 0). Let us further assume that a learning program has found the rule with one condition [x = 0]. The program will then generate candidate solutions with x = 0 in all cases, and with y distributed over the range [-5, 5]. In the next

iteration, the program will learn rules containing only the attribute y, since there is no differentiation among the x-values any longer. During the instantiation phase, the program will assign values of the attribute x randomly, which means that the information from the previous iteration is lost. Thus, the rules may not converge to the solution.

Another method of value selection, that solves the above problem, is to select the value from a randomly selected existing candidate solution. The candidate solution can be selected from the entire population, the H-group, or non-L-group candidate solutions. Experiments have shown that when values are selected from the H-group, the program tends to lose diversity of candidate solutions, and may converge very quickly to a point that is not the target solution. The method that LEM3 uses by default selects candidate solutions from the whole population probabilistically in proportion to their fitness values.

The presented instantiation algorithm has a very severe weakness. It does not work well for multimodal functions. Candidate solutions selected to provide values of attributes that are not specified in a rule may be located in different parts of the search space, thus causing new candidate solutions to be generated in the wrong parts of the space. As an extension to the algorithm, a constraint may be added to ensure that generated events match the rule that is instantiated. However, this also may not keep the evolution process from straying in all cases. The second instantiation algorithm was designed to cope with the described problems.

3.4.2 Instantiation Algorithm 2

The second instantiation algorithm selects parent candidate solutions and then modifies them according to learned rules. As mentioned before, it is designed to cope with problems that appear when using the first instantiation algorithm. The difference between this and the previous algorithm is in the way a candidate solution from the old population is selected. This algorithm is appropriate for using with multimodal functions.

```
Loop while not all candidate solutions have been created:
  Select probabilistically a candidate solution (parent) based on its
    quality
  Create a list of rules satisfied by the selected candidate solution
  Select a matching rule probabilistically in proportion to its
    significance
  Create a new candidate solution:
      For all attributes:
        If the attribute is specified in the selected rule
          Select a random value satisfying the rule
        Else select the value from the parent
```

Figure 3-5: Instantiation algorithm 2.

The typical quality measure of candidate solutions is simply their fitness. It is not, however, the only possibility; for example, a quality measure may also take into consideration the number of rules that match the candidate solution. When AQ21 is working in the theory formation (TF) mode (i.e., learning covers that are complete and consistent with respect to the training data), it is guaranteed that all candidate solutions from the H-group will satisfy some rules. Although LEM uses the TF mode by default, it is also possible to learn rules in the approximate theory formation, or the pattern discovery modes. In these modes, stronger rules/patterns may be favored over complete and consistent rulesets. In such a case, some candidate solutions from the H-group may not be covered by any rule. Therefore, it is important to select only those candidate

solutions that are covered by at least one rule. Values of attributes that are specified in the rule are selected in the same way as in the Algorithm 1.

3.4.3 Instantiation Algorithm 3

Regardless of the learning mode, AQ programs guarantee that each rule will cover at least one high-performing candidate solution. Using that information, it is possible to combine the two algorithms presented above. Similarly to Algorithm 1, the third instantiation algorithm computes the number of candidate solutions to be instantiated for each rule. A significant difference from the first algorithm is that the program does not select random values according to the rules, but instead modifies an existing candidate solution that is covered by the rule. This guarantees that all the rules will be instantiated, and multimodal fitness functions will be treated appropriately.

```
For all rules:
  Compute the number of candidate solutions to be created
  For all candidate solutions to be created:
    Select probabilistically a candidate solution covered by the rule
    For all selectors in the rule:
      Modify value of the candidate solution within the selector
```

Figure 3-6: Instantiation algorithm 3.

3.4.4 Instantiation of Alternative Hypotheses

The AQ21 program has the unique feature to learn not only one ruleset per class, but also a number of alternative descriptions/rulesets for the same class (Michalski, 2004b; Wojtusiak et al., 2006a). In LEM, when the number of attributes may be much larger

than the number of examples, it is highly probable that the program can generalize the positive examples in many different ways, creating several alternative hypotheses.

LEM3 handles alternative hypotheses learned by AQ21 in two ways: (1) the intersection of the covers can be instantiated, or (2) the union of the covers can be instantiated. Once the program computes either the intersection or union, one of the three algorithms described above is then used to instantiate.

By using the intersection method, the program creates candidate solutions in an area covered by at least one rule from each alternative ruleset. Suppose that RS_1, RS_2, …, RS_n are alternative rulesets describing the high-performing candidate solutions. Ruleset RS_1 consists of k1 rules: RS1 = { $R_{1,1}$, $R_{1,2}$, … $R_{1,k1}$ }, ruleset RS_2 consist of k2 rules RS_2 = { $R_{2,1}$, $R_{2,2}$, … $R_{2,k2}$ }, and so on. A ruleset is a disjunction or rules that are conjunctions of selectors, so the intersection of rulesets is equivalent to a conjunction of rulesets and is given by the following formula:

$$\Lambda_{i=1..N} RS_i = \Lambda_{i=1..N} (R_{i,1} \vee R_{i,2} \vee \ldots \vee R_{i,ki}) \qquad (3\text{-}1)$$

Using De Morgan's and absorption laws the intersection RS can be easily computed. In fact, computation of such an intersection is one of the most common operations in the AQ algorithm applied during the star generation phase. In LEM3 we use this feature of AQ21 to compute the intersection. By nature, the intersection of alternative rulesets for a given class is also a ruleset.

When AQ21 works in TF mode and all rulesets are complete and consistent, the intersection is also a complete and consistent ruleset. To prove this, it is sufficient to

mention two facts: by the assumption, none of the rules in the rulesets cover any negative examples, so their intersection cannot cover any such examples; and each positive example is covered by at least one rule from each ruleset, so it will be covered in the intersection. Let e be a positive example that is covered by rules $R_{1,m1}$, $R_{2,m2}$, … $R_{n,mn}$. It is straightforward that $\Lambda_{1..n} R_{i,mi}$ covers the example e. Instantiation of the intersection of alternative rulesets speeds up the evolution process by limiting the area covered by learned rules. It may, however, lead evolution in wrong direction, since intersected rulesets may be too specialized, and the program may converge to a point that is not necessarily an optimal solution.

The second method is to take the union of alternative rulesets. The union is defined using the following formula:

$$V_{i=1..N} RS_i = V_{i=1..N} (R_{i,1} \vee R_{i,2} \vee \ldots \vee R_{i,ki}) \quad (3\text{-}2)$$

In this case, computation of RS is trivial and requires only the use of absorption laws in order to remove unnecessary rules. Similarly to the case of intersection, it can be proven that the union of complete and consistent rulesets is itself also a complete and consistent ruleset. Unlike intersection, the union expands the area in which new candidate solutions are instantiated. This slows down the evolution process, but increases the chance that the target solution is covered.

3.5 LEM3 Implementation of the Learnable Evolution Model

The presented work is based on the most recent version of the learnable evolution model implemented in the George Mason University Machine Learning and Inference

Laboratory, called LEM3. It is up to date the most advanced implementation that combines several features not present the previous implementations (e.g. Wojtusiak, 2004b; Wojtusiak and Michalski, 2006). This section briefly describes the LEM3 system, and presents its algorithm and selected novel features.

3.5.1 LEM3 Algorithm

Following the regular schema of evolutionary computation, LEM3 algorithm consists of several standard features such as generation of the initial population, evaluation of candidate solutions, and selection of populations. Other features introduced in LEM3 are action selection, adjustment of representation, and use of different innovation actions. The system is also able to support different attribute types defined in attributional calculus (see Section 2.3.1), which makes it applicable to a wide range of real world problems. The LEM3 algorithm in pseudocode is presented Figure 3-7, and its flowchart in Figure 3-8.

```
Generate initial population
Loop until the stop condition is satisfied
   Evaluate candidate solutions
   Select parent population
   Select one or more of the following actions:
      Learn and instantiate hypothesis that discriminates high and low
        performing candidate solutions in the parent population
        (Learning Mode)
      Generate new individuals through Darwinian-type operators
        (Probing Mode)
      Change the representation of individuals
      Randomize the population (either partially or via a start-over
        evolution process)
      Search locally
Compute statistics and display results
End LEM3
```

Figure 3-7: Pseudocode of LEM3 Algorithm.

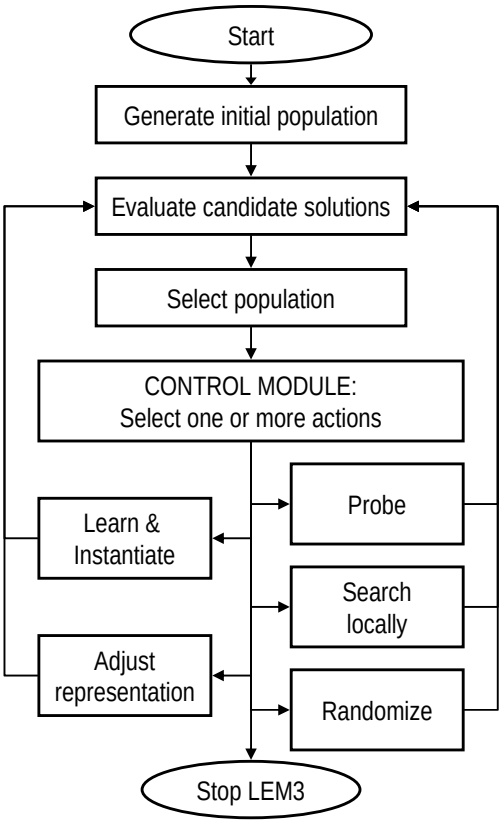


Figure 3-8: Flowchart of LEM3 Algorithm.

The following sections describe two unique features of LEM3: innovation operators and their selection. Adjustment of representation, which is one of two main topics of this dissertation, is discussed in Chapter 5.

3.5.2 Innovation Operators in LEM3

The process of creating new candidate solutions is called *innovation*. LEM3 is a multistrategy evolutionary computation method that integrates several innovation methods and can apply them in different combinations. The main innovation action in the learnable evolution model and its LEM3 implementation is hypothesis formulation and instantiation. It uses the AQ21 learning module to hypothesize why some candidate

solutions perform better and some perform worse. Such hypotheses are instantiated in different ways in order to produce new candidate solutions.

In addition to standard learning and instantiation action, LEM3 implements conventional methods of creating candidate solutions. These include *probing* that applies mutation and recombination, well known in the field of evolutionary computation, *randomizing* that randomly generates a given number of candidate solutions, or restarts the evolutionary process, and *searching locally* that applies user-defined local search methods (e.g. gradient-based operators).

3.5.3 Action Selection

Execution of different modes of operation is a unique feature of LEM3 that distinguishes it from other implementations of the learnable evolution model and from many other evolutionary computation methods. As mentioned above, the two basic modes that guide the evolution process are learning mode, which uses hypotheses creation and instantiation, and probing mode, which employs Darwinian-type operators such as mutation and crossover. In addition to the two modes of operation, LEM3 employs additional actions including adjusting discretization and randomizing designed to help the evolution process.

An important question is when each action should be executed. The two main modes of operation can be executed in parallel, or the program can switch between them as defined in *duoLEM* (e.g. Michalski, 2000). Other operations such as changing representation

need to be executed separately. To control the application of different actions, LEM3 defines an *action profiling function* (APF) that, based on the performance of different types of operators, decides which operators should be applied in the next step. It also decides how many new individuals to create in each mode. For example, if the total number of new individuals to be created is 100, the APF may decide to generate 70 by the learning mode, 25 by the probing mode and 5 by randomizing. The APF should adapt during the evolution process to reflect which operators are the most relevant for the optimization problem. Controlling the learning and probing modes can be done by two simple rules:

```
If average-learning-fitness >> average-probing-fitness then
        Increase number of individuals in learning mode
If average-learning-fitness << average-probing-fitness then
        Increase number of individuals in probing mode
```

where averages are computed for candidate solutions created in one or more iterations of the respective modes. In order to avoid extinction of one mode, a minimum number of candidate solutions in each mode can be defined, regardless of its performance.

APF can also support problem-oriented operators defined by an expert. For example, in numerical domains, a wide range of gradient-based operators can be programmed into LEM3 and appropriately handled.

During the evolution process, it may happen that over a number of iterations, the program does not make sufficient progress in terms of the value of the fitness function. This situation can be identified through the use of the *no-progress condition* that utilizes two program parameters, *learn-probe* and *learn-threshold*. Learn-probe defines the

maximum number of iterations that are performed even if there is unsatisfactory progress, as defined by learn-threshold, the minimal acceptable increase of fitness of the best candidate solution. When the *no-progress condition* is satisfied, several possible operations are considered. If the no-progress condition is met, mutation, adjust discretization (or in general, adjust representation), and/or start-over operators are invoked. LEM3 tries to apply mutation for *mutation-probe* iterations. If there is still no progress, the program then tries to adjust discretization for *discretization-probe* iterations. If there is still no progress, LEM3 tries to run the start-over operation for no more than *start-over-probe* iterations.

```
Increment learn-probe-counter
If learn-probe-counter >= learn-probe
      Learn-probe-counter = 0
      If mutation-probe-counter < mutation-probe
            Increment mutation-probe-counter
            Mutate candidate solutions
            Evaluate modified candidate solutions
      Else if discretization-probe-counter < discretization-probe
            Increase discretization-probe-counter
            Mutation-probe-counter = 0
            Adjust discretization
            Mutate candidate solutions
            Evaluate modified candidate solutions
      Else if start-over-probe-counter < start-over-Probe
            Increment start-over-probe-counter
            Discretization-probe-counter = 0
            Mutation-probe-counter = 0
            Rollback discretization
            Add the best solutions to a list of local optima
            Start-over
            Evaluate candidate solutions
      Else
            Stop LEM3
```

Figure 3-9: Pseudocode of no-progress condition.

The order of mutation, adjust discretization, and start-over operations is not accidental. Mutation is performed in order to introduce diversity into a population, and test if the

program did not get stuck "close" to the optimal value (this could be a local or global optimum). It is usually the case that AQ21 is unable to learn hypotheses because of a lack of diverse examples. The next step increases the precision of the search by adjusting discretization. If the change of precision does not make any difference, it may mean that the program has found an optimum. However, the optimum may be local and it may be desirable to start over the evolution process with a new random population to explore different parts of the search space. Details of these actions are described by Wojtusiak and Michalski (2005; 2006).

3.6 Example Execution of LEM3 and EA

To illustrate LEM3 on a simple optimization problem, this section presents an example of applying LEM3 and a conventional, Darwinian-type algorithm, here called evolutionary algorithm (EA) to a function optimization problem (Wojtusiak and Michalski, 2005). The EA was implemented using the evolutionary objects (EO) library available at http://eodev.sourceforge.net. The problem is simple enough to be illustrated graphically using generalized logic diagrams but sufficiently complex to show some important aspects of the LEM3 algorithm.

The optimization problem is to find global maxima of the sample function:

$$f(x_0, x_1, x_2, x_3) = 16 - x_0^2 - x_1^2 - 8 * \cos(2 * x_2) - x_3^2 \qquad (3\text{-}3)$$

Domains of all attributes are ranges [-2, 2]. The cosine part was added to the function in order to make two equal global optimal solutions to the problem. A two dimensional

version of the function, given by the formula $f(x_0, x_1) = 8 - x_0^2 - 4 * \cos(2 * x_1)$ is illustrated in Figure 3-10. The factors 4*n and 2*n (values 16, 8 and 8, 4 in expressions above), where n is the number of attributes, are used the function for scalability and to guarantee that its value is not negative.

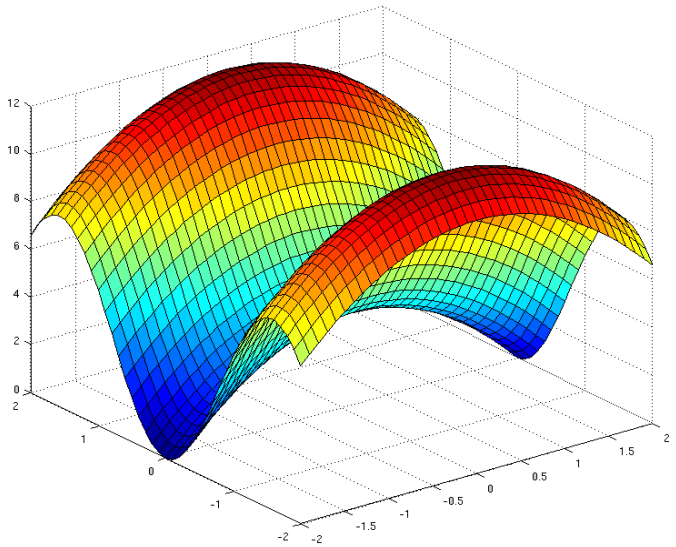


Figure 3-10: A plot of the function $f(x_0, x_1)$.

The following illustrations demonstrate consecutive steps of executing LEM3 and EA. The figures use generalized logic diagrams to represent four dimensional spaces spanned over discretized variables $x_0,\ldots,x_3$. The discretized values {0, 1, 2, 3, 4} in the diagrams correspond to the original values {-2, -1, 0, 1, 2} respectively. For each generation, two diagrams are presented for LEM3, one with the current population and one with selected solutions in the H-group and L-group) and the learned rules. For comparison, steps of executing EA for this problem are also illustrated. A similar method of illustration of LEM2 on the Rosenbrock function was presented by Cervone (1999).

Generation 1: In the first iteration, the population is randomly initialized in the entire search space as shown in Figure 3-11. Initial populations in LEM3 and EA (Figure 3-12) are the same. LEM3 selects candidate solutions for the H- and L-groups indicated as respectively "1" and "2", and provided with the groups AQ21 learns rules presented as shaded areas in Figure 3-13.

LEM3

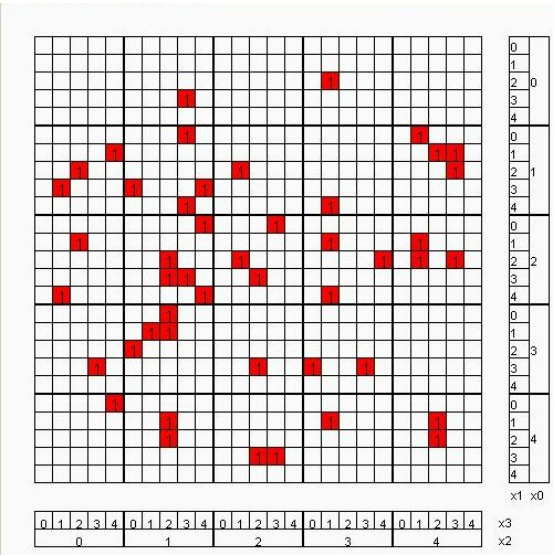


Figure 3-11: Randomly generated initial population (the same for both programs).

EA

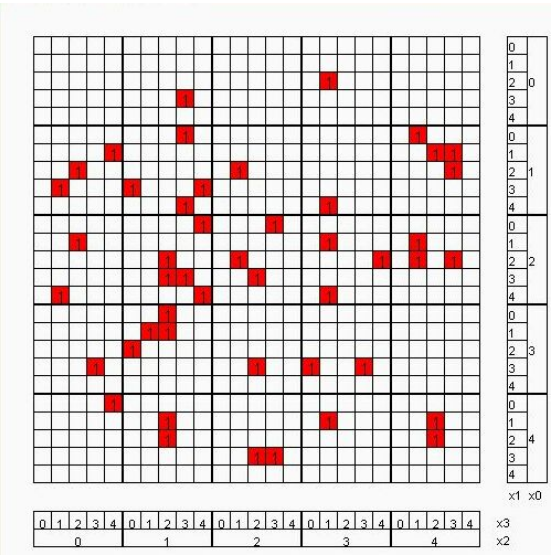


Figure 3-12: Randomly generated initial population (the same for both programs).

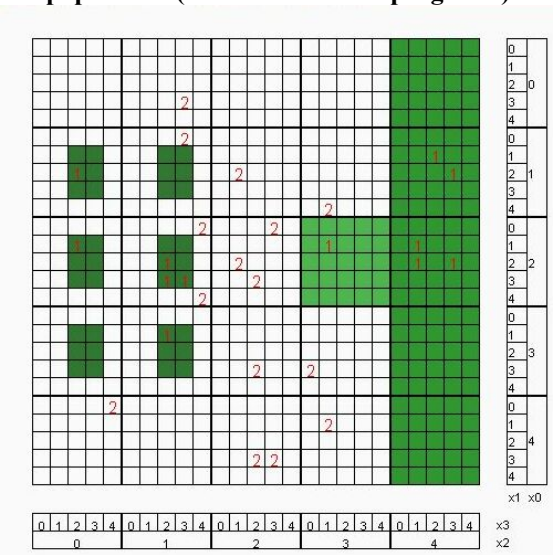


Figure 3-13: Learned hypothesis and H- and L-group candidate solutions in LEM3.

EA applies mutation and crossover to generate new candidate solutions.

EA probability of mutation is *0.1*
EA probability of crossover is *0.1*
EA selection method is *tournament.*

Generation 2: Instantiated candidate solutions are combined with the old candidate solutions and a new population is selected (Figure 3-14). In the instantiated candidate solutions both solutions of the function have already been found (0, 0, -2, 0) and (0, 0, 2, 0). These solutions correspond to the discrete points (2, 2, 0, 2) and (2, 2, 4, 2) illustrated using GLDs. The found solutions are maxima of the function assuming the used discretization, as the real solutions are (0, 0, $\pm\pi/2$, 0). This discretization is used only for demonstration purpose and it is sufficient for this example.

Although the program has found the solutions, it still needs for all candidate solutions to converge to the solutions in order to satisfy the LEM3 stop condition (the only candidate solutions are solutions). Figure 3-16 shows high and low performing candidate solutions selected from the population and learned rules (shaded areas). Evolutionary algorithm slowly converges toward the solutions as shown in Figure 3-15.

LEM3

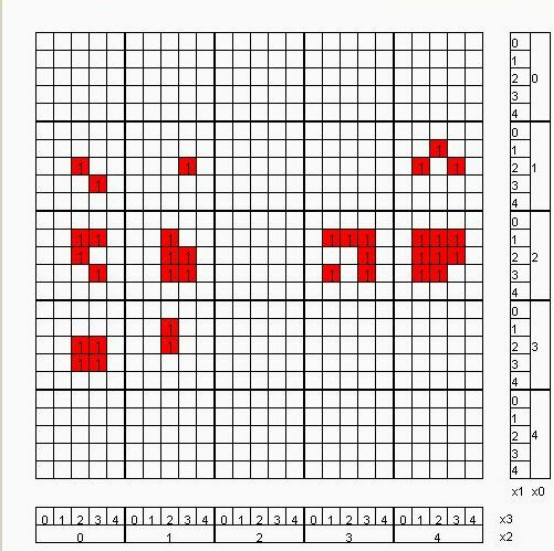


Figure 3-14: LEM3 Population in the second generation (100 fitness evaluations).

EA

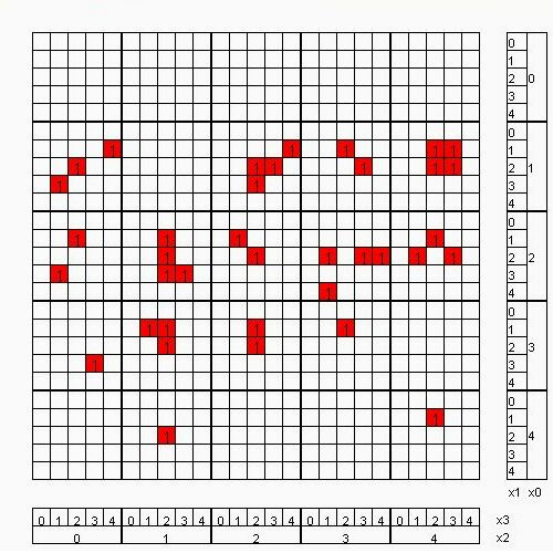


Figure 3-15: EA Population in the second generation (85 fitness evaluations).

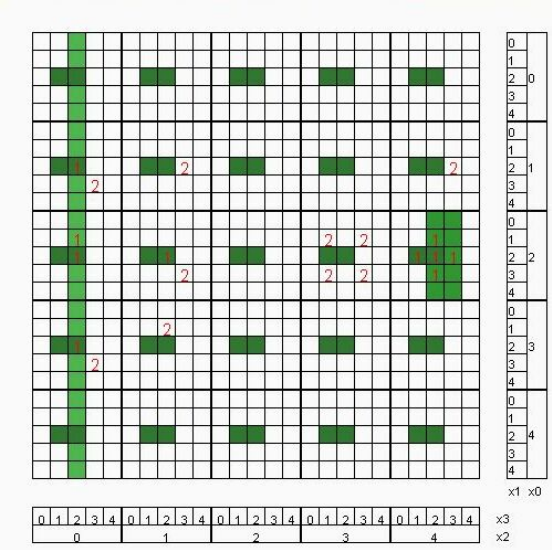


Figure 3-16: Learned hypothesis and H- and L-group candidate solutions in LEM3.

EA applies mutation and crossover to generate new candidate solutions.

EA probability of mutation is *0.1*
EA probability of crossover is *0.1*
EA selection method is *tournament.*

Generation 3: In the third generation, LEM3 candidate solutions converge closer to the solutions as shown in Figure 3-17. After selecting L- and H-groups from the population, one rule is learned as shown in Figure 3-19. EA also found the solutions, but the population is much more distributed over the space (Figure 3-18).

LEM3

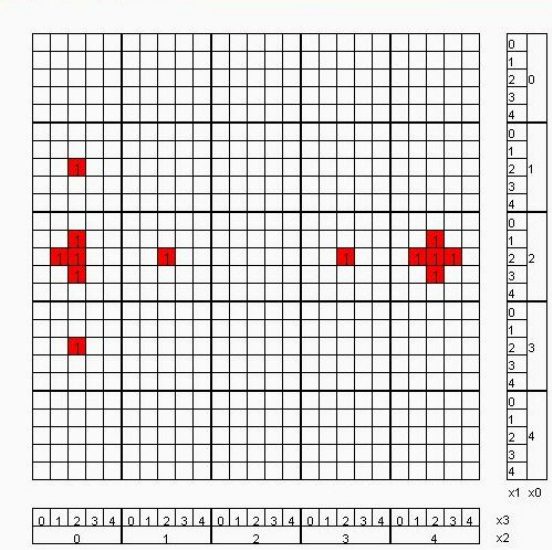


Figure 3-17: LEM3 Population in the third generation (150 fitness evaluations).

EA

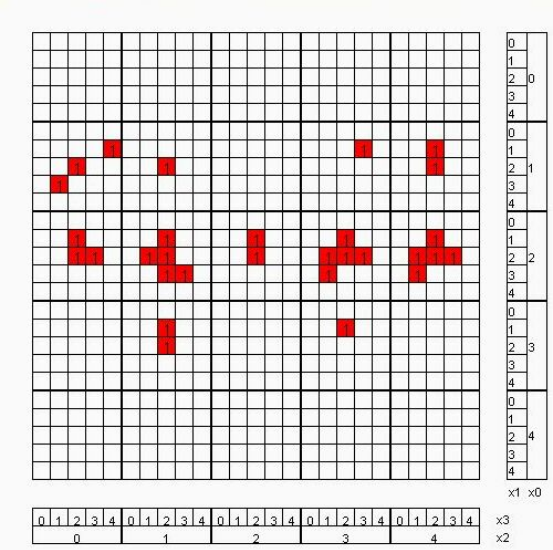


Figure 3-18: EA Population in the third generation (130 fitness evaluations).

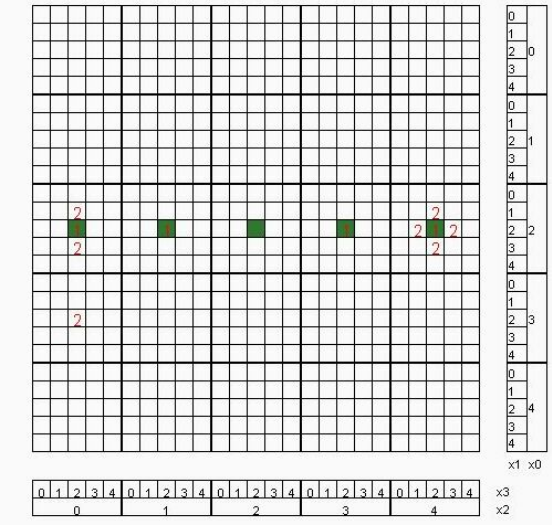


Figure 3-19: Learned hypothesis and H- and L-group candidate solutions in LEM3.

EA applies mutation and crossover to generate new candidate solutions.

EA probability of mutation is *0.1*
EA probability of crossover is *0.1*
EA selection method is *tournament.*

Generation 4: The fourth generation in LEM3 consists of candidate solutions that converged to the two solutions and one candidate solution that is not a solution (Figure 3-20). The only candidate solution that is not a solution is used as the L-group, and all other candidate solutions are included in the H-group. One simple rule describes the H-group against the L-group (Figure 3-22). EA slowly converges towards solutions as depicted in Figure 3-21.

LEM3

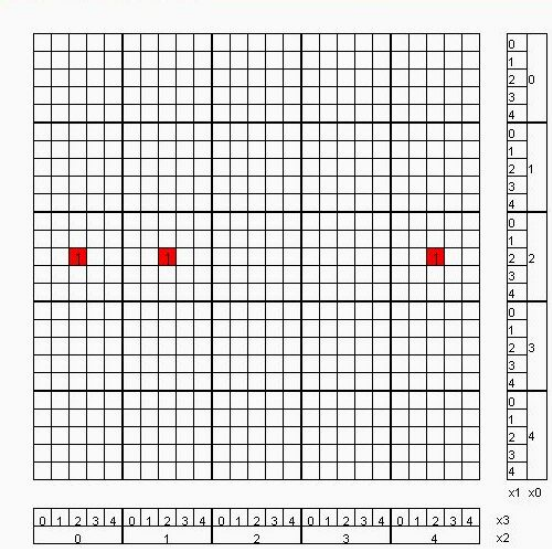


Figure 3-20: LEM3 population in the fourth generation (200 fitness evaluations).

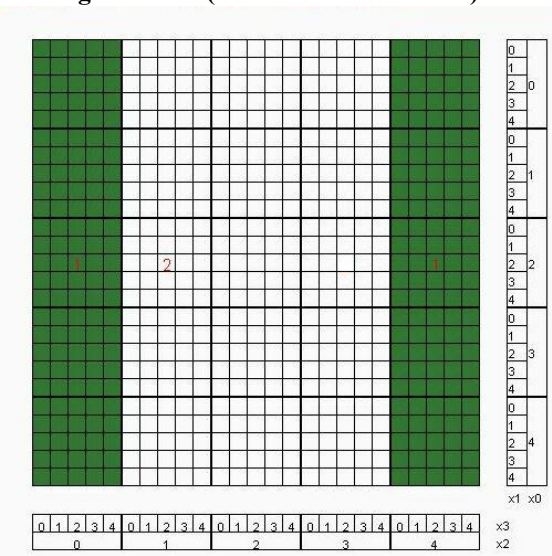


Figure 3-22: Learned hypothesis and H- and L-group candidate solutions in LEM3.

EA

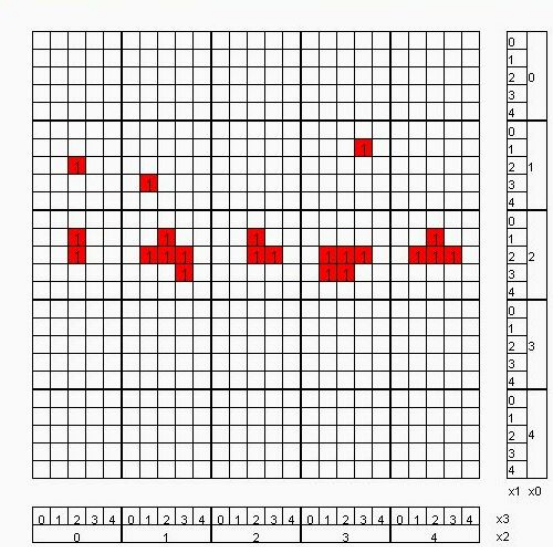


Figure 3-21: EA Population in the fourth generation (168 fitness evaluations).

EA applies mutation and crossover to generate new candidate solutions.

EA probability of mutation is *0.1*
EA probability of crossover is *0.1*
EA selection method is *tournament.*

Generation 5: Finally, in the fifth generation (250 fitness evaluations) all candidate solutions generated by LEM3 converged to the two solutions (Figure 3-23). This ends the evolution process in LEM3. EA did not converge to the solutions yet (Figure 3-24).

LEM3

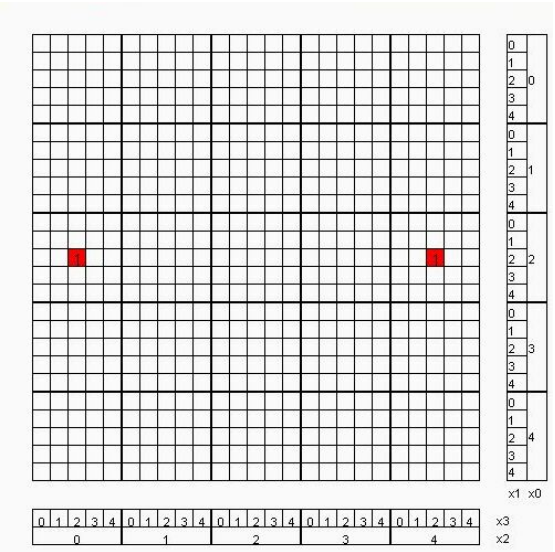


Figure 3-23: Two global optima found by LEM3 in the last, fifth generation (250 fitness evaluations).

EA

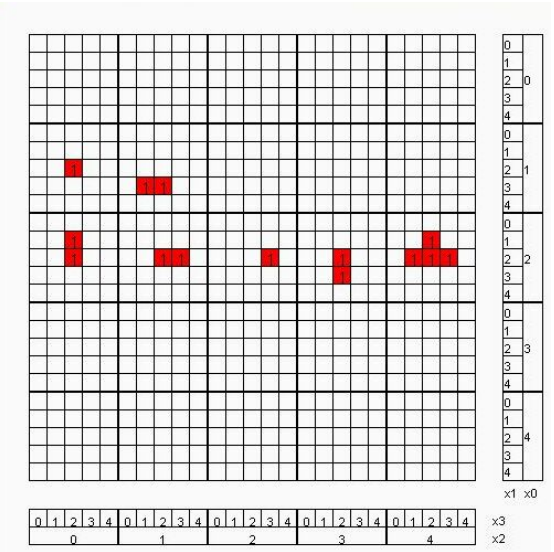


Figure 3-24: EA population in the fifth generation (214 fitness evaluations).

Generation 8: In the eighth generation, EA converged to one of the two solutions shown in Figure 3-25. The total number of fitness function evaluations needed by LEM3 was 250, and by EA was 346. Also, LEM3 found both optima, and EA only one of the two.

LEM3

LEM3 already converged to both optimal solutions after fifth generation, as shown in Figure 3-23.

EA

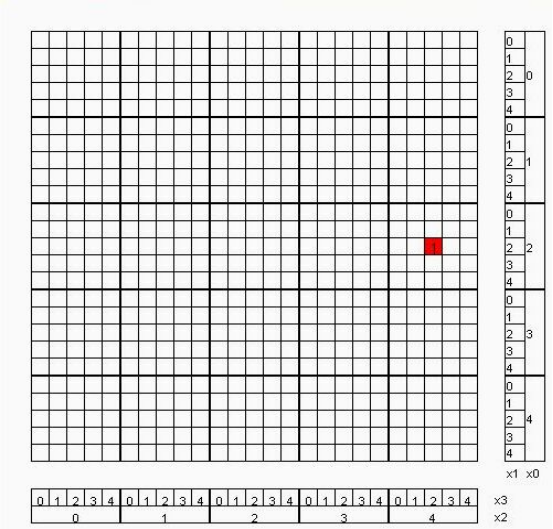


Figure 3-25: One of the two optima found by EA in the eighth generation (346 fitness evaluations).

While the advantage of LEM3 over EA in solving this simple problem (only 4 variables describe the candidate solutions) is relatively large (EA needs about 100 more fitness evaluations than LEM3), a more impressive advantage of LEM3 is seen in problems with larger numbers of variables. This is so because LEM3's advantage grows with the number of variables (Wojtusiak and Michalski, 2005).

3.7 Other Systems Based on the LEM Methodology

The learnable evolution model was introduced by Michalski (1998) and its initial implementation, LEM1, has shown very promising results (e.g. Michalski 1998; 2000; Michalski and Zhang, 1999). This initial implementation has several limitations on methods used and classes of problems it can be applied to. As a learning module LEM1 used the AQ15 learning program, with which it communicated through text files. The most important difference that distinguishes LEM1 from the later LEM systems is that it instantiates only the strongest rule from the learned hypothesis. Because of this fact, the program may be inadequate for multi-modal functions, and can easily got stuck at a local optimum (described by the strongest rule), while more advanced LEM implementations are able to search in parallel local areas near different local optima. Another limitation is that LEM1 can be applied only to numerical data. Shortly after LEM1 was introduced, a more advanced implementation, LEM2, was developed (Cervone, Kaufman, and Michalski, 2000). It is based on the AQ18 learning program (Kaufman and Michalski, 2000b) and uses a more advanced instantiation method, which instantiates all rules from learned hypotheses. It also implements an adaptive discretization method, ANCHOR (Michalski and Cervone, 2001), whose improved version is implemented in LEM3 as described in Chapter 5.

A class of LEM-based systems for heat exchanger optimization has been developed. These include ISHED systems for optimizing evaporators and ISCOD systems for optimizing condensers (e.g. Kaufman and Michalski, 2000a,c; Domanski et al., 2004; Michalski and Kaufman, 2006). These specialized systems combine LEM's learning and

instantiation operators with specialized probing operators that are specifically designed to work with heat exchangers. They also implement methods for handling constraints which either describe physically impossible designs (strict constraints) or expert knowledge about which designs are reasonable (flexible constraints). Based on the ISHED and ISCOD systems Michalski and Kaufman (2006) proposed a general LEMd methodology for optimizing complex systems.

An independent implementation of the learnable evolution model for multi-objective optimization, LEMMO (Jourdan et al., 2005), is based on rules derived from decision trees learned by the C4.5 program (Quinlan, 1993). LEMMO was developed for application to a water quality optimization problem. Decision trees and rules derived from them by C4.5 are significantly limited when compared to these learned by AQ systems (e.g. there is no internal disjunction), and therefore many more rules are usually needed to describe the same concept. Despite these limitations, the authors reported very promising results.

For completeness of this section, it is important to note that LEM is also used as an optimization method in the VINLEN inductive database system (e.g. Kaufman et al. 2007). The system combines a database with a knowledge base and several knowledge generation operators. In the current implementation LEM3 is used as an optimization tool in the system. Its initial population can be loaded from the database, where results of optimization are also stored. Intermediate hypotheses can be stored in its knowledge base in order to provide users with additional information about the optimization problem.

3.8 Related Research on Non-Darwinian Evolutionary Computation

Research on evolutionary computation, which originated in the early 1960s, followed a general Darwinian principle of evolution. Simple operators for creating new candidate solutions are used and selection of candidate solutions into a new population follows the principle of the survival of the fittest. More recently, mostly in the 1990s, a number of advanced evolutionary computation methods were introduced. These methods significantly differ form the original evolutionary computation techniques by use of advanced reasoning and/or machine learning for creation of new candidate solutions, use of advanced methods for selecting new populations, and multistrategy application of different methods and modes of operation on different stages of evolution. In addition to the learnable evolution model discussed in this dissertation, among the best well-known methods from this class are cultural algorithms, estimation of distribution algorithms, and memetic algorithms. The next sections discuss these methods and their relation to LEM, in particular its LEM3 implementation.

3.8.1 Estimation of Distribution Algorithms

Estimation of distribution algorithms (EDAs) use statistical inference and learning to generate distributions of high-performing candidate solutions selected from one population (Mühlenbein and Paaß, 1996; Larrañaga and Lozano, 2002) usually without using a contrast set of low-performing candidate solutions. The most popular statistical inference methods in EDAs include building different variants of Gaussian distributions. There are also a number of implementations that use Bayesian and Gaussian networks as models for representing hypotheses.

This approach is significantly different from the learnable evolution model, which uses symbolic learning to distinguish between high- and low-performing candidate solutions. EDAs also use values of fitness functions only for selecting candidate solutions for learning, while LEM can effectively use the values during learning process (e.g. by learning significance-based descriptions; Wojtusiak, 2004b).

3.8.2 Cultural Algorithms

Another class of methods close in spirit to the learnable evolution model are *cultural algorithms* (Reynolds, 1994; Reynolds and Zhu, 2001; Saleem and Reynolds, 2001; Reynolds and Peng, 2004), which use information about candidate solutions to guide mutation and recombination operators. The cultural algorithms perform a constrained optimization process in which constraints are created in parallel to the evolution process. The constraints, called beliefs, are stored in a belief space that is updated to reflect the fitness profile or the current population. Candidate solutions that are stored in an optimization space are modified (via constrained mutation and recombination) so that they satisfy the beliefs. The belief space is built based on statistical information about candidate solutions, which usually consists of intervals containing the fittest candidate solutions.

3.8.3 Memetic Algorithms

Another form of non-Darwinian evolutionary computation is called *memetic algorithms* (MAs). They use an idea of switching between global search performed by evolutionary

computation and local search in order to improve selected candidate solutions in the population. Most research on memetic algorithms concerns combining evolutionary computation with methods of local search in context of particular applications. Issues investigated in memetic algorithms include selection of appropriate local search methods and their integration with evolutionary search. A good summary of memetic algorithms was prepared in the book edited by Hart, Krasnogor, and Smith (2004).

The idea of switching between different modes of operation is also present in LEM. In LEM3 it is realized by the action profiling function (see Section 3.5.3) which executes several actions such as learning and instantiating, probing, randomizing, and searching locally.

3.8.4 Wise Breeding Genetic Algorithm

An effort to combine ideas from the learnable evolution model and the estimation of distribution algorithms has been made by Llora and Goldberg (2003). The method is in fact a simplified version of LEM, which uses statistical information about a population to instantiate attributes not included in rules. The method uses a fixed bit representation of candidate solutions and passes high-performing and low-performing examples to the ID3 (Quinlan, 1986) decision tree learning program. Rules obtained (from trees) by ID3 consist of information which bits are present in high-performing candidate solutions. For bits not included in the rules the method uses an idea of the *population based incremental learning* (PBIL; Baluja, 1994; Baluja and Caruana, 1995) which uses incrementally updated probabilities of values of bits in the best candidate solutions. Such a method of

selecting values of bits (attributes) not included in rules is equivalent to method implemented in LEM3 that takes values from a probabilistically selected candidate solution with proportion to its fitness. Moreover, LEM3 operates on level of phenotypes, not bits, therefore it is more general. Experiments have shown that PBIL have outperformed standard genetic algorithms in solving some problems, and underperformed in solving others.

3.8.5 Other methods related to LEM

There are several attempts of using learning in evolutionary computation such as adaptive control of crossover (Sebag and Schoenauer, 1994; Sebag, Schoneauer and Ravise, 1997a). Learning rules that are used to prevent new generations from repeating past errors by keeping track of past evolution failures is described in (Ravise and Sabag, 1996; Sebag, Schoenauer and Ravise, 1997b).

An approach that extends the traditional Darwinian approach can be found in the GADO algorithm (Rasheed, 1998). GADO is an evolutionary algorithm developed for engineering problem optimization. It differs from traditional genetic algorithms primarily in the way new candidate solutions are generated. It uses five different crossover operators, three of which are introduced in GADO: line crossover, double line crossover, and guided crossover. However, the algorithm does not apply any generalization operations for guiding the evolution, such as one used in LEM's learning mode.

The STAGE method uses statistical learning (e.g., linear or quadratic regression) to approximate the fitness function. In the area with the predicted best value the method applies a local search method such as hill climbing or simulated annealing (Boyan and Moore, 2000). STAGE can be viewed as a guided method in the sense that learning is used to guide local search as it approximates the fitness function. In contrast to LEM, STAGE does not learn hypotheses distinguishing high- and low-performing individuals.

To conclude this section, it is important to mention that a significant amount of work on optimization has been done in operations research (e.g., Rardin, 1997; Hiller and Lieberman, 2004). Specifically, optimization is present in such fields as planning (e.g., Jensen, Veloso, and Bryant, 2004; Riley and Veloso, 2006) and scheduling (e.g., Kovalov, Ng, and Cheng, 2007). These methods are based on different principles than the learnable evolution model, but many of the ideas they use can potentially enrich LEM3 or future LEM-based systems.

CHAPTER 4 HANDLING CONSTRAINTS

This chapter presents the problem of handling constraints during the evolutionary optimization process, and presents methods for constrained optimization in the learnable evolution model. Section 4.1 gives an introduction and formally defines constrained optimization problems, and Section 4.2 presents an overview of methods applicable to the learnable evolution model described in the literature. A proposed classification of constraints is presented in Section 4.3, and methods for handling selected classes of instantiable constraints, and for handling general constraints, are presented in Sections 4.4 and 4.5, respectively. Sections 4.6 and 4.7 describe special issues concerning flexible constraints and starting with no feasible solutions.

4.1 Introduction and Definitions

Most real world optimization problems are constrained. The constraints may represent physical limitations of objects being optimized, experts' domain knowledge, or simply the user's desire to get results with specific properties. For example, when designing the shape of one-gallon containers, their dimensions may vary, but the volume has to be constant, or when designing a network it is important that all considered nodes are connected to the network.

Optimization problems to which LEM is applied involve finding optima (either minima or maxima) of a given fitness function:

$$f(X_1,..,X_n): E \rightarrow \mathcal{R} \qquad (4\text{-}1)$$

where $E = D_1 \times D_2 \times \ldots D_n$ is a Cartesian product of domains of attributes $X_1, \ldots X_n$. It is often the case that not all points in E represent feasible solutions. In addition to domains D_i, i=1..n, a constrained optimization problem specifies sets of constraints limiting the representation space. The constraints define a set of feasible solutions, a subset of E. In this dissertation, *constraints* and *constrained optimization* problems are defined as:

A *constraint* is a condition (selector) that is used to define feasible solutions.

A *constrained optimization problem* in the learnable evolution model is to find a vector $x^* = (x_1^*, \ldots x_k^*)$ that is an optimum of a fitness function in the form (4-1) that satisfy constraints expressed in attributional calculus.

The constraints can be given, for example, in disjunctive normal form (DNF). For example, suppose that E is a Cartesian product of domains of three attributes: Color with domain {red, green, blue}, Length with domain [0 ..10], and Width with domain [0..10], and f: E → $\mathcal{R}$ is a fitness function. Example constraints for such a problem are:

[Color=red] & [Length > 4] ∨ [Color=blue ∨ green] & [Length+Width < 10] (4-2)

which can be paraphrased as "feasible solutions are those for which Color is red and Length is greater than 4 or Color is blue or green and the sum of Length and Width is less than 10."

These definitions are more general than those often found in literature (e.g. Michalewicz and Schoenauer, 1996; Liang et al., 2005) where two types of constraints are recognized, namely *equality*, and *inequality* constraints. Attributional calculus allows using functions in conditions, thus constrained problems available in literature can be immediately transformed into the form used in this dissertation. It also means that a constraint may involve an external simulator, for example, as in (4-3) where sim is a function that returns a Boolean value based on the result of the simulator execution.

$$[\text{Color}=\text{blue} \vee \text{green}] \;\&\; [\text{sim}(\text{Color}, \text{Length}, \text{Width})] \qquad (4\text{-}3)$$

An important question is if it is possible by the learnable evolution model to generate candidate solutions, through hypotheses generation and instantiation, which satisfy constraints. The following theorem guarantees that when the learning mode is applied in LEM it is possible to generate feasible candidate solutions.

Theorem 1: If all high-performing candidate solutions provided to AQ learning program are feasible, then each rule in a learned hypothesis can be instantiated with feasible candidate solutions.

Proof: Each rule covers at least one high-performing candidate solution. Since the high-performing candidate solutions are feasible, the candidate solution is also feasible.

The above theorem makes the assumption that all constraints are strict, meaning that they need to be satisfied, in contrast to flexible constraints, which do not necessarily need to be satisfied. A general version of the theorem is presented in Section 4.6. Also, the

theorem guarantees the existence of feasible solutions, but it does not guarantee the existence of any new feasible solutions, i.e. solutions that are not included in the previous population. It is the role of a learning program to inductively generalize provided examples. Learned hypotheses cover previously unknown parts of the space that are likely to contain new feasible solutions.

4.2 Summary of Methods of Handling Constrains

The literature includes numerous papers addressing the problem of handling constrained optimization problems using evolutionary computation. Over the past decades a number of methods have been proposed and applied in different evolutionary computation methods. A good description of the constraint handling techniques in evolutionary computation is presented in the book edited by Back, Fogel and Michalewicz (2000; Vol. 2, Part 2). The following sections present a brief summary of different methods of handling constraints and their relation to methods used in the learnable evolution model.

4.2.1 Penalty Functions

Probably the best well known and the most popular method of handling constraints is to penalize infeasible candidate solutions by modifying their fitness values. Suppose that the optimization problem is to find a minimum of the fitness function f: E → $\mathcal{R}$. The penalty function methods reformulate the optimization problem to the problem of finding an optimum of the function F: E → $\mathcal{R}$ given by (4-4).

$$F(x) = f(x) + p(x) \tag{4-4}$$

p(x) is a penalty function equal zero for feasible candidate solutions and positive for infeasible candidate solutions. Similarly for maximization problems, the penalty function is subtracted instead of being added. Penalty functions may reflect number of violated constraints, distance to the feasible area, etc. Values of penalty functions are usually significantly larger than values of the corresponding fitness functions. In the extreme case, the value equals infinity in so called death penalty methods.

In the learnable evolution model, constrained optimization problems can be directly handled by penalty functions without any additional changes to the program. This requires only modifying the fitness function definition, which is treated by LEM as a black box. The LEM algorithm will then correctly execute in all modes, even though the population may be a mixture of feasible and infeasible candidate solutions. Also, it is important to note that the Theorem 1 applies to the penalty function methods as well.

4.2.2 Constraint Preserving Operators

An important class of methods of handling constrained optimization problems is based on constraint preserving operators. The application of such an operator guarantees feasibility of newly generated candidate solutions. These operators, however, need to be designed for specific application domains and need to incorporate problem-oriented background knowledge. The constraint preserving operators have been designed for a number of specific problems such as numeric optimization with linear constraints (Michalewicz, 2000) and the traveling salesman problem (Whitley et al., 1989).

In the learnable evolution model, methods of handling instantiable constraints are classified to this category. Given a learned hypothesis and a set of constraints, the instantiation process generates new candidate solutions that satisfy both the hypothesis and constraints (see Section 4.4).

ISHED, an implementation of the learnable evolution model tailored to heat exchanger design, implements change operators that preserve constraints in both learning and probing modes (e.g. Kaufman and Michalski, 2000a). In the learning mode, the program uses heuristics to instantiate hypotheses in such a way that constraints are never violated.

4.2.3 Rejection Methods

Rejection methods represent the simplest and most natural approach to the problem of handling constraints. If application of an operator generates an infeasible candidate solution, simply reject the solution and try again. This process is illustrated by the following pseudocode.

```
Loop while candidate solution is infeasible and not exceeded maximum
number of trials
      Generate new candidate solution
      Test feasibility of the solution
```

Figure 4-1: Pseudocode of rejection method for handling constraints.

The rejection methods assume that the operator is not deterministic, meaning that each time it will generate a different candidate solution with some probability. In the learnable evolution model, the rejection method is implemented for handling general constraints, and is described in detail in Section 4.5. Given a learned hypothesis, the instantiation algorithm is applied until a feasible candidate solution is generated. The applied operators may use heuristics in order to minimize the probability of generating infeasible solutions. For example, one of the methods described in Section 4.5 applies inductive learning to find an approximation of the feasible space. The approximation is used to reject candidate solutions without complete evaluation of constraints. Theorem 1 applies to the rejection methods, meaning that it is always possible to generate feasible solutions if the number of trials is sufficiently large.

The ISHED system in probing mode uses the rejection strategy; it tries to apply one of several available operators until a fully feasible solution is found. If the total number of trials exceeds a user-defined threshold, no new designs are generated. For details of this method, see for example Kaufman and Michalski (2000a,c).

4.2.4 Representation Change

Methods based on a change of representation use the idea that a candidate solution may be encoded in such a representation space in which only feasible solutions are represented. For fitness evaluation, each candidate solution is decoded into the original representation space, thus this class of methods is also known as *decoders*. The main

disadvantage of this class of methods is that they are problem-oriented and require a precise definition of the representation space. It is important that all feasible solutions from the original representation space are represented in the modified space, or at least the best solutions are represented. Otherwise, the global solution to the optimization problem may be missed.

In the literature there are several known applications of the representation change methods to specific problems, most commonly to combinatorial optimization problems. For example, a method of solving the traveling salesman problem using representation change was presented by Grefenstette et al. (1985). Other applications include scheduling and partitioning (e.g. Syswerda, 1991; Jones and Beltramo, 1991).

4.2.5 Repair Methods

A result of the application of a genetic operator may be an infeasible candidate solution. Instead of rejecting such a candidate solution, it may be repaired, or transformed into a feasible candidate solution. In the literature, two possibilities are investigated: to use the transformed candidate solution for the evaluation only, or to replace the infeasible candidate solution with its new feasible version. Similarly to the representation change, the repair methods require problem-specific solutions. For each particular application, a repair algorithm needs to be constructed, and more importantly there are no general heuristics that could help in constructing such an algorithm (Michalewicz, 2000). In the literature these methods are mostly known in combinatorial optimization, e.g. in the

knapsack problem (Michalewicz, 1996). The repair methods are also used in the Genocop III system (Michalewicz and Nazhiyath, 1995) that utilizes two populations of so-called search points and reference points. The latter consist of only feasible candidate solutions, which are used as reference to repair possibly infeasible candidate solutions from the search population.

4.2.6 Multi-objective Optimization Methods

A number of authors considered the problem of handling constraints by reformulating the initial problem into its multi-objective equivalent (e.g. Surry, Radcliffe, and Boyd, 1995; Deb, 2001). The general idea behind this approach is to convert constraints to additional objectives in the multi-objective optimization. Let f: E $\rightarrow \mathcal{R}$ be the fitness function and C_1, …, C_n be the constraints. One possible method is to use the multi-objective optimization to find an optimum of (f(x), #C(x)) where #C(x) denotes the number of unsatisfied constraints in the minimization problems. Another possibility is to treat each constraint as a separate objective, whose value indicates the degree to which the constraint is violated.

A result of applying multi-objective optimization methods is usually a Pareto front consisting on several possible solutions. In the case of using of the multi-objective optimization to constraints, the most interesting solutions are ones where #C(x) = 0.

4.3 Classification of Constraints

It is usually the case that methods more universal and applicable to many problems are less efficient than methods specialized to do particular tasks. This is also the case for the methods for handling constrained optimization problems. Formally, the problem of constraint satisfaction (CSP) that is equivalent to instantiation in LEM is known to be NP-complete (e.g. Mackworth, 1977). The presented methodology distinguishes between different types of constraints and methods for handling them.

The following is a classification of constraints into four types, from the easiest to the most difficult to handle by LEM's instantiation process. Detailed algorithms used for handling selected constraints of these types are presented in Sections 4.4 and 4.5.

To illustrate the proposed types of constraints let

E = D(Length) x D(Width) x D(Height) x D(Color) x D(Background Color)

where Length, Width, Height, Color, and Background Color are attributes and D(X) denotes the domain of the attribute X. The optimization problem is to find an optimum of a fitness function f: E→$\mathcal{R}$ given a set of constraints C.

Type 1: Constraints are defined using conditions that include attributes from the original representation space, that is, in the form (4-5). Here, ATT is an attribute from the original problem definition, VALS is a set of values from D(ATT), and rel is a relation that applies to ATT and VALS.

[ATT rel VALS] (4-5)

Example:

[Length=1..5] & [Color=red] ∨ [Length=2..7] & [Height<4]

This type of constraints is the simplest to handle. It can be done by intersecting conditions in learned hypotheses with the constraints. According to Theorem 1, such an intersection is always non-empty whenever feasible high-performing candidate solutions were provided to the learning program. No additional changes to LEM's instantiation algorithms are needed.

Type 2: Constraints are defined using conditions in the form (4-6), where ATT is an attribute from the original problem definition (representation space), EXPR is an expression that may include only constants and attributes from the original representation space, excluding ATT, and rel is a relation applicable to ATT and EXPR.

[ATT rel EXPR] (4-6)

Example:

[Length = 5 + Width * Height]

These constraints can also be instantiated by intersecting them with hypotheses describing high-performing candidate solutions. Once the expression on the right side of the condition is evaluated, such a condition takes the form [ATT rel VAL], which is equivalent to a Type 1 constraint defined in the original representation space.

Type 3: Ordered conjunctions of Type 2 conditions in the form (4-7), where ATT_i for i=1..n are attributes from the original representation space; rel_i are relations (e.g., =, <, >); $EXPR_1$ is an expression that does not include attributes ATT_i for i=1..n, $EXPR_2$ is an expression that does not include ATT_2, .., ATT_n, $EXPR_n$ is an expression that does not include ATT_n.

$$[ATT_1\ rel_1\ EXPR_1]\ \&\ [ATT_2\ rel_2\ EXPR_2]\ \&\ \ldots\ \&\ [ATTi_n\ rel_n\ EXPR_n] \qquad (4\text{-}7)$$

Example:

$$[X_7 = 5 - X_{32} - X_{12}]\ \&\ [X_5 \leq X_8 - 6]\ \&\ [X_{10} = X_7 - X_{52} + X1]$$

Constraints of Type 3 can be sequentially evaluated. The first constraint does not depend on any other constraint, thus it can be evaluated independently. The second constraint may depend only on the first constraint, which is already evaluated. In general a constraint C_i may depend only on constraints C_j for $j < i$. This prevents interdependency between constraints, so they can be efficiently solved in a sequence.

Type 4: General constraints which cannot be directly instantiated or an efficient instantiation method is not available at the time. Among methods for handling general constraints are those known from the field of evolutionary computation (e.g. penalty functions, repair algorithms; see Section 4.2) and methods designed especially for the LEM's learning mode. The latter include methods that approximate the feasible area of the optimization space, trim learned rules in order to minimize their intersection with the infeasible area, and use infeasible solutions as contrast sets for learning. These methods are described in detail in Section 4.5.

4.4 Instantiable Constraints

The instantiable constraints are those for which it is possible to apply the LEM's learning mode in such a way that it preserves the constraints. Each newly generated candidate solution is guaranteed to satisfy constraints. There are several different forms of such constraints classified from the LEM's perspective in three different types. This section presents methods of handling constraints of Types 1-3. Although these types of constraints may seem not to be practically important, as they constitute a very narrow class, they are important for instantiating hypotheses learned in modified representation spaces presented in Chapter 5. The AQ21 learning module equipped with constructive induction is able to learn hypotheses in modified representation spaces that are equivalent to Type 2 and 3 constraints.

Suppose that the problem is to find an optimum (either minimum or maximum) of a function f: E $\rightarrow \mathcal{R}$ given set of constraints $C = C_1 \vee C_2 \vee \ldots \vee C_n$ in disjunctive normal form, where $C_i = C_i^1 \& C_i^2 \& \ldots \& C_i^{ki}$ for i=1..n and C_i are of Types 1 – 3. Suppose also that the program is at a given stage of evolution, meaning that there is a population P of feasible candidate solutions and learning mode has been selected as an innovation operator. Learning mode is then applied in the steps presented in Figure 4-2 as an extension of one presented in Figure 3-7.

1. Select H- and L-groups of high- and low-performing candidate solutions, respectively
2. Apply rule learning to generate a hypothesis, H, characterizing H-group against L-group
3. Intersect the learned hypothesis H with constraints C to create a *target constrained hypothesis* (TCH)
4. Reduce the size of TCH based on coverage
5. Instantiate the TCH to create new candidate solutions

Figure 4-2: Steps of LEM's learning mode with instantiable constraints.

The selection of high-performing and low-performing in candidate solutions into the H-group and L-group, respectively, is done using one of the standard methods available in the learnable evolution model, namely population-based selection or fitness-based selection (Section 3.2). Application of a rule learning program to the sets of high- and low-performing solutions also does not differ from standard (unconstrained) LEM. Because high-performing candidate solutions are feasible, Theorem 1 guarantees that it is possible to generate feasible candidate solutions from learned hypotheses. The following methods assume that the learned hypotheses are in the form of attributional rulesets, which can be learned by programs from the AQ family, for example LEM3's AQ21.

Hypotheses which are attributional rulesets, as generated by AQ learning programs, usually consist of several attributional rules needed to describe the group of high-performing candidate solutions (4-8).

$$H = R_1 \vee R_2 \vee \ldots \vee R_m \tag{4-8}$$

Intersection of the hypothesis H with constraints C gives (4-9).

$$TCH = (R_1 \vee R_2 \vee \ldots \vee R_m) \,\&\, (C_1 \vee C_2 \vee \ldots C_n) \tag{4-9}$$

After transforming (4-9) into disjunctive normal form the TCH becomes (4-10).

$$TCH = (R_1\&C_1 \vee R_1\&C_2 \vee \ldots \vee R_1\&C_n \vee \ldots \vee R_m\&C_1 \vee \ldots \vee R_m\&C_n) \quad (4\text{-}10)$$

The rules R_iC_j are called *target constrained rules* (TCR), as they represent rules intersected with constraints. The total number of the target constrained rules after intersection is *m x n* where *m* is the number of rules in H and *n* is the number of conjunctions of constraints in C. Some of the rules may be eliminated by applying absorption laws.

It is often the case that many of these constrained rules are empty, meaning that the intersection R_iC_j cannot be satisfied. For such constrained rules there is no point in even starting the instantiation, because it cannot be successfully finished. For example, let H = [x = 1..2] ∨ [y = 1..2] and C = [x > 3] ∨ [y > 3]. The intersection of H with C is TCH = [x=1..2] & [x>3] ∨ [x=1..2] & [y > 3] ∨ [y = 1..2] & [x > 3] ∨ [y = 1..2] & [y >3], in which the first and the last target constrained rules cannot be satisfied. Finally, TCH consists of two rules with CONSEQUENT parts [x=1..2] & [y > 3] and [y = 1..2] & [x > 3]. In many situations it may be difficult to determine whether an intersection of a rule with a constraint is empty or not, especially when constraints are defined using expressions, not just values as in the above simple example. This observation is the reason for using nonzero positive coverage of target constrained rules as a basis for choosing them for the instantiation process. For each constrained rule the method computes its positive coverage (number of high-performing candidate solutions that satisfy the rule). All rules for which the coverage is empty are removed from the target constrained hypothesis. This guarantees that each rule passed to the instantiation module can be instantiated. The instantiation module, described in detail in Section 4.4.1,

sequentially assigns values of attributes in a created candidate solution. Doing this for many attributes involves computing values of expressions that define constraints.

4.4.1 Instantiation Algorithm

Creation of new candidate solutions in LEM's learning mode is realized by the instantiation of learned hypotheses. For constrained optimization problems, the instantiation takes the *target constrained hypothesis* (TCH) as an input and produces a set of feasible candidate solutions that satisfy the given TCH. Because the TCH is created by intersecting the learned hypothesis with constraints, the newly generated candidate solutions will be feasible and likely high-performing. Also, because the TCH is reduced to rules that cover positive examples (high performing and feasible candidate solutions), each rule is instantiable.

The pseudocode in Figure 4-3 describes an algorithm for generating new candidate solutions from a given target constrained hypothesis. It is assumed that all constraints are of Types 1-3, meaning that they can be sequentially evaluated while generating a new candidate solution. Whenever a constraint cannot be satisfied because of previous selections of attribute values, a simple backtracking algorithm is used.

In the algorithm, the number of new candidate solutions generated for each constrained rule is proportional to its positive coverage (number of covered high-performing solutions). At this point the set of already created new candidate solutions is empty. It is important to correctly order conditions (attributes) in the current constrained rule, so all

attributes needed to instantiate an attribute are already instantiated. After the list of attributes to be instantiated is prepared, the algorithm creates new candidate solutions until their total number reaches the desired level, or until the total number of unsuccessful conditions' instantiations exceeds a given threshold.

```
For each rule in TCH
   Compute the number of candidate solutions to be generated
   New Solutions = NULL
   Compute order of attributes based on constraints
   While size of New Solutions < number of solutions to be generated
      For all attributes in the computed order
         Compute all expressions with the attribute on left side
         Compute condition for the attribute
         If condition is empty
            If the total number of tries exceeds a threshold
               Stop instantiating the current rule
            Jump to one of attributes before based on backtracking
         Else
            Instantiate the condition
      Add the candidate solution to New Solutions
```

Figure 4-3: Top level instantiation algorithm for constrained rules.

Each new candidate solution is created by selecting values of attributes in the order defined in the previous step. If for an attribute being considered there is no condition in the instantiated constrained rule, the program uses one of the methods described in Chapter 3. This includes taking a value from a randomly selected high-performing candidate solution that satisfies the rule, or selecting a value from the entire attribute's domain. This situation happens if the attribute is included neither in the original rule nor in the constraints. For attributes included in the constrained rule, the final condition to be instantiated is computed based on conditions from the constraints and the hypothesis. For example, the hypothesis may include a rule with condition [Height $\geq$ 34] and the intersected constraints may include conditions [Height $\geq$ Width + Length – 2] and

[Height ≤ 50]. To select a value of the attribute Height, it is necessary to compute conjunction of the three conditions: [Height ≥ 34] & [Height ≥ Width + Length – 2] & [Height ≤ 50]. Because it is assumed that all constraints are of Types 1-3, the values of attributes Length and Width will already be selected, and therefore all expressions in all conditions can be evaluated. For example if Width = 30 and Length = 10, we have [Height ≥ 34] & [Height ≥ 38] & [Height ≤ 50], which can be reformulated as [Height=38..50]. The final condition consists of a range that can be instantiated using one of the methods discussed in Chapter 3.

An important question arises what should be done in a situation when a constraint of Type 2-3 (defined by an expression) cannot be satisfied because of the choice of previously selected values of attributes needed to evaluate a condition. Suppose that in the previous example the new candidate solution was assigned value 31 for the attribute Width and value 25 for the attribute Length (the attributes were instantiated in this order). In such a case conditions for the attribute Height are [Height ≥ 34] & [Height ≥ 54] & [Height ≤ 50] which is an empty interval and obviously cannot be satisfied. To solve this problem the program needs to reinstantiate the attribute Length and compute conditions for Height again. Because values of attributes that satisfy conditions are assigned randomly (according to some distribution) it may happen that the condition is not satisfied again and the operation needs to be repeated. This process is continued no longer than a specified number of times, when the program tries to reinstantiate a previous attribute (in this case Width) also for a specified number of times. This process repeats until the condition is satisfied or a total number of tries (usually very large) is

exceeded and the rule is ignored. According to Theorem 1, it is always possible to find a set of values that satisfy all conditions, but it may take a very large number of trials.

The backtracking-like algorithm described above is presented in Figure 4-4. The notation assumes that attributes X_1 .. X_{i-1} are successfully instantiated, and X_i is the attribute currently being instantiated.

```
If the condition for attribute Xi is empty
   If the count of trials for the attribute Xj is below threshold
      Reinstantiate Xj and all attributes Xj+1..Xi
      Increase counter
   Else
      Decrement j (if j > 1)
      Counter = 0
Else
      Set j = i
      Counter = 0
```

Figure 4-4: Backtracking-like method for resolving constraints of Types 2-3.

The algorithm does not invoke full backtracking, but uses a counter for a group of attributes being reinstantiated. It was mentioned above that according to Theorem 1, there is always a feasible solution that satisfies all of the conditions. It can be achieved if the threshold for the total number of trials is set to infinity. The threshold is added to increase the efficiency of the entire instantiation function in LEM.

4.4.2 An Example Execution of the Instantiation Algorithm

A simple optimization problem can be used to illustrate the execution of the instantiation algorithm for constraints of Types 1-3. Let c1: $E \rightarrow \mathcal{R}$ be a generalized n-dimensional sphere function $c1(X_1,\ldots,X_n) = \sum X_i^2$, $E=[-n, n]^n$, and constraints are given by (4-11).

$$C = \Pi(X_i \geq X_{i+1} + 1),\ i=1..n-1 \tag{4-11}$$

The function has one global minimum $S^0 = ((n-1)/2, (n-1)/2-1, \ldots, -(n-1)/2+1, -(n-1)/2)$. For example for n=5 the solution is (2, 1, 0, -1, -2) and for n=4 the solution is (3/2, 1/2, -1/2, -3/2). In this example no discretization is used.

Suppose that LEM is applied to the 5-dimensional c1 function and at a given step of evolution its learning module discovered the rule (4-12) characterizing selected high performing candidate solutions.

$$[\text{S is high performing}] \Leftarrow [X_1=2..4] \ \&\ [X_3=-1..1] \ \&\ [X_5=-4..2] \tag{4-12}$$

The learned hypothesis may consist of several of such rules, but for simplicity this section focuses on problem of instantiating only one rule. The rest of this section presents a step-by-step illustration of the instantiation of one candidate solution.

Step 1: Intersection of the learned rule with constraints.

At this step the hypothesis (4-8) is intersected with constraints (4-11) resulting a constrained target rule given by (4-13).

$$\begin{gathered}[X_1=2..4] \ \&\ [X_1 \geq X_2+1] \ \&\ [X_2 \geq X_3+1] \ \&\ [X_3=-1..4] \ \& \\ [X_3 \geq X_4+1] \ \&\ [X_4 \geq X_5+1] \ \&\ [X_5=-4..2]\end{gathered} \tag{4-13}$$

Step 2: Order attributes (conditions).

Type 2 and 3 constraints require evaluation of expressions and the outcome of the evaluation depends on previously selected values. Here, X_4 depends on X_5, X_3 depends on X_4, X_2 depends on X_3, and finally X_1 depends on X_2. Thus, the only allowed order of

the attributes to be instantiated is X_5, X_4, X_3, X_2, and X_1. The ordered target constrained rule is given by (4-14).

$$[X_5=-4..2] \,\&\, [X_4 \geq X_5+1] \,\&\, [X_3=-1..4] \,\&\, [X_3 \geq X_4+1] \,\&\, [X_2 \geq X_3+1] \,\&\, [X_1=2..4] \,\&\, [X_1 \geq X_2+1] \quad (4\text{-}14)$$

Please note that for many optimization problems in general, many different possible orders of the attributes may exist.

Step 3: Create a new candidate solution and select its attribute values.

In this step values of attributes are selected according to the given target constrained rule in the order determined in Step 2. First an empty candidate solution is generated (4-15). The "*" indicates that no value is selected for a given attribute.

$$(*, *, *, *, *) \quad (4\text{-}15)$$

Because X_5 is the first attribute to be instantiated a value is selected satisfying the condition $[X_5=-4..2]$. Here, it is assumed that the uniform distribution is used to instantiate conditions.

$$(*, *, *, *, -1.75) \quad (4\text{-}16)$$

The next attribute to be instantiated is X_4, whose value needs to satisfy condition $[X_4 \geq X_5+1]$. Because the value of X_5 was selected as -1.75, the condition takes the form $[X_4 \geq -0.75]$, and is also limited by the upper bound of the domain of attribute X_4. Suppose finally that the randomly selected value satisfying the condition [x4=-0.75..5] is 2.1, which gives (4-17).

$$(*, *, *, 2.1, -1.75) \quad (4\text{-}17)$$

The next attribute to be instantiated is X_3, whose value needs to satisfy the conditions $[X_3=-1..4]$ & $[X_3 \geq X_4+1]$. After evaluating the conditions, we have $[X_3=-1..4]$ & $[X_3 \geq 3.1]$, which is equivalent to $[X_3=3.1..4]$. A random value satisfying the condition is selected, say 3.7, resulting in the instantiated candidate solution shown in (4-18).

$$(*, *, 3.7, 2.1, -1.75) \quad (4\text{-}18)$$

The next attribute to be instantiated is X_2, whose value needs to satisfy condition $[X_2 \geq X_3+1]$, which takes the form $[X_2 \geq 4.7]$. Because the upper bound of the domain of X_2 is 5, the condition takes the form $[X_2=4.7..5]$, which can give a random value 4.9, and the candidate solution takes the form (4-19).

$$(*, 4.9, 3.7, 2.1, -1.75) \quad (4\text{-}19)$$

The next attribute to be instantiated is X_1, whose value needs to satisfy conditions $[X_1=2..4]$ & $[X_1 \geq X_2+1]$. After evaluation, the conditions are $[X_1=2..4]$ & $[X_1 \geq 5.9]$, which cannot be satisfied. Using the backtracking method, the attribute X_2 is reinstantiated to, say, 4.8 which satisfies the condition $[X_2=4.7..5]$, and gives (4-20).

$$(*, 4.8, 3.7, 2.1, -1.75) \quad (4\text{-}20)$$

Again, X_1 needs to satisfy conditions $[X_1=2..4]$ & $[X_1 \geq X_2+1]$, which are $[X_1=2..4]$ & $[X_1 \geq 5.8]$, and cannot be satisfied. Suppose now for simplicity that the threshold for backtracking is set to 1. The program reinstantiates the attribute X_3 to satisfy condition $[X_3=3.1..4]$. If the value 3.15 is selected, we have the new candidate solution in form:

$$(*, *, 3.15, 2.1, -1.75) \quad (4\text{-}21)$$

Now, X_2 needs to satisfy condition $[X_2=4.15..5]$ resulting in (4-22).

$$(*, *, 3.15, 2.1, -1.75) \quad (4\text{-}22)$$

Suppose that after selection of value of X_2 the solution takes form (4-23). The conditions for attribute X_1, which are $[X_1=2..4]$ & $[X_1\geq5.22]$ still cannot be satisfied.

$$(*, 4.22, 3.15, 2.1, -1.75) \quad (4\text{-}23)$$

The backtracking threshold is met again and the next attribute to be reinstantiated is X_4, which needs to satisfy condition $[X_4\geq-0.75]$. Suppose that the value -0.5 is selected, giving (4-24).

$$(*, *, *, -0.5, -1.75) \quad (4\text{-}24)$$

Suppose now that instantiation of attributes X_3 and X_2 result in (4-25) and (4-26), respectively.

$$(*, *, 0.7, -0.5, -1.75) \quad (4\text{-}25)$$

$$(*, 2.8, 0.7, -0.5, -1.75) \quad (4\text{-}26)$$

Now, X_1 needs to satisfy conditions $[X_1=2..4]$ & $[X_1\geq3.8]$ which is $[X_1=3.8..4]$. If the value of X_1 is selected as 3.91, the new candidate solution takes the form (4-27) which satisfies both the learned rule and the problem constraints.

$$(3.91, 2.8, 0.7, -0.5, -1.75) \quad (4\text{-}27)$$

The new candidate solution is added to the list of new candidate solutions, and its fitness value will be later evaluated. In the presented example, the selected value $X_4=2.1$ caused an eventual problem when instantiating attribute X_1, which was three attributes later.

4.5 General Constraints

The previous sections discussed a specific type of instantiable constraints, that is, constraints for which there are defined constraint-preserving instantiation methods. In this section it is assumed that constraints are given as a function (4-28),

$$c: E \rightarrow \{true, false\} \tag{4-28}$$

where c(s) = true if a candidate solution s is feasible (satisfies all constraints) and c(s) = false otherwise (at least one constraint is violated). This is equivalent to the attributional condition (4-29) that consists of one selector which includes the function c.

$$[\, c\, (\, X\,)\,] \tag{4-29}$$

Because of this assumption, the methods described in this section are not provided with any prior knowledge about the structure of the constraints; they can only check whether they are satisfied. They are also not provided with the degree to which constraints are violated. The degree of violation of constraints is, however, used later in methods that allow starting with no feasible solutions.

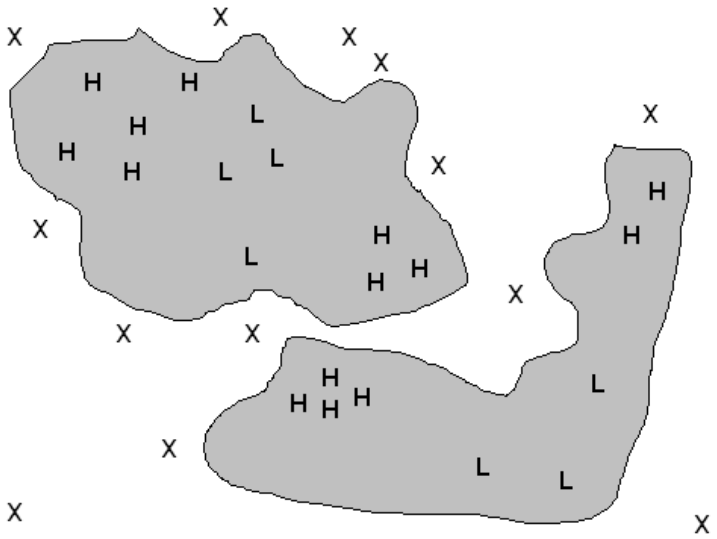
When handling the general constraints, rejection-like methods briefly discussed in Section 4.2.3 can be used. These methods led to two important considerations: (1) the number of fitness evaluations needed to achieve a desired solution, and (2) the number of infeasible candidate solutions generated. Suppose that the fitness evaluation of a candidate solution takes in average t_f time units and the evaluation of constraints takes in average t_c time units. Let FE be the total number of fitness evaluations needed to find a solution and CE be the total number of constraint evaluations needed to find the solution. The total execution time of LEM is given by (4-30),

$$T = FE * t_f + CE * t_c + R \quad (4\text{-}30)$$

where R is the total time of all additional operations such as hypothesis formulation, instantiation, population selection etc. For any problem for which the fitness function evaluation is non-trivial, the factor FE * t_f is greater than R, sometimes very significantly (e.g. Wojtusiak and Michalski, 2005; 2006). The same applies to evaluation of constraints, which may take more time than R. By keeping FE and CE as low as possible, the total time T can be minimized. However, keeping CE low may cause FE to increase, as stricter instantiation which minimizes CE may lead to lack of diversity and improper exploration of the space. It eventually may cause increases of FE and T. Prior knowledge of the evaluation times t_f and t_c may lead to a proper selection of a constraint handling method.

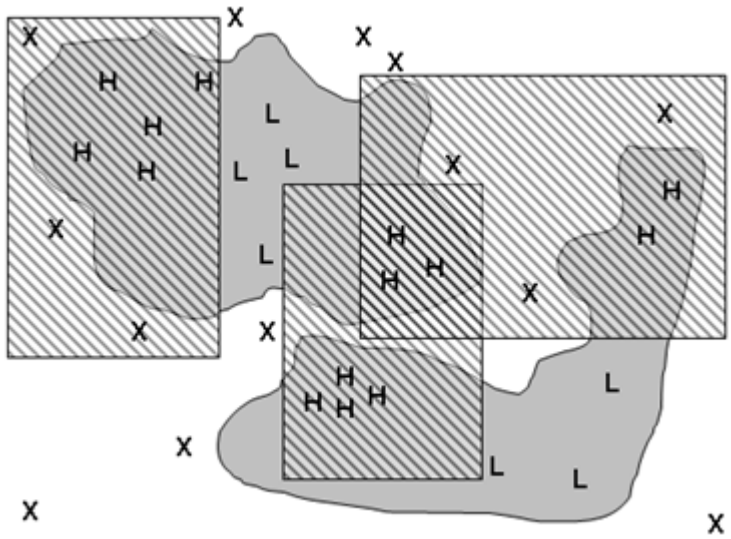
Three methods of handling general type constraints are presented. They are applicable to LEM's learning mode only. The methods are illustrated using a simple example. Initial results from testing the performance of early versions of these methods are presented by Wojtusiak (2006) and further investigated in Chapter 6. Suppose that an optimization problem is defined in a two-dimensional representation space, as illustrated in Figure 4-5. Candidate solutions on the plot are marked H (high-performing), L (low-performing) and X (infeasible), and the shaded area represents the feasible region. Given sets of positive examples (H-group) and negative examples (L-group) the learning program may generate rules characterizing the H-group illustrated in Figure 4-6. The rules are complete and consistent with regard to training data (they cover all positive and no negative examples), but also cover a number of infeasible candidate solutions, and large portions of the

infeasible region. When instantiating these rules, the program may generate many infeasible solutions that would have to be rejected. In real world optimization problems, evaluation of constraints may be a very time-consuming process, sometimes as time-consuming as evaluation of the fitness function. For example, one of the constraints used in the application to finding the best discretization, presented in Chapter 7, requires running an external program. Thus, the presented methods of handling constraints are designed to minimize the number of infeasible solutions generated during the optimization process and at the same time not increase the total number of fitness function evaluations (the evolution length).



H – high-performing candidate solution, L – low-performing candidate solution,
X – infeasible candidate solution, gray area – feasible space

Figure 4-5: Feasible and infeasible candidate solutions in the example problem.

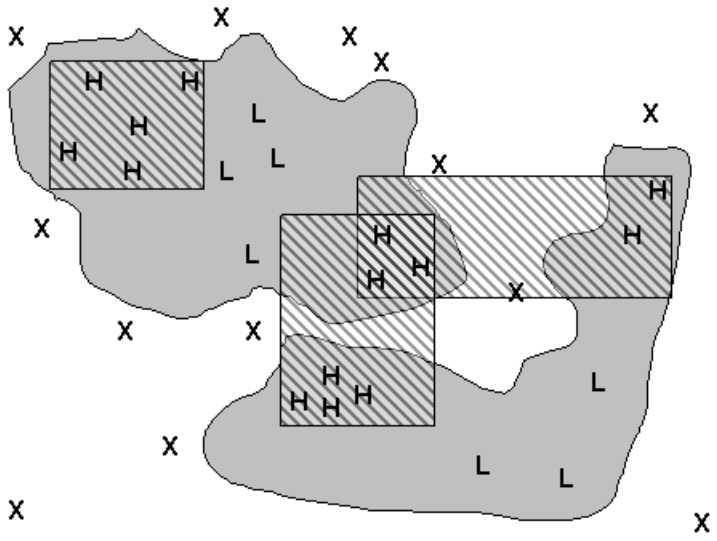


H – high-performing candidate solution, L – low-performing candidate solution,
X – infeasible candidate solution, gray area – feasible space, striped rectangles – rules

Figure 4-6: Example rules discriminating high- and low-performing candidate solutions.

4.5.1 Trimming of Rules

The AQ21 learning program learns rules with controllable levels of generality. Figure 4-6 shows rules that cover a much larger area than is needed to cover the high-performing examples. The rules also cover large portions of the infeasible region. The first presented method of handling general constraints in the learnable evolution model trims the learned rules, so they do not extend far beyond the high-performing examples. As shown in the Figure 4-7, the trimmed rules cover a significantly smaller part of infeasible area, but large portions of the feasible region are also not covered, thus some solutions may be missed.



H – high-performing candidate solution, L – low-performing candidate solution,
X – infeasible candidate solution, gray area – feasible space, striped rectangles – rules

Figure 4-7: Trimmed rules for the example problem.

It can be also noted that the rule in the right part of the diagram covers a large portion of the infeasible area, because the program does not have any information that the area is infeasible. The problem arises when the feasible area consists of disjoint parts, or in

general, it cannot be precisely covered by attributional rules that can be interpreted as hyperrectangles.

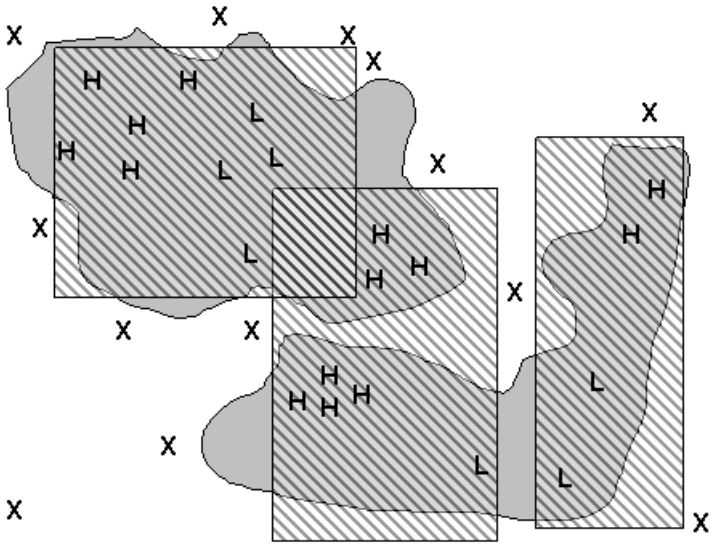
4.5.2 Learning an Approximation of the Feasible Area

The idea behind this method is to learn an approximation of the feasible area in parallel to the evolutionary optimization process. Thus, in addition to the achieved optima, the program reports the approximation of the feasible space, which can be useful for domain experts. The method applies the AQ21 learning program to sets of feasible and infeasible solutions in order to learn a general description of the feasible area.

Let S_f be set of all feasible candidate solutions, and S_n be set of infeasible solutions created during the evolutionary optimization process in LEM. Using S_f as the set of positive and S_n as the set of negative examples, the method learns an approximation of the feasible region. Because both sets are growing during the evolution process, the approximation is becoming more accurate. Instead of checking the newly created candidate solutions against the original constraints, which may be time-consuming (see application to automatic discretization in Chapter 7), the candidate solutions are first checked against the approximation, which is a very fast operation. Only solutions that are included in the approximation are checked against the original constraints.

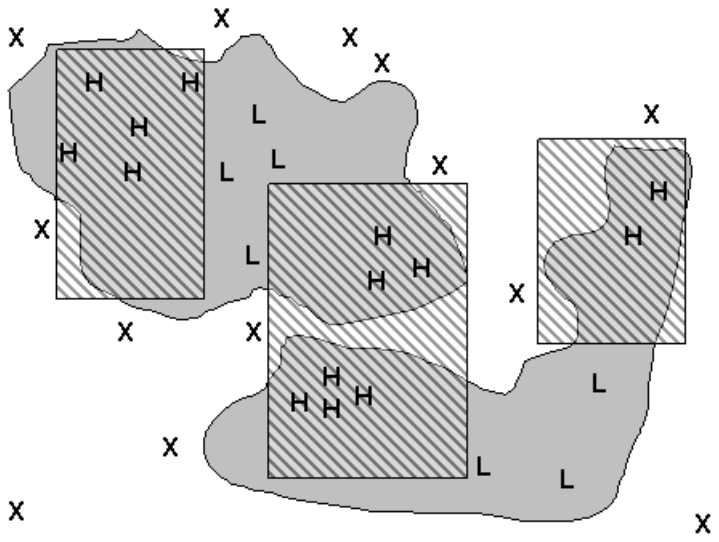
The approximation of the feasible area may miss the actual solution when the learned description is overspecialized. In such cases, some feasible candidate solutions may be missed, and the actual solution to the optimization problem may not be achieved. This

may happen especially when it approaches the border of the set of feasible solutions. This problem can be solved by (1) learning maximally general descriptions of feasible solutions, (2) checking randomly selected candidate solutions that do not satisfy the learned description of the feasible area, or (3) using a flexible rule interpretation, as defined in attributional calculus by Michalski (2000a). After instantiation, sets S_f and S_n are updated with new candidate solutions. The method is illustrated in Figure 4-8 and Figure 4-9.



H – high-performing candidate solution, L – low-performing candidate solution,
X – infeasible candidate solution, gray area – feasible space, rectangles – rules

Figure 4-8: Feasible space approximation.



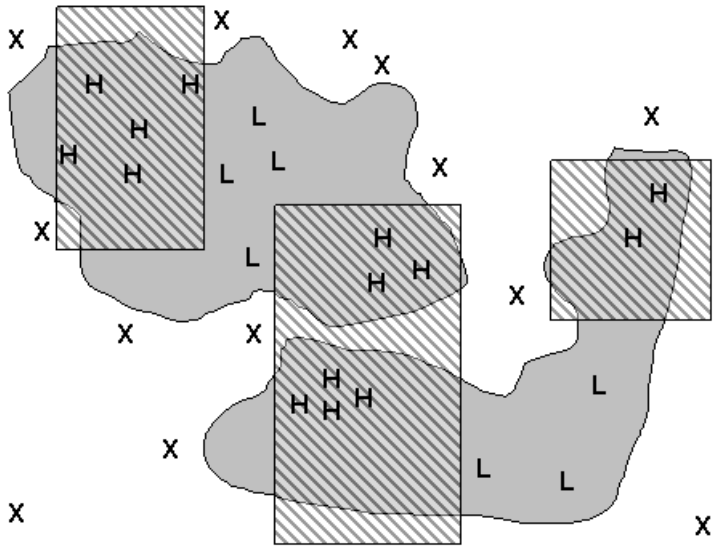
H – high-performing candidate solution, L – low-performing candidate solution,
X – infeasible candidate solution, gray area – feasible space, rectangles – rules

Figure 4-9: Intersection of the learned hypothesis and the feasible space approximation where candidate solutions are created and tested against constraints.

4.5.3 Using Infeasible Candidate Solutions as a Contrast Set for Learning

The latter described method for handling general constraints keeps a list S_n of infeasible solutions and uses them to constrain hypothesis formation in LEM by adding the solutions from S_n to the group of low-performing candidate solutions. Hypotheses learned using such a method not only describe high-performing candidate solutions, but also avoid areas with infeasible solutions. After instantiation, the set S_n is updated with new infeasible candidate solutions. Similarly to the previously presented methods, to avoid missing solutions, when rules are overspecialized, flexible rule interpretation may be used.

Figure 4-10 presents an example hypothesis learned with infeasible examples used as a contrast set.



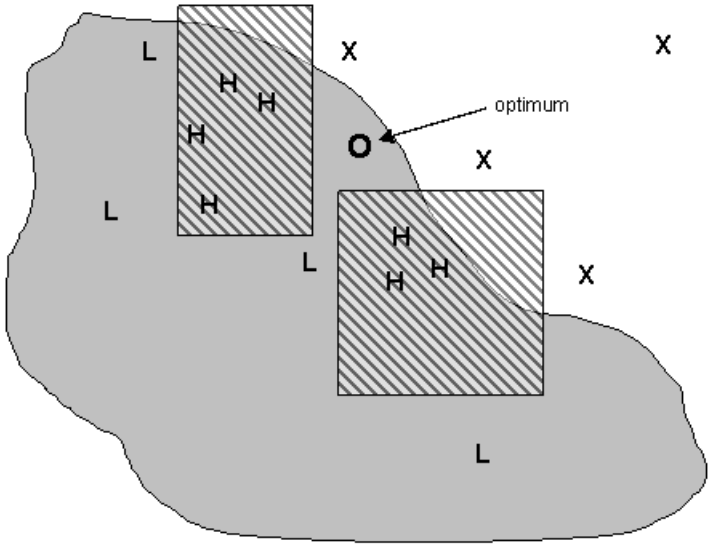
H – high-performing candidate solution, L – low-performing candidate solution,
X – infeasible candidate solution, gray area – feasible space, rectangles – rules

Figure 4-10: Hypothesis learned with set of infeasible solutions used as negative examples.

4.5.4 Discussion

An important issue concerning the second and third method for handling general constraints is that the sets S_f and S_n may be very large. This may happen when the evolution process is long, or when many candidate solutions are rejected. This may negatively affect the performance of the learning program. To overcome this problem, subsets of S_f and S_n are used. Selection of the subsets can be random, or through choosing the most representative candidate solutions. The selected subset should be relatively small in order to guarantee efficient learning, yet it should be large enough for close approximation. Methods for selecting examples applicable to AQ learning were investigated, for example, in the ESEL program (Michalski and Larson, 1978).

It is often the case that one or more constraints are active at the optimum, meaning that the optimum is located at the edge of the feasible area. In such a case, the optimum cannot be surrounded by feasible, high-performing known candidate solutions; those are present from only "one side." Moreover, attributional rules learned by AQ programs represent hyperrectangles that may easily miss such a solution. This problem, easily illustrated for numerical domains (Figure 4-11), can be generalized to symbolic domains.



H – high-performing candidate solution, L – low-performing candidate solution,
X – infeasible candidate solution, gray area – feasible space, striped rectangles – rules

Figure 4-11: An example of a missed optimum for a constrained optimization problem.

It is important to note that this problem applies also to non-constrained optimization. For example, when optimizing the Rosenbrock function (see Chapter 6), for which the optimum is located on the narrow ridge where values of all attributes are equal, learned rules may easily miss the solution. This problem can be solved by using flexible rule interpretation described in attributional calculus by Michalski (2004a). A condition with

a linear attribute (ordinal, cyclic, interval, ratio, etc.) can be interpreted *strictly* (match if the condition is satisfied, do not match otherwise) or *flexibly* (where the *degree of match* depends on the distance of the matched value to the condition). If the condition is satisfied, the rule is matched with the degree of match equal one, otherwise, the condition can still be matched with the degree of match less than one (Michalski, 2004a). The degree decreases linearly with the distance from the condition. For example is a condition says [Date=May 1 … October 19] and the date is October 20, there is a high chance that the date also should be matched, but with degree of match lower than if the condition was matched strictly. A linear function that defines the degree of match for the flexible selector interpretation is proposed by Michalski (2004a). Other than linear functions that can be used for the flexible selector interpretation have been investigated in fuzzy logic as described by Zadeh (1965).

Flexible interpretation applies also to the process of instantiation in the learnable evolution model, as described in Chapter 3. For each condition, the method generates a value that matches strictly the condition with probability p and is outside the condition with probability 1-p (e.g., p=0.95). The probability of a value being selected decreases with distance from the condition. Experiments have shown that for several problems (both constrained and not constrained) the method slightly slows convergence of candidate solutions, but also helps to maintain diversity.

4.6 Flexible Constraints

All methods discussed in the previous sections were designed to cope with strict constraints that need to be satisfied. Flexible constraints, on the other hand, may be violated if such a violation leads to better performance (fitness) of a candidate solution or when the program is unable to find a feasible solution. The latter may happen for example for overconstrained problems in which there is no possible solution that satisfies all constraints and the goal is to find a solution which maximizes the fitness function and satisfies as many constraints as possible.

Strict constraints are usually added to the problem specification in order to eliminate candidate solutions that make no sense. For example an aircraft design where wings are not attached to the fuselage does not make sense, and it is obviously infeasible. Flexible constraints usually represent experts' knowledge, and are used to guide the evolution process. This is done by not allowing candidate solutions that contradict the general knowledge in the area, or desired features of designs. However, flexible constraints can be ignored if that leads to better solutions. It is important to note that there are two ways of giving advice to the system, by flexible constraints and by plausible modifications of the representation space, as discussed in Chapter 5.

A flexible constraint's *importance* is a number that defines the *degree of infeasibility* when the constraint is violated. Higher importance is assigned to constraints that should not be violated, and lower importance is assigned to those that may be violated if needed.

The *infeasibility* of a candidate solution is the sum of importances of the constraints violated by the candidate solution. Individuals with infeasibility equal to zero are feasible. Theorem 2 generalizes Theorem 1 presented in Section 4.1 to flexible constraints. In particular, it describes the situation when high-performing candidate solutions have infeasibility greater than zero.

Theorem 2: If k is the highest infeasibility of high-performing candidate solutions provided to AQ learning program, then each rule in a learned hypothesis can be instantiated with a candidate solution of infeasibility at most k.

Proof: Each rule covers at least one high-performing candidate solution. Since infeasibility of all high-performing candidate solutions is at most k, the candidate solution covered by any rule has infeasibility at most k.

To handle flexible constraints, the lexicographical evaluation functional given by (4-31) can be used. It allows solutions with some infeasibility when their fitness is high.

$$<(\text{MinInfeasibility}, \tau); (\text{MaxFitness}, 0)> \tag{4-31}$$

Here, τ is a tolerance of allowing infeasible solutions. Strict constraints should be assigned very high importances, in order not to allow them to be selected by the LEF (see also Section 4.7).

4.7 Starting with no Feasible Solutions

Methods for handling constrained optimization problems presented in the previous sections make the assumption that there is a known population of feasible candidate solutions to start the evolution process. It is, however, often the case that such solutions are not known prior to evolution and may be difficult to find. Methods that can be used to start evolution with no initial feasible solutions can be classified into sequential and parallel. The former start by searching for a starting population of feasible candidate solutions and then apply the methods discussed in previous sections, while the latter search for feasible candidate solutions in parallel to the optimization process.

In the sequential methods, candidate solutions can be randomly generated in the search space until sufficiently many feasible solutions are found. This method is often very ineffective, especially when the feasible area is small in relation to the entire search space, but its advantage is that it gives good diversity to the initial set of feasible solutions. Another possibility is to use two-step optimization by applying first an evolutionary search to find a population of feasible solutions (here fitness reflects feasibility of the candidate solutions and ignores the actual optimization problem). When the starting population of feasible candidate solutions is found, the normal constrained optimization process is performed as described in the previous sections.

The second class of methods searches for feasible solutions in parallel to the optimization process. This can be done in several different ways, for example by applying multiobjective optimization (e.g., Surry, Radcliffe, and Boyd, 1995; Deb, 2001) or

penalty functions. Another possibility is to use the lexicographical evaluation functional (see Chapter 2) with criteria (4-32) for selection of candidate solutions into a new population and optionally also during selection into the H-group and L-group before learning.

$$<(\text{MinInfeasibility}, \tau); (\text{MaxFitness}, 0)> \tag{4-32}$$

By using MinInfeasibility with a given tolerance τ as the first criterion, the method prefers selection of candidate solutions with lower infeasibility (those that violate fewer constraints) and in this initial stage treats their fitness as less important (the second criterion). Once all candidate solutions are feasible, there is no need for using the LEF any more, and the selection can be based solely on the fitness values.

To illustrate the method based on LEF, the program was applied to the optimization of the G1 function, a well-known constrained optimization testing problem. For this example, infeasibility is defined as the number of violated constraints. Figure 4-12 illustrates infeasibility of the best individuals in each of 50 generations. The program was applied with four different values of tolerance in the LEF, and each experiment was repeated ten times. For details of the experiment and the G1 function, please refer to Chapter 6.

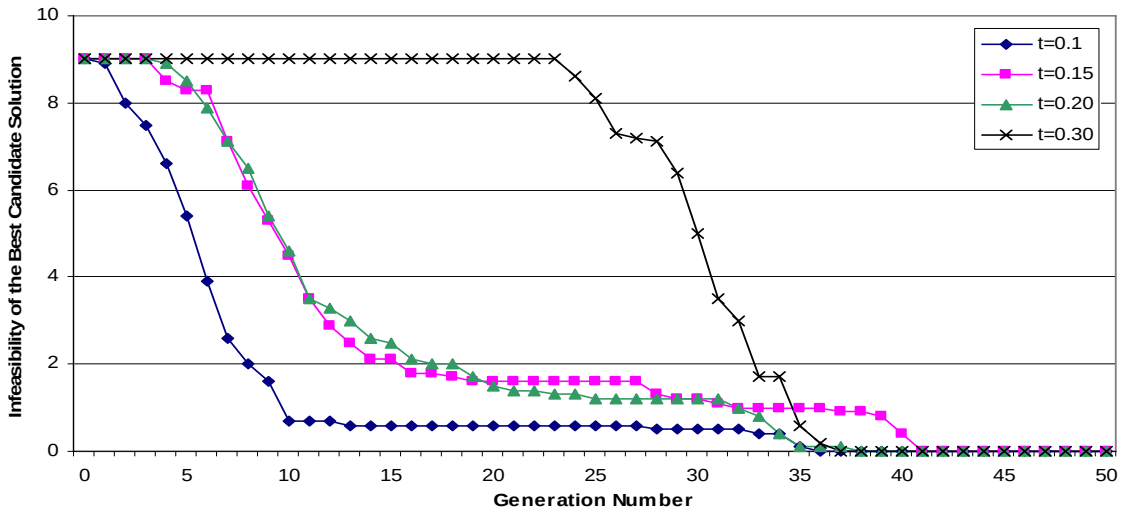


Figure 4-12: Infeasibility of the best individuals in different generations when optimizing the G1 function. Each line represents average of ten executions for a given tolerance t.

It is not surprising that the higher the tolerance is, the slower the program's convergence to the feasible solutions. For this particular problem, however, the feasible solutions are found after roughly the same numbers of generations.

4.8 Conclusion

This chapter presented several methods for handling constraints. In addition to an overview of methods known in the literature, such as penalty functions, repair algorithms, and multiobjective optimization, it presented methods specifically developed for the learnable evolution model. Constraints, depending on their form, can be classified as instantiable, which can be used directly to produce new candidate solutions, and general for which it can be computed whether they are violated or not (possibly with a degree of violation). Instantiable constraints, which consist of ordered lists of conditions in the specific form [ATT rel EXPR] are handled by the presented backtracking algorithm. The

method is more efficient in terms of execution time than a standard rejection algorithm. General constraints can be handled by three different methods that incorporate them into the learning process, either as a contrast set, or by a secondary learning process.

Additionally, this chapter discussed problems of handling flexible constraints that may be violated under specific conditions, and methods for starting with no feasible solutions in the initial population. Experimental application of the presented techniques to a set of problems is presented in Chapter 6.

CHAPTER 5 REPRESENTATION SPACE

The problem *representation space*, also known as *search space*, is the set of all possible problem solutions. Finding an appropriate representation space for a given optimization or learning problem is one of the most important and challenging tasks. This chapter introduces the topic of representation space in the learnable evolution model and proposes automated methods for its improvement. These methods are built upon results previously obtained in the field of machine learning, in which constructive induction has been introduced. This chapter is organized in the following way: Section 5.1 presents two simple examples illustrating the importance of improving the representation space. Section 5.2 presents a general representation-based learnable evolution model schema. Detailed descriptions of methods of searching for the best representation space, creation of new candidate solutions from modified representations, and how LEM controls these processes are presented in Sections 5.4 - 5.6.

5.1 Two Examples Illustrating Modifications of Representation Space

In order to illustrate the importance of representation space, two simple examples are used in this section. The examples are based on the generalized Rosenbrock function and a designed circular shape function. The Rosenbrock function, defined by expression

(6-3) and graphically illustrated in Figure 6-26, has an almost flat ridge for all variables equal, and is very steep outside of the ridge. Provided with a sufficient number of known candidate solutions and past hypotheses, it may be possible to discover that it is necessary only to search for the solution along the ridge. When such information is discovered, the optimization problem can be reduced to one dimension. When the generalized Rosenbrock function is defined using a large number of attributes, even discovery of the fact that values of some of the variables need to be equal may significantly reduce the complexity of the optimization problem.

For example, Figure 5-1 presents rules learned by LEM3 during optimization of the Rosenbrock function of 5 variables without discovering the concept of the ridge, and Figure 5-2 presents a rule learned for the same data with the ridge discovered. In the example, 4 high-performing and 7 low-performing candidate solutions were provided to LEM's learning module.

```
[C is high-performing] ⇐ [x0=-1.5..2.5: 3,3,50%] &
                         [x1=-1.5..2.5: 3,4,42%] &
                         [x2=-1.5..2.5: 3,3,50%]  : p=3,n=0,u=2

[C is high-performing] ⇐ [x0=1.5..5: 2,2,50%] &
                         [x1=1.5..5: 2,2,50%] &
                         [x2=1.5..5: 2,3,40%] &
                         [x3=1.5..5: 2,1,66%]     : p=2,n=0,u=1
```

Figure 5-1: Rules learned during optimization of the Rosenbrock function of 5 attributes.

The above hypothesis describes high-performing candidate solutions using two rules. The first rule consists of three conditions specifying attributes x_0, x_1, and x_2, all defined in the range -1.5 to 2.5. The annotation after the rule indicates that it covers 3 high-performing candidate solutions (p=3), 0 low-performing candidate solutions (n=0), and has a unique

coverage that is 2 (u=2). The small-sized numbers used in the conditions represent positive and negative support, and confidence of conditions, respectively. The second rule can be interpreted similarly.

```
[C is high-performing] ⇐
        [equal(x0, x1, x2, x3, x4; ε=0): 4,0,100%] : p=4,n=0,u=3
```

Figure 5-2: Rule learned during optimization of the Rosenbrock function of 5 attributes with discovered ridge.

The hypothesis presented in Figure 5-2 consists of one rule that consists of a single condition sufficient to consistently describe all provided candidate solutions, as it covers all four high-performing candidate solutions (p=4) and no low-performing solutions (n=0). The only condition included in the rule uses equal(x_i; ε), i=1..k function that returns true if values of all attributes x_i are equal within margin/tolerance ε, and false otherwise. The margin can be defined either as maximum difference between values of attributes. All candidate solutions satisfying the condition lie on the narrow ridge of the Rosenbrock function. In this case, by introducing to the representation space a new attribute, the optimization problem is reduced to one dimension.

The rule presented in Figure 5-2 is sufficient to discriminate between high- and low-performing candidate solutions. However, experimental evaluations of the learnable evolution model indicate that characteristic descriptions tend to perform better. In the characteristic mode, the program discovers the rule shown in Figure 5-3. It combines the condition with the equal(xi; ε) function with a basic condition.

```
[C is high-performing]
  ⇐ [x2=-1.5..5: 4,5,44%] &
     [equal(x0,x1,x2,x3,x4; ε=0): 4,0,100%]: p=4,n=0,u=3
```

Figure 5-3: Characteristic rule learned during optimization of the Rosenbrock function of 5 attributes with discovered ridge.

The procedure for instantiating rules with the equal function consists of three steps. First, the intersection of conditions with all attributes used by the equal function is computed. If no conditions are present for a given attribute, its entire domain is used. In the second step a value is selected from the intersection. Finally, all attributes used in the equal function are assigned the selected value.

The second example presented in this section shows a suggestion-taking capability that allows the user to propose plausible transformations of the representation space to the system. Such transformations may help in learning simpler and more accurate descriptions of high-performing candidate solutions. Hypotheses learned in better representation spaces are more likely to contain optimal solutions.

Suppose that during the evolutionary process of maximizing the function given by (5-1), the learnable evolution model generated 30 positive and 30 negative examples graphically illustrated in Figure 5-4 (high- and low-performing candidate solutions are marked as "+" and "-", respectively).

$$f(x,y)=\cos(x^2+y^2-1)+x^2/4 \qquad (5\text{-}1)$$

The LEM's learning module, provided with the examples, generated six rules shown in Figure 5-5

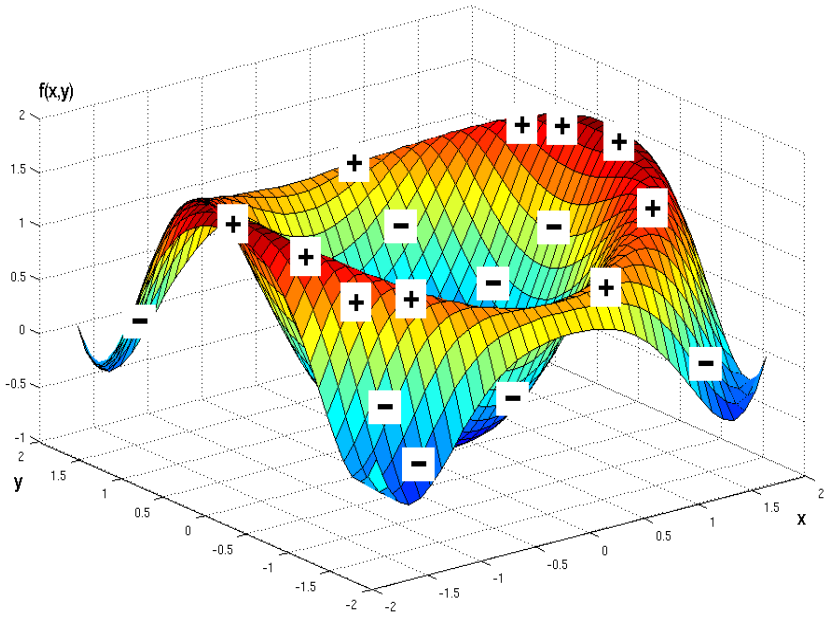


Figure 5-4: An illustration of function (5-1) with marked high- and low-performing candidate solutions.

```
[Group=H]
  ⇐ [x=-1.36..-0.77: 8,3]&[y=-0.82..-0.006: 13,5] : p=7,n=0
  ⇐ [x=-1.071..0.07267: 19,9]&[y=0.52..1.05: 11,3] : p=7,n=0
  ⇐ [x=-0.58..1.07: 18,12]&[y=-0.81..-0.08: 10,4] : p=6,n=0
  ⇐ [x=0.25..1.17: 11,6]&[y=0.03..0.99: 13,8] : p=5,n=0
  ⇐ [x=0.27..0.39: 2,0] : p=2,n=0
  ⇐ [y=0.4288..0.5237: 1,0] : p=1,n=0
```

Figure 5-5: Rules leaned by LEM when optimizing the function (5-1).

Suppose now that the user, based on domain knowledge, provides the program with a suggestion to change the search space. The suggestion is to introduce two new attributes w and z, defined as mathematical formulas: $z = \sin(x + y)$ and $w = x^2 + y^2$. In the presented methodology, some of the suggestions given to the system may be correct, and some may be incorrect, or no suggestions given to the system may be correct. It is the role of the LEM learning module to determine which of the provided suggestions help to describe high performing candidate solutions, and which suggestions should be ignored.

In the presented example, the second provided attribute, w, helps the system to easily capture the high performing candidate solutions. Given the new attribute, LEM learns the rule (5-2).

$$[\text{Group=H}] \Leftarrow [w = 0.8375..1.213: 30,0]: p=30, n=0 \quad (5\text{-}2)$$

Due to the suggestion, LEM is able to reduce the optimization problem to one dimension, w. In next steps LEM adds additional conditions in order to narrow the search along the w dimension, producing rules such as (5-3).

$$[\text{Group=H}] \Leftarrow [w=0.8..1.2: 30,7]\&[x=1..3: 10, 5]: p=10, n=0 \quad (5\text{-}3)$$

In the presented example, an explicit definition of a new attribute was provided, but in general, LEM can be provided with a general statement such as w = poly(x, y; 2), meaning polynomial of x and y of degree 2, or simply a general request for discovering new attributes.

5.2 Representation Space in the Learnable Evolution Model

Evolutionary computation methods, including the learnable evolution model, can be viewed as a population of candidate solutions and a set of operators that modify the population. In this model, selection operators that are used to choose candidate solutions that remain in the population and evaluation operators (fitness and constraints) are representation independent, that is, evaluated in the original representation. Innovation operators used to create new candidate solutions, e.g., by learning and instantiation or probing, may depend on the representation of candidate solutions. The role of representation was illustrated in Section 5.1, where LEM's learning and instantiation

operator was applied in the original and modified representations. Operators that are used to modify the representation space are also representation-dependent, as they relate to the current representation space, which they may further modify. These dependencies are graphically illustrated in Figure 5-6.

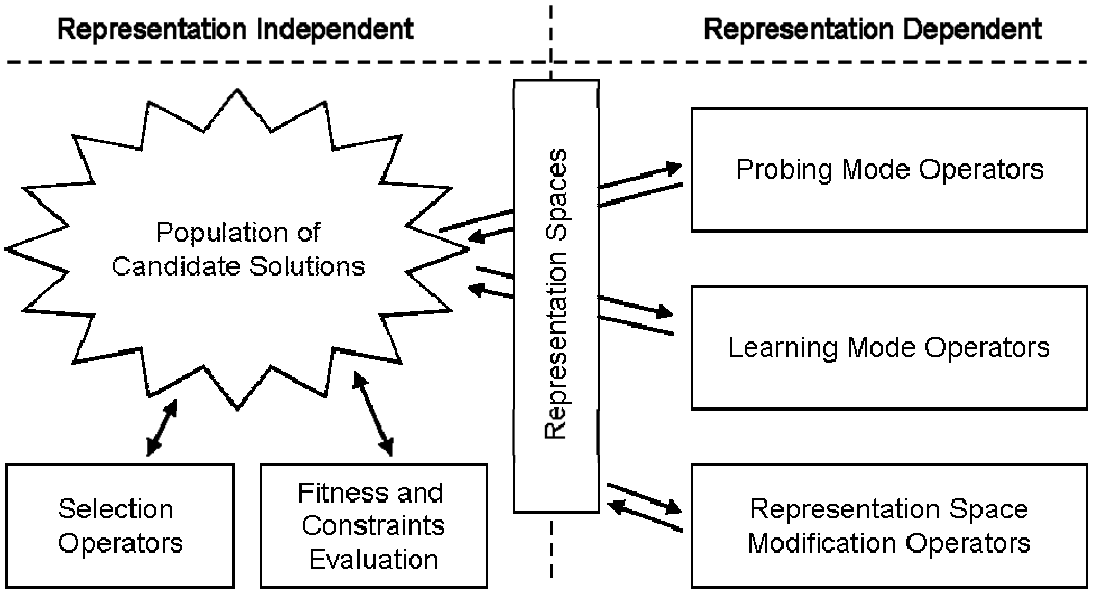


Figure 5-6: A general schema of dependencies on representation in LEM.

The above schema that assumes that candidate solutions are stored in their original representation is only one of many possibilities. Others include a population of candidate solutions in a modified representation and evaluation of fitness or constraints that requires translation to the original space. Such a solution is often used in evolutionary computation. For example, in genetic algorithms that use bit coding of individuals (genotypes), the fitness evaluation requires decoding these bits into the original representation (phenotypes).

In LEM3, candidate solutions are represented in the original space and translated into modified spaces in order to apply change operators. This is possible because LEM3 implements several attribute types available in attributional calculus. Storing candidate solutions in their original representation have several advantages. Each candidate solution needs to be in the original space in order to evaluate its fitness and constraints. It is easier to translate solutions from the original space into the modified one than to define a reverse transformation of the space. New candidate solutions are created in the original representation and the reverse transformation is realized by the instantiation process, i.e. when hypothesis formulation includes representation space transformations. Also, because the process of updating representation space is incremental, the reverse transformation would need to consist of multiple steps, which is not practical.

5.3 Constructive Induction

The original representation space provided to a machine learning, data mining, or evolutionary computation system may be inadequate for performing the desired task for concept learning, pattern discovery, optimization, etc. A careful design of the representation space is considered one of the tasks most important to successful applications in these areas. Most concept learning programs use so called *selective induction*, meaning that hypotheses are generated by selecting attributes and their values in form of patterns describing the provided data. Such a process does not involve modifications to the original representation space.

Constructive induction (CI) methods automatically create new representation spaces based on the original representations, which allows the determination of relationships that cannot be represented in the original spaces. New representations are created by removing attributes irrelevant to the considered problem, modifying domains of attributes (for example by abstracting values), and by creating new attributes. Formally, the constructive induction process can be characterized by the function (5-4), where E is the original representation space and E_C is the modified representation space.

$$\Psi: E \rightarrow E_C \tag{5-4}$$

The problem of searching for the best representation space has been investigated in field of machine learning by numerous researchers (e.g. Bloedorn and Michalski, 1991; 1998; Wnek and Michalski, 1994; Bensusan and Kuscu, 1996; Markovich and Rosenstein, 2002; Muharram and Smith, 2005). Constructive induction methods can be classified into four categories: *data-driven constructive induction*, which uses results of analysis of data to modify the representation space, *hypothesis-driven constructive induction*, which uses results of analysis of preliminary hypotheses to modify the representation space, *knowledge-driven constructive induction*, which uses domain knowledge provided by experts to modify the representation space, and *multistrategy constructive induction*, which combines the above methods.

Data-driven constructive induction (DCI) searches for the best representation space by analyzing the data and the current representation. Among the best known DCI methods are AQ17-DCI (Bloedorn and Michalski, 1991; 1996) and methods based on evolutionary

search for the best representation space (e.g. Bemsusan and Kuscu, 1996; Krawiec, 2002; Muharran and Smith, 2005).

AQ17-DCI uses an extensive search over the set of possible representations. Newly generated attributes, e.g., by abstraction or construction, are evaluated using a statistical measure. When the quality of all attributes according to the statistical measure is satisfactory, the program employs AQ learning in order to generate hypotheses using a part of the data (the primary training dataset), and evaluates the hypotheses using rest of the training data (the secondary training dataset). If the quality of the learned hypotheses (measured as predictive accuracy) is not satisfactory, the program returns to representation space modification; otherwise the program learns hypotheses using the entire dataset and ends execution. A general schema of the presented algorithm is illustrated in Figure 5-7. A version of this algorithm is implemented in the AQ21 system used for hypotheses formulation in LEM3.

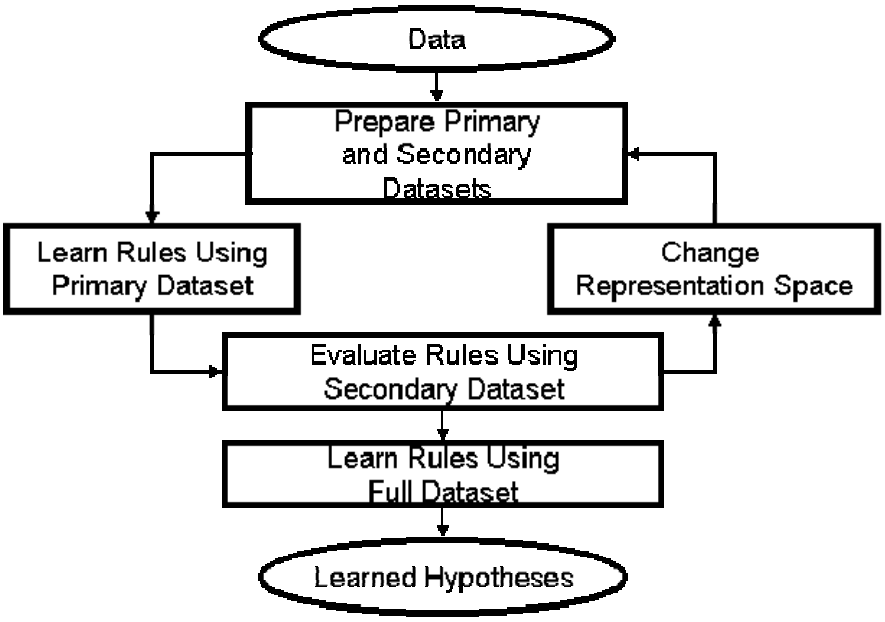


Figure 5-7: General diagram of constructive induction in AQ learning.

Hypothesis-driven constructive induction (HCI) searches for the best representation space by analyzing previously learned hypotheses (e.g. Wnek and Michalski, 1991; Wnek, 1993; Wnek and Michalski, 1994). The AQ17-HCI system follows the general framework of AQ constructive induction systems presented in Figure 5-7. It extends the representation space by creating new attributes that represent strong relationships in previously learned rules. Such patterns can be groups of rules, parts of rules, individual conditions, or groups of attribute's values from a condition.

Knowledge-driven constructive induction (KCI) improves the representation space based on background knowledge provided by an expert or accumulated during past experiments. The knowledge provided by an expert may include information about attributes, information about particular learning (optimization) problem, information about dependencies between attributes, previously learned hypotheses etc. Background

knowledge about attributes may be communicated in the form of their types (nominal, structured, rank, absolute, ratio, etc.), discretization, importance, cost, and all other information potentially useful when inducting hypotheses. Dependencies between attributes are given in the form of arithmetic expressions (A-rules) or logic expressions (L-rules) that extend the representation space and possibly guide the system toward the problem solution.

Knowledge about plausible representation space transformations is provided in the form of suggestions of three different levels of generality. On the lowest level, explicit suggestions are given, for example, "There is a relation between variables x, y, and z such that an optimal solution satisfies: x = 4*y + z," or "There may be a relation between variables x, y, and z such that an optimal solution satisfies: x = 4*y + z." There is a fundamental difference between the two statements. The former one can be viewed as a constraint, making feasible individuals only those that satisfy the given formula, while the latter states that it is plausible that the solution satisfies the formula. If the knowledge is given as a constraint, one of the methods used in Chapter 4 should be applied. In the case of plausible modifications, the suggestions provided to the system may be contradictory or incomplete, and it is the role of a learning system to determine their correctness and how to use them in the learning process.

In the second level of suggestion, background knowledge is given in the form of equations in which coefficients need to be discovered by the system. For example "There might be a relation between variables x, y, and z such that an optimal solution satisfies:

x = a*y + b*z, where a and b are parameters." The program needs to find the coefficients a and b based on the known candidate solutions. This can be done using known methods, such as least squares.

On the most general level of suggestion, the system is provided with background knowledge in the form of a general statement about possible relationships in the data. For example, "There is a polynomial relation between x, y, and z," or "There is a trigonometric relation between variables x1, x2, and x3." These types of suggestions are handled by applying data-driven constructive induction in which operators and functions are defined by the user, and only to specified attributes. It is not clear where to put a distinction between knowledge- and data-driven constructive induction methods in this case. Clearly, the distinction should be based on the amount of background knowledge used by the system. With an increasing amount of background knowledge, less data is needed to search for the best representation.

Multistrategy constructive induction (MCI) combines hypothesis- and data-driven constructive induction methods. MCI uses advanced reasoning to choose which operators should be applied to modify the representation space. The decision is made based on high level meta-rules tailored to the problem (e.g., Bloedorn, Michalski, and Wnek, 1993; Bloedorn, 1996; Bloedorn and Michalski, 1998).

5.4 Automated Improvement of Representation in the Learnable Evolution Model

In order to better characterize high-performing candidate solutions, it is sometimes necessary to change their representation, as illustrated by the two examples in Section 5.1. This section illustrates how methods of constructive induction, in particular a combination of data-driven constructive induction and suggestion-taking (a special case of knowledge-driven constructive induction), can be applied to improve hypothesis generation in the learnable evolution model. Methods for creating new candidate solutions from hypotheses learned in such transformed representation spaces will be discussed in Section 5.5.

The considered method for transforming representation spaces in the learnable evolution model follows the general constructive induction methodology presented in Section 5.3 and illustrated in Figure 5-7. At each step of evolution, LEM applies machine learning to distinguish between selected high- and low-performing candidate solutions. This process may be performed in the original representation space, in a previously transformed representation space, or may involve transformations of the representation space. For instance, LEM3 while applying the AQ21 learning program may use its constructive induction capabilities. The following subsections describe in detail methods used in LEM for transforming representation spaces and their relation to the original constructive induction methods used in concept learning.

5.4.1 Transformation Algorithm

The methodology used in this dissertation for the learnable evolution model transformation of representation spaces, follows the general algorithm used in constructive induction presented in Figure 5-7. The program starts by splitting the training dataset into primary and secondary sets; the former is used to hypothesize rules that describe high-performing candidate solutions, and the latter is used to test the hypotheses. When the quality of hypothesis learned in the current hypothesis is below a given threshold, the program tries to improve the representation space. The improvements aim at helping to better capture the high-performing candidate solutions in the learned hypotheses. The improvements include selection of the most relevant attributes, construction of new attributes, and adjustment of discretization of continuous attributes. The learning program is applied to examples in the new representation, and the resulting hypothesis is evaluated. The following sections describe this algorithm in detail.

5.4.2 Construction of Attributes

The most important feature of constructive induction presented in this dissertation is its ability to create new attributes. These new attributes are designed to more adequately capture high-performing candidate solutions, and help in distinguishing high- and low-performing solutions with simple well performing hypotheses. The new attributes can be in the form of equations involving numeric attributes, and special forms involving symbolic attributes (e.g., count, equality). Unlike in concept learning, where the goal is to learn simple and well-performing hypotheses, in LEM an additional requirement needs to be satisfied. The requirement is that the learned hypotheses must be in such a form that

the instantiation process is efficient. The following paragraphs describe two algorithms for constructing new attributes, first one that constructs attributes in a general form, and then one that constructs attributes in the special instantiable form.

```
Attributes = Current attributes
For depth = 1 to maxdepth
  For each attribute Att in Attributes
    For each attribute Att1 ≠ Att in Attributes
      Create attributes using operators +, -, *, / on Att and Att1
    Create attributes Att * Att and Att + Att
    Create attributes using requested special functions on Att
    Evaluate quality of new attributes and add to Attributes list if
      the quality is above a given threshold
Remove all attributes with quality below a threshold
Keep only k best attributes
```

Figure 5-8: Algorithm for constructing general form of attributes.

The algorithm presented in Figure 5-8 starts with attributes from the current representation space. The representation may be the original or it may be already modified. New attributes are created by applying standard arithmetic operators (+, -, *, /) to each pair of already existing attributes, and by applying user-specified special functions such as sin(X), cos(X), sqrt(X), etc. that apply to numeric attributes. For symbolic attributes, program may construct count, and equality attributes. Each newly created attribute is evaluated and if its quality is above a given threshold, it is added to the list of attributes. Finally, new representation space is assembled using at most the k highest quality attributes, where $k << P$ is a user-defined parameter significantly smaller than the number of high-performing examples in the current population.

The above algorithm creates new attributes in a general form. In order to efficiently instantiate hypotheses that include constructed attributes, new attributes need to be in

special instantiable form such as (4-6). The following algorithm creates new attributes in a special form (5-5) similar to (4-6):

$$\text{ATT} +/- \text{EXPR} \tag{5-5}$$

```
Attributes = Current attributes
For depth = 1 to maxdepth
  For each attribute Att in Attributes
    Add Att to the list of used attributes
    For each attribute in Attributes and not in the used attributes
      Create attributes using operators +, -, *, / on Att and Att1
      Evaluate quality of new attributes
    Create attributes using requested special functions on Att
    Evaluate quality of new attributes and add to Attributes list if
      the quality is above a given threshold
Remove all attributes with quality below threshold
Remove worst quality attributes so only k best are kept
```

Figure 5-9: Algorithm for constructing instantiable attributes.

The constructed attributes are used by the AQ learning module as standard numeric attributes, as their values can be computed for each example prior to execution of the learning module. Because of constraints imposed on the form of these attributes, there are transformations that cannot be represented. For example (5-6) cannot be created by the presented algorithm.

$$X^2 + X + Y^2 + Y \tag{5-6}$$

Both algorithms presented above tend to be inefficient or even impossible to apply for representation spaces with very large numbers of attributes. For example, if there are 100 original attributes and the program seeks only combinations of pairs of attributes, there are 40,000 potential new attributes for which the quality measure needs to be computed. Even if the method implements additional heuristics, such as checking monotonicity of the attributes, as in the ABACUS system (e.g. Falkenhainer, 1984; Falkenhainer and

Michalski, 1990), the number of possible combinations will still be very large. In concept learning problems in which exploration of these possibilities is usually done only once, it still may be possible to do. In LEM, however, such a process is repeated many times at different stages of evolution, thus it seems to be computationally too expensive for practical applications. One possibility of solving this problem is to combine only high quality attributes. Such attributes are more likely to produce new high quality attributes than combinations of low quality ones. Following this idea, the attribute generation algorithms presented in Figure 5-8 and Figure 5-9 need to be modified to narrow the search space. This is done by applying beam search – at each step of attribute generation only max best attributes are selected, where max is a parameter.

A simple experiment performed for minimization of the Rosenbrock function (see Chapter 6) shows that with an increased max parameter LEM3 finds better solutions after 10 generations (1100 fitness evaluations), but its execution time significantly increases. In this example the program was run for only few steps for demonstration purposes, thus the solutions are far from optimal. This is illustrated in Table 5-1 where fitness of the best obtained result and the time of execution are compared for LEM3 without constructive induction, and LEM3 with constructive induction for which the max parameter set to 5 and 10, respectively. The minimal fitness value of the Rosenbrock function is 0.

Table 5-1: The best fitness value and time of LEM3 execution with no constructive induction, and constructive induction with max parameter set to 5 and 10 respectively.

Max	Fitness Value	Time of Execution
No CI	121116	3 seconds
5	117872	169 seconds
10	109351	1007 seconds

Although computation time significantly increases with the max parameter, for problems for which the fitness function evaluation takes a significant amount of time, it is desirable to keep it large. For example, if evaluation of the fitness takes about 10 minutes, which is not unusual in design problems, 1100 evaluations takes about 11,000 minutes (over 7 days) which is significantly larger than 1004 seconds (less than 17 minutes) overhead used by applying the constructive induction in LEM.

5.4.3 Discretization of Continuous Attributes

Full precision of continuous attributes is often too high, especially at the early stages of evolution. Thus, it seems plausible to roughly discretize attributes at the beginning of evolution and increase their precision in the most promising areas as the evolution progresses. This section describes an adaptive discretization method used to improve representation space in LEM. The method is a slightly modified version of the *adaptive anchoring discretization* (ANCHOR) described by Michalski and Cervone, (2001).

The method starts by creating the first order approximation FOA, defined as the closest numbers with a digit that can be followed only by zeros. For example the value 12.375 is approximated by 10, the value 1836.3672 is approximated by 2000, and -37,446.22 is approximated by -40,000. As one can see such a method follows the general idea of natural induction, as it resembles the process of discretizing numbers by humans to the closest "round" number, not necessarily evenly distributed. Note that such a method is strongly biased toward higher precision near zero. Because the AQ21 learning program requires definition of domains of all attributes, such a domain for attribute X_i discretized

to Xd_i is defined as $\{y_i: y_i = FOA(x_j)$ for all x_j in domain of $X_i\}$. For example if the domain of the attribute X is [-33.2, 476.20], it is replaced by {-30, -20, -10, -9, -8, …, -1, 0, 1, 2, …, 10, 20, …, 90, 100, 200, …, 400, 500}.

Discretization is adjusted in the most promising areas during the evolution process. Suppose that during an evolution process, the value of the attribute X in the best individual is x and the program determines a need for adjusting discretization. The program increases precision in a neighborhood of x between its two neighbors in the discretization. Each point in this neighborhood is replaced by the second order approximation, which is defined by the closest numbers with at most two digits followed by zeros or a single digit followed by one decimal digit (for x in [-10, 10]). For instance if in the previous example x = 40, precision in the range [30, 50] is increased by adding values 31, 32, …, 39, 41, 42, …, 49 to the domain of attribute Xd. If the discretization needs to be adjusted in area in which the second order discretization was introduced, the third order discretization is created in an analogous way, and so on.

The discretized attributes are semantically equivalent to their original forms, as discretization preserves type-invariant operations (Michalski and Wojtusiak, 2007). This means that a discretized ratio attribute still have properties of a ratio attribute, a discretized interval attribute still have properties of an interval attribute etc. A process of applying transformations to discretized attributes works as follows: values in discretized examples are first undiscretized, to get continuous values in the intervals, then the operations are executed. Results are discretized back into the discrete form. This feature

of discretized attributes is important, as it makes it possible to construct new attributes that involve mathematical formulas with discretized attributes.

Another important property of the discretization is the fact that it does not preserve feasibility in constrained optimization problems. A feasible (satisfying all problem constraints) candidate solution in the original representation space may not be feasible if one or more attributes are discretized. To illustrate this fact, it is enough to consider a simple example. Suppose that the optimization space consists of one continuous attribute Z and a constraint C = [Z < 47.23]. A first order approximation of value Z = 46 which is feasible is Zd = 50 which does not satisfy C. Note that to solve this problem, in many cases it is not sufficient to simply discretize values in the constraint. A similar example can be given for which an infeasible candidate solution becomes feasible after discretization. This means that when applying the learnable evolution model to constrained optimization problems, one must carefully consider consequences of using discretization. Feasible solutions may not exist in a modified optimization space with discretized attributes. This also can be addressed by relaxing strict constraints and flexible constraints instead. Such a method may not be, however, applicable to all problems.

5.4.4 Selection of Attributes

Selection of the most relevant attributes (a.k.a. feature selection) is one of the most important representation space modifications, especially when the original space is very large. For optimization problems with tens or hundreds of attributes, most of the

attributes are not included in hypotheses describing high-performing candidate solutions. The problem of attribute selection is widely known in machine learning, data mining and knowledge discovery, and other fields. Several workshops, conference sessions, special issues of journals, and books have been dedicated to selection of the most relevant attributes (e.g. Liu and Motoda, 1998; Liu, Stine, and Auslender, 2005). Many of the methods described in the literature apply to selection of the most relevant attributes in LEM, which uses concept learning to distinguish between high- and low-performing candidate solutions.

LEM3 uses the AQ21 learning system, which currently implements two well-known methods of evaluating quality of attributes. These methods are PROMISE, described in (Baim, 1982; Kaufman, 1997), and Gain Ratio, used by the c4.5 decision tree learning program (Quinlan, 1993). AQ21 first computes quality of all attributes, and then removes attributes whose quality normalized to range [0, 1], is below a given threshold.

5.4.5 Selection of Representation

Another important issue in constructive induction, both in concept learning and the learnable evolution, is how to detect that a transformed representation space is better than the original one. Methods presented in (Wnek and Michalski, 1994; Bloedorn and Michalski, 1998) use predictive accuracy as the main criterion for testing quality of representation spaces. The training dataset is split into primary and secondary sets, the first of which is used for learning, and the latter for testing. If predictive accuracy on the secondary set is above a given threshold, the new representation is accepted. Although

the accuracy is a very important criterion, it cannot be used as the only one. A slight gain in accuracy may cause significant increase in complexity of learned hypotheses (i.e. when very complex constructed attributes are used), or a modification of representation space may cause significant decrease in complexity, while accuracy does not change. The latter is illustrated in Figure 5-10 where the hypothesis learned in the original representation space is 100% accurate, but requires 4 rules (left), while a hypotheses that uses a constructed attribute $Y = X \pm d$, where d is half the thickness of the stripe on the Figure, requires only one rule, and is also 100% accurate. A similar situation is found when optimizing the Rosenbrock function, whose optimum lies on a narrow ridge.

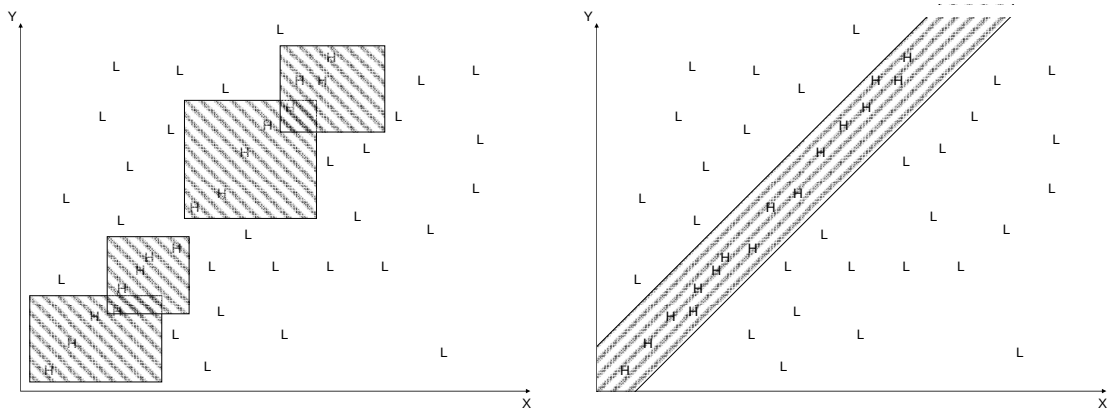


Figure 5-10: An illustration of improved simplicity when transforming the representation space.

The constructive induction module in AQ21 seeks representations in which both accuracy and simplicity are as maximized. A new representation space is accepted if both accuracy and complexity are not worse than in the original representation space and at least one of the two is strongly better than in the original representation. In general, one can define a representation quality measure (i.e. using LEF) that involves more properties of a representation than only accuracy and complexity of learned hypotheses.

5.5 Instantiation of Hypotheses Learned in Transformed Representations

In the learnable evolution model, new candidate solutions are generated by instantiating hypotheses that describe known high-performing solutions in contrast to known low-performing ones. New candidate solutions need to be defined in the original optimization space in order to calculate their fitness. Hypotheses in the form of attributional rules learned in the original representation space can be directly instantiated using one of the methods described in Chapter 3. Instantiation of hypotheses learned in modified representation spaces is a more complicated process, as it may require generating attribute values from attributional conditions with constructed attributes (e.g. equations, discretized attributes, special functions). This section describes methods for creating new candidate solutions from hypotheses learned in specific types of transformed representation spaces.

5.5.1 Instantiation of Discretized Attributes

Each value from the symbolic domain of a discretized continuous attribute corresponds to an interval in the domain of the original attribute. These intervals are disjoint and their union covers the entire domain of the original attribute. Because candidate solutions consist of values of the original attributes, the instantiation process has to translate values of discretized attributes into corresponding original attributes. This process may vary depending on the method used to create the discretized attributes.

To describe the process of instantiation of conditions with discretized attributes, suppose, that A is the original continuous (e.g. ratio) attribute, with domain [lb, ub], and Ad is an

attribute created by discretizing A into intervals $i_1, i_2, \ldots, i_n$. Thus the domain of Ad is $D(Ad) = \{i_1, i_2, \ldots, i_n\}$. Suppose now that a learning program induced a rule with condition given by (5-7) where $i_k = (lb_k, ub_k)$ and $i_l = (lb_l, ub_l)$ are elements of D(Ad).

$$[A' = i_k .. i_l] \tag{5-7}$$

The condition (5-7) is equivalent to (5-8) expressed in the original representation space.

$$[A = lb_k .. ub_l] \tag{5-8}$$

Thus, the condition (5-7) can be instantiated either by selecting a discrete interval from the condition and selecting a value from that interval, or by instantiating condition (5-8), producing a value of A in the original space. Even if at both stages uniform distributions are used, these instantiation methods are not equivalent.

When adaptive anchoring discretization is used, candidate solutions are allowed to have only values which are anchor points, so each value from D(A) is replaced with its corresponding anchor point, as defined in Section 5.4.3. This anchoring point may be found directly by selecting its corresponding interval when instantiating a condition, or by discretization of a selected real value from D(A).

5.5.2 Rejection of Unsatisfied Conditions with Constructed Attributes

The simplest method of instantiating hypotheses learned in modified representation spaces is in spirit similar to the rejection method for handling constrained optimization problems. The program creates new candidate solutions using all attributes from the original representation space and checks if conditions with constructed attributes are satisfied. If they are satisfied, the new candidate solution is accepted and added to the set

of new candidate solutions; otherwise it is rejected, and another candidate solution is generated. Because conditions with constructed attributes can be treated as constraints, Theorem 1 presented in Chapter 4 guarantees existence of solutions that satisfy the rule.

For attributes not present in regular conditions, but present in definitions of constructed attributes used in the instantiated rule, values need to be selected from their entire domains, not from existing individuals, in order to keep proper generalization of the rule. Otherwise, the program may generate only duplicates of existing candidate solutions, without actually exploring new parts of the search space defined by the constructed conditions. For attributes that are neither directly nor indirectly present in the rule, one of the instantiation methods described in Chapter 3 can be used. The instantiation algorithm incorporating constructed attributes is presented in Figure 5-11.

```
Determine the number, n, of candidate solutions to be created
Create list, L, of attributes that are present in definitions of
   constructed attributes
While number of created solutions is smaller than n
      Create new candidate solution s
      Assign random values to all attributes from L in s
      For each regular condition c in R
            Assign value in s according to c
      For each attribute A in s without assigned value
            Instantiate A using one of the available methods
      Match s against conditions with constructed attributes in R
      If matched
            Insert s into the set of created solutions
      Else
            Drop the solution s
Return created solutions
```

Figure 5-11: Instantiation algorithm for rules with constructed attributes.

The presented algorithm tends not to be very efficient, especially when domains of attributes used in conditions with constructed attributes are large. According to

experimental results, however, the results obtained from this method are very good in terms of achieved solutions after a given number of fitness evaluations. Details of the experimental evaluation are described in Chapter 6.

5.5.3 Instantiation of Conditions with Constructed Attributes

New attributes created by the algorithm presented in Figure 5-9 are in the instantiable form (4-6), and therefore their values can be computed. Because the constructed attributes in learned hypotheses can be interpreted as Type 3 constraints, the algorithm for handling instantiable constraints (Figure 4-3) can be applied. As described in Chapter 4, the algorithm checks one constraint at a time, and uses backtracking when a constraint is not satisfied. Specifically, AQ learning systems treat constructed attributes as regular numeric attributes creating conditions in the form (5-9).

$$[\text{CA rel VALS}] \tag{5-9}$$

Here, CA is a constructed attribute in the form (5-5), VALS is either a value, or a range of values, and rel is $<$, $>$, $\leq$, $\geq$, or $=$. The (5-9) can be rewritten as (5-10).

$$[\text{ATT +/- EXPR rel VALS}] \tag{5-10}$$

By selecting a random value v from the VALS, the condition (5-10) can be rewritten as (5-11), where "+" and "-" operators are appropriately changed.

$$[\text{ATT rel v +/- EXPR}] \tag{5-11}$$

Values of all attributes in the expression EXPR can be instantiated from the previous conditions, or their domains, therefore v +/- EXPR can be effectively computed. This transforms (5-11) into a form that is instantiable by methods discussed in Chapter 3.

According to experimental study, this algorithm tends to be more efficient than one that checks all constraints simultaneously, as was presented in the previous section. A detailed experimental comparison of these algorithms is presented in Chapter 6.

5.6 Controlling the Search for the Best Representation Space

The previous sections described how to improve representation spaces, how to create new candidate solutions from hypotheses learned in the modified representations, and related issues. Another important issue, discussed in this section, is when to apply a search for the best representation space. The search can be performed in each LEM iteration, in every *n* iterations, whenever the current representation is inadequate, or only once for the entire optimization process. Selection of the method should depend on the optimization problem and the types of representation modifications to be performed.

The search for the best representation space can be performed by a learning method, such as AQ21, which is equipped with data-driven constructive induction. In such a case, it is convenient to invoke the representation space modification operator in all iterations. This method is implemented in the LEM3 system and used in experimental evaluation presented in Chapters 6 and 7.

5.7 Conclusion

Learning accurate and useful knowledge from data requires an adequate representation space. In the fields of machine learning, data mining, and statistical learning, a significant amount of research has been conducted to study representation-related issues.

Different methods of constructive induction methods have been developed in machine learning to automatically improve representation spaces by selecting the most relevant attributes, discretizing numerical attributes, and constructing new attributes.

Because the learnable evolution model uses machine learning to hypothesize why some candidate solutions perform better than others, constructive induction methods can be applied to improve the representation space used for learning. This process, however, needs to satisfy an additional criterion, namely, the instantiation operator in the learnable evolution model needs to be able to efficiently create new candidate solutions that satisfy the learned hypotheses. This implies a need for either explicit reverse transformation that transforms the modified representation spaces back to the original forms, or an efficient method for creating solutions based on hypotheses learned in the modified spaces.

An important observation used in this research is that methods for handling constrained optimization problems can be applied to the process of instantiating hypotheses learned in modified representation spaces. Because of that, the instantiation methods presented in this chapter are based on methods discussed in Chapter 4.

CHAPTER 6 EXPERIMENTAL EVALUATION

This chapter presents experimental evaluation of the learnable evolution model, methods of handling constrained optimization problems, and automatic improvement of representation spaces, discussed in Chapters 3 - 5. Because constrained and non-constrained optimizations require different sets of testing problems, this chapter is split accordingly into two parts. The goal of the experimental evaluation is to compare LEM3's performance with different sets of parameters and to compare its performance with selected other methods. Because of that, a set of popular testing problems has been selected. The results presented in this chapter along with previous ones described by Wojtusiak and Michalski (2005), illustrate very good LEM3's performance when compared to other evolutionary computation methods. In addition to good performance on numerical optimization, LEM3 is applicable to optimization problems where different types of attributes are used. The results presented here include only standard numerical optimization problems whose results are available in the literature and to which other methods can be applied for comparison. An example of an optimization problem that is defined using different types of attributes is presented in Chapter 7.

6.1 Evaluating the Learnable Evolution Model on Non-Constrained Optimization

This section presents results of the application of LEM3 to selected non-constrained test optimization problems. The main goal of this evaluation is to test the effects of different program parameters on the evolutionary process and obtained results, and in particular the role of constructive induction on the evolutionary optimization process. Subsection 6.1.1 presents optimization problems to which LEM3 is applied, Subsection 6.1.2 describes methods of comparing the program's performance, and Subsection 6.1.3 presents and discusses optimization results.

6.1.1 Optimization Problems

This section presents evaluation problems on which the learnable evolution model is evaluated. These problems include functions frequently used in testing evolutionary computation methods, namely, *Rastrigin*, *Griewangk*, *Rosenbrock*, *Sphere*, and *Step*.

Optimizing (minimizing) the *Rastrigin* function is a well-known problem used in testing evolutionary algorithms. As shown in Figure 6-12, the function has a large number of local optima, and one global minimum equal to zero. It is reached when each of the variables equals zero. A general expression of the Rastrigin function is given by (6-1) and its two-variable plot is presented in Figure 6-12.

$$f(x_1,..., x_n) = 10 * n + \sum_{i=1}^{n} (x_i^2 - 10 * \cos(2 * \pi * x_i)) \quad (6\text{-}1)$$

Optimizing (minimizing) the *Griewangk* function is a well-known problem used in testing evolutionary algorithms. The function also has a large number of local optima, and one global minimum equal to zero. It is reached when all the variables equal zero. The domain for all variables in the performed experiments was [-5.12, 5.12]. The general n-dimensional Griewangk function is given by the expression (6-2).

$$f(x_1,\ldots,x_n) = 1 + \sum_{i=1}^{n} \frac{x_i}{4000} - \prod_{i=1}^{n} \cos(x_1/\sqrt{i}) \tag{6-2}$$

A plot of its 2 dimensional case is presented in Figure 6-13.

Optimizing (minimizing) the *Rosenbrock* function is a well-known problem used in testing evolutionary algorithms. The function has one global minimum reached when values of all attributes equal one. The Rosenbrock function is a hard optimization problem due to the high correlation of variables and the almost flat ridge on which the optimum is located. The function is given by the equation (6-3), and a plot of its 2-dimensional case is presented in Figure 6-26.

$$f(x_1,\ldots,x_n) = \sum_{i=1}^{n-1} (100*(x_{i+1} - x_i^2)^2 + (x_i - 1)^2) \tag{6-3}$$

Optimizing (minimizing) the *Sphere* function is a well-known problem used in testing evolutionary algorithms. The function is relatively simple, as it has a smooth surface and one optimum. Its minimum is reached when all the variables equal zero. The domain for all variables in the performed experiments was [-5.12, 5.12]. The general n-dimensional Sphere function is given by the expression (6-4) and a plot of its 2 dimensional case is presented in Figure 6-15.

$$f(x_1,\ldots,\ x_n) = \sum_{i=1}^{n} x_i^2 \tag{6-4}$$

Optimizing (maximizing) the *Step* function is a well-known problem used in testing evolutionary algorithms. The function is very hard for many numerical optimization methods because it is not continuous. Its global maximum is reached when all the variables equal their maximum allowed value (here values at least five). The domain for all variables in the performed experiments was [-5.12, 5.12]. The general n-dimensional Step function is given by the expression (6-5) and a plot of its 2 dimensional case is presented in Figure 6-16.

$$f(x_1,\ldots,\ x_n) = \sum_{i=1}^{n} floor\ (x_i) \tag{6-5}$$

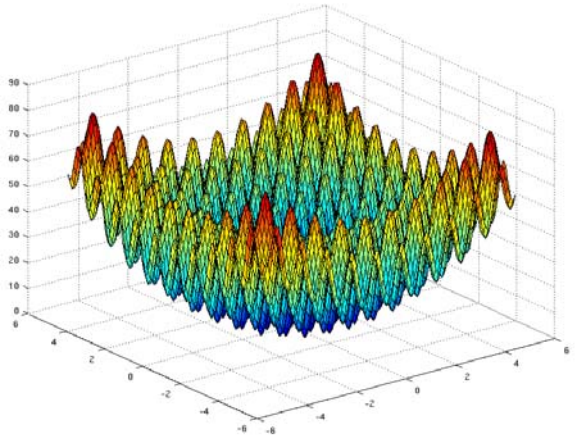


Figure 6-12: The Rastrigin function of 2 variables.

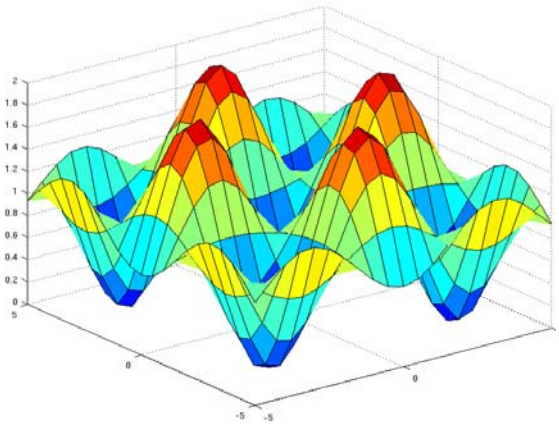


Figure 6-13: The Griewangk function of 2 variables.

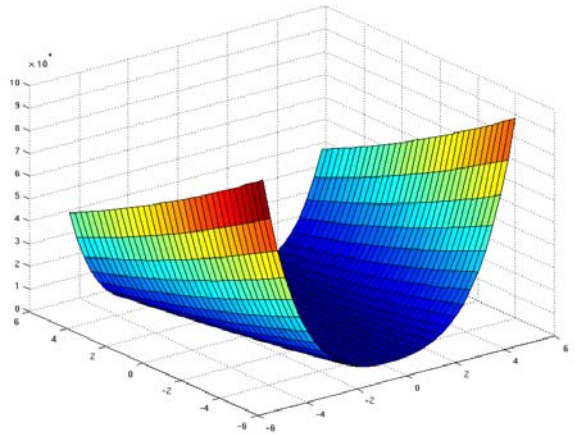


Figure 6-14: The Rosenbrock function of 2 variables.

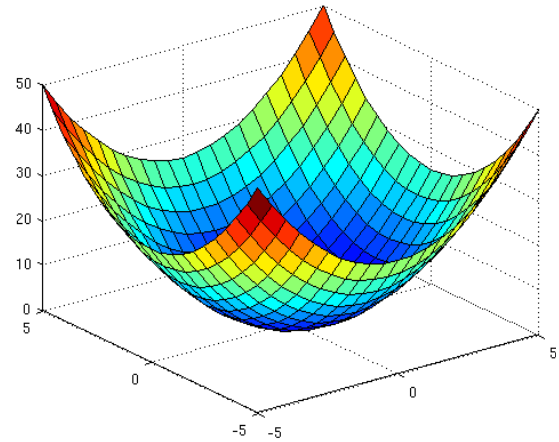


Figure 6-15: The Sphere function of 2 variables.

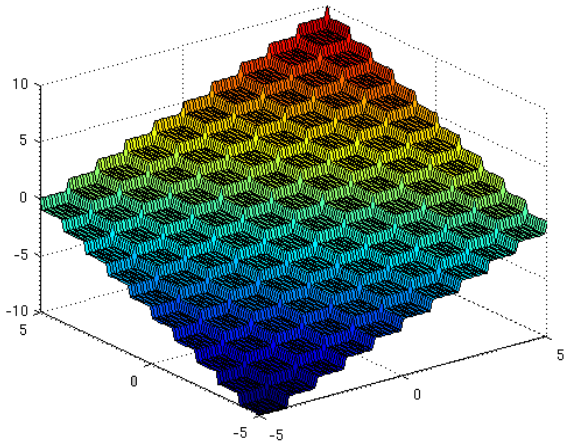


Figure 6-16: The Step function of 2 variables.

6.1.2 Evaluating Results

There are many possible methods of reporting results of testing optimization methods. The most common are to report the best result obtained after a given number of fitness evaluations (or generations), or to report the number of fitness evaluations needed to achieve a given solution. Other possibilities include computation time needed to achieve a given solution on a given computer, the number of fitness evaluations needed to achieve a given improvement, and so forth.

In the latter method the results are reported for *δ-close solutions* that are characterized by a normalized distance from the optimal solution (Michalski, 2000; Wojtusiak and Michalski, 2005; 2006). This method can be used to evaluate performance on test problems to which solutions are known. The δ-close solution, s, is a solution for which function δ(s), defined as (6-6) reaches an assumed δ-target value, where init is the evaluation (fitness value) of the best solution in the initial population, opt is the optimal value, and v(s) is the evaluation of the solution s.

$$\delta(s) = \frac{|opt - v(s)|}{|opt - init|} \tag{6-6}$$

Such a measure works for both maximization and minimization problems, that is, for problems in which the optimal solution has maximal or minimal evaluation.

This definition of δ-close solution suggests two possible ways of analyzing performance of evolutionary computation methods. First, one may consider the problem of how many fitness function evaluations are needed to achieve a given δ=k by the best individual in the population, denoted as FE(δ=k), where k is a number between 0 and 1. Secondly, one

may consider the problem of finding δ(v) after given number of fitness evaluations, where v is the fitness value of the best individual after a given number of fitness function evaluations. The latter is the primary way of reporting results in this dissertation. Figure 6-17 illustrates concept of the δ-close solution.

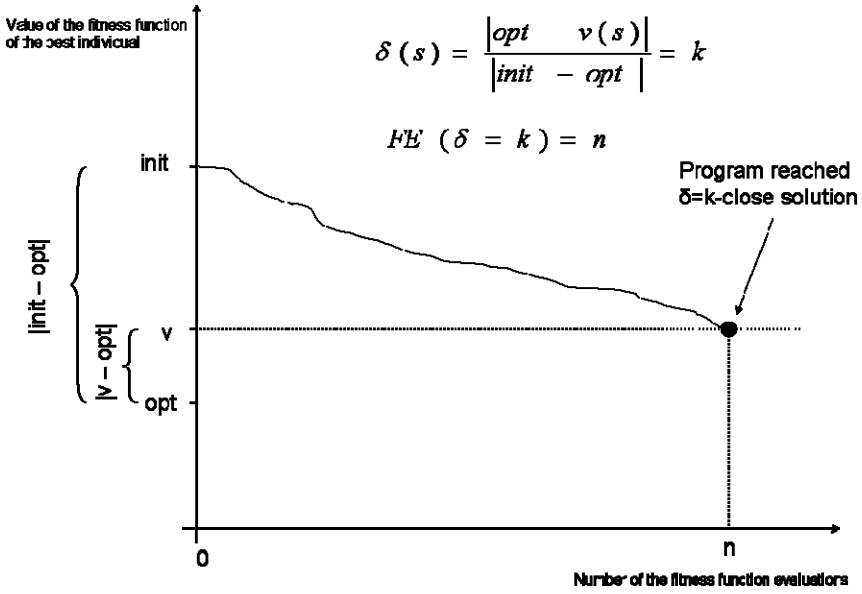


Figure 6-17: Illustration of a δ-close solution (from Wojtusiak and Michalski, 2005).

For example if the fitness value of the best individual in the initial population is 100 and during the process of minimization the program achieved value 0.1, and the optimal value is 0 then δ=0.001, indicating that program found a solution within 0.1% distance from the optimal solution, normalized by the fitness value of the best individual in the initial population.

6.1.3 Results

To compare performance of the learnable evolution model with and without constructive induction, it has been applied to the test problems described in Section 6.1.1. The Table 6-2 presents parameter settings of LEM3 used in these experiments. Because of the very large number of performed experiments (about 700 combinations of parameter settings, each repeated 10 times), the presented results are aggregated. They are grouped by numbers of attributes, optimization problems, and methods for improving representation spaces. The main goal of these experiments is to compare performance of LEM3 with and without automatic improvement of representation spaces. The results are presented in terms of the average $\delta(v)$ values (see Section 6.1.2) obtained after 100 generations of LEM3's evolutions. The results grouped by the number of attributes are presented in Table 6-3, and the results grouped by optimization problem are presented in Table 6-4. LEM3 equipped with automated improvement of representation spaces by constructive induction gave on average better results than LEM3 without constructive induction. However, it can be observed from the tables that the advantage of LEM3 equipped with constructive induction tends to diminish with increasing numbers of attributes. The average result for 100 variables is better for LEM3 without constructive induction than for LEM3 that improves representation spaces. This fact can be attributed to the used method of improving the representation spaces, which is unable to create new attributes that include many attributes from the original spaces. Thus, the method is unable to capture complex numerical relationships involving many attributes.

Table 6-2: List of parameters used in experimental evaluation.

Parameter Name	Used Values	Description
Population size	30, 50, 100	The number of candidate solutions in one population.
Number of children	30, 50, 100	The number of candidate solutions created in each iteration.
Adaptive anchoring discretization	Yes, No	If yes, adaptive anchoring discretization is used to improve the representation of candidate solutions. If no, pure continuous attributes is used.
Data-driven constructive induction (DCI)	Yes, No	If yes, data-driven constructive induction is applied to improve the representation space. If no, only the original attributes are used in learning.
Method of instantiating constructed attributes	Rejection, Instantiation	Defines the method in which rules with constructed attributes are instantiated. Rejection means that all non-constructed attributes are instantiated, and then the conditions with constructed ones are checked (similarly as constraints). Instantiation means that the constructed attributes are evaluated and instantiated along with regular attributes.

Another important issue is that the advantage of LEM3 equipped with the methods for improving representation space depends on the optimization problem to which it is applied. From the results in Table 6-4, it is clear that using constructive induction is not appropriate for the Step function. This can be attributed to the fact that there is no correlation between attributes, and it is sufficient to learn rules with conditions in the form [X > a], where X is an attribute and a is a value. Also for the Griewangk function LEM3 without constructive induction tends to perform in average slightly better than LEM3 with that feature enabled.

Table 6-3: Average δ(v) values after 100 generations for different numbers of attributes for Griewangk, Rastrigin, Rosenbrock, and Sphere functions.

Number of Attributes	No DCI	DCI with Instantiation	DCI with Rejection
2	0.021942	0.019636	**0.011148**
4	0.012114	**0.009134**	0.010078
10	0.050666	**0.048895**	0.058637
50	0.213508	**0.208315**	0.215043
100	**0.277329**	0.295008	0.280424
average	0.115112	0.116198	**0.115066**

Table 6-4: Average δ(v) values after 100 generations for the Griewangk, Rastrigin, Rosenbrock, Sphere, and Step functions with 2, 4, 10, 50 and 100 attributes.

Function	No DCI	DCI with Instantiation	DCI with Rejection
Griewangk	**0.260521**	0.270987	0.260965
Rastrigin	0.134349	**0.130729**	0.133836
Rosenbrock	0.019625	**0.018622**	0.018889
Sphere	0.045952	**0.044452**	0.046573
Step	**0.09915**	0.104438	0.11328

Table 6-5: Average δ(v) values after 100 generations for the Griewangk, Rastrigin, Rosenbrock, Sphere, and Step functions with 2 attributes.

Function	No CI	CI with Instantiation	CI with Rejection
Griewangk	0.076674	0.071342	**0.043669**
Rastrigin	0.007828	0.004404	**0.000566**
Rosenbrock	0.003236	0.002796	**0.000355**
Sphere	2.8E-05	1.73E-06	**9.57E-07**
Step	0.01	0.033529	**0**

Detailed results for two attributes, for which the used method of constructive induction gives very significant improvement of LEM's performance, are presented in Table 6-5.

6.2 Evaluating the Learnable Evolution Model on Constrained Optimization

This section presents an experimental evaluation of the learnable evolution model on selected constrained optimization problems. Methods for handling constraints implemented in the LEM3 system are compared in terms of their performance. Additionally the results are compared with those obtained by two of the methods that won the CEC 2006 competition on constrained optimization (Liang et al., 2005).

6.2.1 Constrained Optimization Problems

Problems used in this evaluation were selected from the list of twenty four problems used at the competition organized during 2006 Congress on Evolutionary Computation. The complete list of problems used for the competition has been published as a technical report by Liang et al. (2005).

The *G1* function (Floudas and Pardalos, 1987; Michalewicz and Schoenauer, 1996) is given by the formula (6-7) and constrained by (6-8). The domain of attributes x_1, x_2, …, x_9, x_{13} is the interval [0,1], and the domain of attributes x_{10} x_{11}, and x_{12} is [0, 100]. The function reaches its maximum $G1(x^*) = -15$ for $x^*=(1,1,1,1,1,1,1,1,1,3,3,3,1)$.

$$G_1(x) = 5\sum_{i=1}^{4} x_i - 5\sum_{i=1}^{5} x_i^2 - \sum_{i=5}^{13} x_i \tag{6-7}$$

$$\begin{aligned} g_1(x) &= 2x_1 + 2x_2 + x_{10} + x_{11} - 10 \le 0 \\ g_2(x) &= 2x_1 + 2x_3 + x_{10} + x_{12} - 10 \le 0 \\ g_3(x) &= 2x_2 + 2x_3 + x_{11} + x_{12} - 10 \le 0 \\ g_4(x) &= -8x_1 + x_{10} \le 0 \\ g_5(x) &= -8x_2 + x_{11} \le 0 \\ g_6(x) &= -8x_3 + x_{12} \le 0 \\ g_7(x) &= -2x_4 - x_5 + x_{10} \le 0 \\ g_8(x) &= -2x_6 - x_7 + x_{11} \le 0 \\ g_9(x) &= -2x_8 - x_9 + x_{12} \le 0 \end{aligned} \tag{6-8}$$

The *G12* function (Koziel and Michalewicz, 1999) is given by (6-9) and constrained by (6-10), where $D(x_i) = [0, 10]$, i=1,2,3, and p,q,r=1,2,..,9. A candidate solution is feasible if there exists a combination of p, q, and r such that (6-10) is satisfied.

$$G_{12}(x) = -(100 - (x_1 - 5)^2 - (x_2 - 5)^2 - (x_2 - 5)^2)/100 \tag{6-9}$$

$$G(x) = (x_1 - p)^2 - (x_2 - q)^2 - (x_2 - r)^2 - 0.0625 \le 0 \tag{6-10}$$

The *G19* function (Himmelblau, 1972) is given by (6-11) and constrained by (6-12), where $D(x_i) = [0, 10]$, i=1,…15. Values of a, b, c, and d are in (Liang et al. 2005).

$$G_{19}(x) = \sum_{j=1}^{5}\sum_{i=1}^{5} c_{ij} x_{(10+i)} x_{10+j} + 2\sum_{j=1}^{5} d_j x_{(10+j)}^3 - \sum_{i=1}^{10} b_i x_i \tag{6-11}$$

$$g_j(x) = -2\sum_{i=1}^{5} c_{ij} x_{(10+i)} - 3d_j x_{(10+j)}^2 - e_j + \sum_{i=1}^{10} a_{ij} x_i \le 0, \text{ j=1,..5} \tag{6-12}$$

The *G24* function (Floudas, 1999) is given by (6-13) and constrained by (6-14), where $D(x_1) = [0, 3]$ and $D(x_2) = [0, 4]$.

$$G_{24}(x) = - x_1 - x_2 \tag{6-13}$$

$$\begin{aligned} g_1(x) &= -2x_1^4 + 8x_1^3 - 8x_1^2 + x_2 - 2 \leq 0 \\ g_2(x) &= -4x_1^4 + 32x_1^3 - 88x_1^2 + 96x_1 + x_2 - 26 \leq 0 \end{aligned} \tag{6-14}$$

6.2.2 Results of the Experimental Evaluation

The presented results are reported in terms of the best obtained results, numbers of fitness function evaluations, and numbers of constraint evaluations. Three methods of handling constrained optimization problems described in Chapter 4 are compared: trimming, learning approximations, and using infeasible solutions as negative examples for learning. Other controlled parameters include population sizes and numbers of instantiated candidate solutions, adaptive anchoring discretization, and use of constructive induction.

Except for few cases, the LEM3 system with the three described methods for handling constraints achieved very similar fitness values after a given number of generations. There are, however, differences in the numbers of infeasible solutions generated while executing the program with the three methods. For all studied problems, the simple trimming method that uses only feasible solutions in the learning process, required testing the largest number of infeasible candidate solutions. The normalized average numbers of infeasible solutions are illustrated in Figure 6-18. For each tested function, the average numbers of generated infeasible solutions were normalized into the range [0, 1],

separately for experiments with continuous and adaptively discretized attributes. Finally, averages over all testing problems and different parameters' settings were computed.

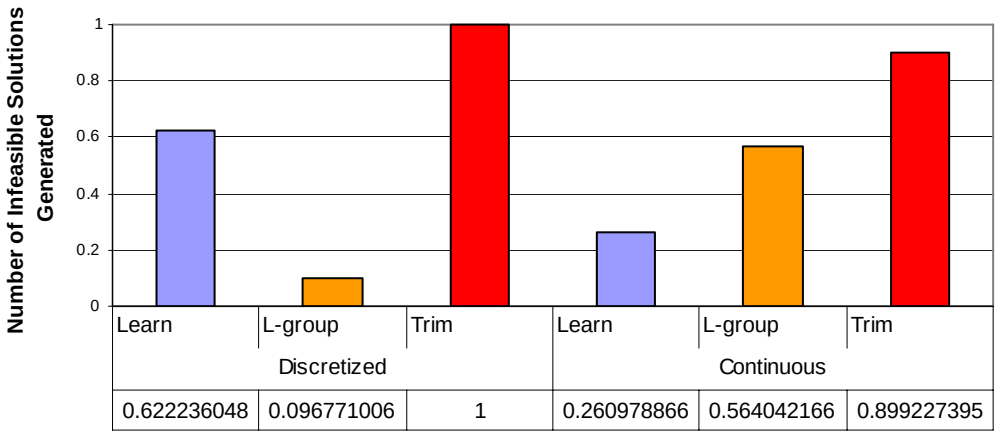


Figure 6-18: The average normalized numbers of infeasible candidate solutions generated during 100 generations of LEM3 execution on the G1, G12, G19, and G24 functions.

The performance of learning of feasible space approximation and using unfeasible solutions as a contrast set depends on the use of discretization. For discretized attributes, using infeasible solutions as a contrast set significantly outperforms other methods, while for pure continuous attributes approximation of the feasible space scores the best.

In order to compare LEM3's performance on the four selected constraint optimization problems, two other methods were selected. The methods scored the best during CEC 2006 competition on constrained optimization. The first method, εDE, applies the ε constrained method to differential evolution (Takahama and Sakai, 2006), and the second method combines DMS-PSO dynamic multi-swarm optimization, with SQP sequential quadratic programming (Liang and Suganthan, 2006). Other results are reported at the

competition website, and results of applying all five accepted methods are included in the conference proceedings. Following the format of results published at the conference, the results are reported in the form of errors from the actual solutions and numbers of violated constraints after 5,000 fitness evaluations. Additionally, LEM3 reports the total number of infeasible solutions generated. LEM3 is able to reject infeasible solutions without computing their fitnesses, while for both compared methods, the number of fitness evaluations is always equal to the number of constraint evaluations. In the presented experiments, LEM3 was executed with and without adaptive discretization of continuous attributes. The results also were compared for LEM3 starting with and without a population of feasible candidate solutions.

Results of the comparison are summarized in Table 6-6. For all four considered functions LEM3 achieved better results in terms of the value of the fitness function. However, in some cases, when executed with feasible starting population, LEM3 required very large number of constraints evaluations as it generated many infeasible solutions. On the other hand, LEM3 achieved also very good results when starting without known feasible individuals, and produced significantly fewer rejected infeasible solutions (below 800 in all experiments per 5,000 fitness evaluations). This result indicates an important issue to be considered in constrained optimization. For many constrained problems the fitness function is defined only for feasible candidate solutions, for example when constraints define a set of parameters under which a given simulator can be correctly executed. In such cases starting with no feasible solutions is not possible and neither the two compared methods can be applied.

Table 6-6: Comparison of errors (E) for εDE, DMS-PSO + SQP, and LEM3 on G1, G12, G19, and G24 functions after 5000 fitness evaluations. For LEM3, the number of infeasible solutions (I) and for DMS-PSO + SQP the number of violated constraints (C), are also reported.

Method		G1	G12	G19	G24
εDE	E	10.536	0.00001230	963.18	0.00000024962
DMS-PSO + SQP	E	9.0738	0.0025467	367.1	0.016628
	C	3	0	0	0
LEM3 + ANCHOR	E	0	0	179.369407	1.12801327
	I	9998	5650	2,924,828	122
LEM3 Continuous	E	0.1271483551	0.000000007012	6.0786283386537	0.00000000006536
	I	2,176,968	5112	529,381	71013
NF LEM3 ANCHOR	E	0	0	181.469407	1.12801327
	I	681	612	643	133
NF LEM3 Continuous	E	1.2970298	0.000001491556	114.772007	0.01910691393536
	I	686	712	739	787

6.3 Conclusions

This chapter presented experimental testing of the learnable evolution model in the LEM3 system on selected well known benchmark optimization problems. These problems included constrained and non-constrained functions to which LEM3 was applied with different settings of parameters. The goal of the experimental evaluation was to test effects of automated improvement of representation spaces and different methods for handling constraints on the evolutionary process and results of optimization.

An important result is that the provided method for modifying representation spaces consistently improved LEM3's performance. This improvement, however, tends to diminish with growing numbers of attributes. This may be due to use of a DCI-based method for automated improvement of representation space that is unable to construct attributes involving many original attributes – the search space is too large. Two potential solutions to this problem include (1) using a different method for modifying

representation, and (2) using background knowledge to narrow the attribute search space. A method (1) for modifying the representation space should be able to construct new attributes that include many original attributes. For example, principal component analysis, which is widely used in statistics (e.g., Hotelling, 1933; Gentle, 2002), constructs new dimensions that may include all original attributes at the same time. The possible solution (2) is to use background knowledge by suggesting possible transformations of the representation space in the form of advices in order to narrow the search for new attributes.

Another important result is that the presented experimental study confirmed the hypothesis that methods for handling general constraints, described in Chapter 4, reduce the total number of generated infeasible solutions. For some problems, that decrease is significant. However, different methods show the best performance on different problems. More importantly, the comparison of LEM3 with two top-ranked programs for constrained optimization showed its advantage. In all tested cases, LEM3 was able to find more accurate feasible solutions after 5,000 fitness function evaluations.

CHAPTER 7 OPTIMIZATION OF PARAMETERS OF COMPLEX SYSTEMS WITH APPLICATIONS IN MEDICINE

Results presented in the previous chapters suggest that LEM is particularly suitable for optimization problems in which evaluation of the fitness function is complex (e.g. requires running a simulator) and to problems represented using different types of attributes. One such a problem is optimization of parameters of complex systems or programs. Two applications investigated in this chapter are: optimizing parameters of the AQ21 machine learning program, and optimizing discretization of continuous variables. Both applications are investigated in context of real world medical datasets.

7.1 Optimization of AQ21 Parameters on Selected Medical Datasets

AQ21 is a complex machine learning system which can be controlled by several parameters, some of which are described in Chapter 2. Depending on the application, different settings of parameters may lead to the best solutions. For large datasets consisting of hundreds or thousands of examples, the learning process takes a considerable amount of time (sometimes in order of hours for very large problems). The large number of parameters controlling AQ21, include both numerical and are symbolic parameters. The space of possible parameter settings consists of multi-type attributes.

There are also several constraints which define impossible combinations of parameters or combinations that don't make sense.

7.1.1 Representation Space

The representation space spans all possible combinations of AQ21's parameters chosen to be optimized. Each of AQ21's parameters defines one attribute in the representation, and possible values of the parameter constitute a domain of the corresponding attribute. The complete list of attributes defined from AQ21's parameters used in this study and their descriptions are presented in Table 7-7.

The space consists of 24 attributes (12 nominal, 10 ratio, and 2 absolute). Assuming that initially the numeric attributes are discretized by adaptive anchoring discretization into 10 ranges, the total size of the space is given by (7-1).

$$3 \times 10 \times 3 \times 10 \times 10 \times 30 \times 2 \times (14 \times 10)^6 \times 2 \times 8 \times 4 \times 10 \times 10 \approx \mathbf{2.6 \times 10^{21}} \quad (7\text{-}1)$$

Assuming that a single execution of AQ21 takes in average only one second the exhaustive search over the entire parameters space would take over 8×10^{11} years. This means that checking all combinations of parameters is not possible.

Table 7-8 presents the problem constraints that define correct combinations of AQ21's parameters. Even significantly smaller search space after introducing constraints is too large for performing the full search. Descriptions of the used parameters and relationships between them are described by Wojtusiak (2004a) in AQ21 User's Guide, and by Wojtusiak et al. (2006a,b).

Table 7-7: Representation space for the problem of optimizing AQ21 parameters.

Attribute Name	Type	Domain	Description
Mode	*Nominal*	TF, ATF, PD	Mode of operation: theory formation, approximate theory formation, and pattern discovery.
W	*Ratio*	[0, 1]	Completeness vs confidence gain weight.
Attribute Selection Method	*Nominal*	Promise, Gain Ratio, None	Method of selecting attributes. None indicates no selection – all attributes are used for learning.
Attribute Selection Threshold	*Ratio*	[0, 1]	Threshold of acceptance of attributes evaluated using the above method.
Maxstar	*Absolute*	1, …, 10	Number of rules selected from a partial star during learning process.
Maxrule	*Absolute*	1, …, 30	Number of rules selected from a star during learning.
Exceptions	*Nominal*	True, false	Invokes algorithm for learning rules with exception clauses.
LEF_ps_c1	*Nominal*	MinNumSelectors, MaxNumSelectors, MaxNewPositives, MaxUniquePositives, MaxPositives, MinNegatives, MaxQ, MinCost, MaxConfidence, MaxNewPositivesQ, MinComplexity, MaxGenerality, MaxSignificance, GainRatio	The first LEF criterion for selecting rules from partial stars.
LEF_ps_t1	*Ratio*	[0, 1]	Tolerance for the first criterion for selecting rules from partial stars.
LEF_ps_c2	*Nominal*	Same as LEF_ps_c1	The second LEF criterion for selecting rules from partial stars.
LEF_ps_t2	*Ratio*	[0, 1]	Tolerance for the second criterion for selecting rules from partial stars.
LEF_star_c1	*Nominal*	Same as LEF_ps_c1	The first LEF criterion for selecting rules from stars.
LEF_star_t1	*Ratio*	[0, 1]	Tolerance for the first criterion for selecting rules from stars.
LEF_star_c2	*Nominal*	Same as LEF_ps_c1	The second LEF criterion for selecting rules from stars.

LEF_star_t2	*Ratio*	[0, 1]	Tolerance for the second criterion for selecting rules from stars.
LEF_trunc_c1	*Nominal*	Same as LEF_ps_c1	The first LEF criterion for selecting the final stars.
LEF_trunc_t1	*Ratio*	[0, 1]	Tolerance for the first criterion for selecting the final rules.
LEF_trunc_c2	*Nominal*	Same as LEF_ps_c1	The second LEF criterion for selecting the final rules.
LEF_trunc_t2	*Ratio*	[0, 1]	Tolerance for the second criterion for selecting the final rules.
Evaluation of Selector	*Nominal*	Flexible, strict	Interpretation of selectors during testing.
Evaluation of Conjunction	*Nominal*	Strict, coverage ratio, selectors ratio, minimum, weighted minimum, product, average, weighted average	Interpretation of conjunction of selectors during testing.
Evaluation of Disjunction	*Nominal*	Average, probabilistic sum, maximum, best only	Interpretation of disjunction of rules (rulesets) during testing.
Acceptance Threshold	*Ratio*	[0, 1]	Minimum degree of match of an event needed for classification.
Equivalence Tolerance	*Ratio*	[0, 1]	Tolerance within which degrees of match are considered equivalent.

Table 7-8: Constraints in the problem of optimizing AQ21s parameters.

Constraints	Explanation
[W > 0] & [W < 1]	Extreme values of W do not make sense.
[LEF_ps_c1 ≠ LEF_ps_c2] & [LEF_star_c1 ≠ LEF_star_c2] & [LEF_trunc_c1 ≠ LEF_trunc_c2]	Two criteria used in a LEF should not be the same.
[Mode = TF] → [LEF_star_c1 ≠ MinNegatives v MaxQ v MaxConfidence v MaxNewPositivesQ] & [LEF_star_c2 ≠ MinNegatives v MaxQ v MaxConfidence v MaxNewPositivesQ]	In theory formation mode where each rule is guaranteed not to cover any negative examples criteria that use numbers of negatives, such as Q(W) and confidence, are not applicable to LEF star.
[Evaluation of selector = flexible] → [Evaluation of conjunction = minimum v weighted minimum v product v average v weighted average]	For flexible selector evaluation, only the listed methods of evaluating conjunctions are applicable.
[Evaluation of selector = strict] → [Evaluation of conjunction = strict v selectors ratio v coverage ratio]	For strict selector evaluation, only the listed methods of evaluating conjunctions are applicable.

7.1.2 Optimization Objective

After defining representation space for the problem of optimizing AQ21's parameters, the next step is to define the optimization objective (a fitness function). In this application, the function is based on the accuracy and complexity of the learned hypotheses. To compute accuracy and complexity, it is needed to execute AQ21 and analyze its results. This section presents definition of the fitness function based on the output from the AQ21 system.

The ATEST program briefly described in Chapter 2 reports results of testing differently than most machine learning programs. One of major differences is the assumption that it is better to give a correct answer which may be imprecise than give a precise, but incorrect answer. Consequently, in addition to predictive accuracy ATEST returns a measure of precision of the given answer (e.g. Reinke, 1984; Wojtusiak, 2004a; Wojtusiak et al., 2006a,b). Similarly to the predictive accuracy, the precision is given by a number varying from 0% to 100%, where 0% represents fully imprecise answers (all testing examples are assigned to all classes), and 100% corresponds to precise answers (only one class is assigned for each testing example). Because of the design of the ATEST program, it is not sufficient to maximize the predictive accuracy, but also precision needs to be maximized. Both predictive accuracy and precision are expressed in percents, so the multiplication of the two can be meaningfully used as given by (7-2).

$$\text{Accuracy}(X)^a \times \text{Precision}(X)^b \tag{7-2}$$

Here, X is a set of AQ21's parameters, Accuracy(X) and Precision(X) are predictive accuracy and precision obtained by AQ21 using the parameter setting X, and a and b are

numbers in range [0, 1]. Due to the use of (7-2) the two criteria optimization problem is converted into the single criterion.

In natural induction simplicity of learned knowledge is equally important to its accuracy. AQ21 learning module computes complexity of learned hypotheses by assigning a weight to each operation using formula (7-3) as proposed in attributional calculus by Michalski (2004a). Default values of the coefficients $c_1,..,c_7$ are in (Wojtusiak, 2004a). The complexity of an exception clause is computed as twice the complexity of its elements.

$$\begin{aligned} CX = c_1 * \text{conjunction count} \; + \; c_2 * \text{disjunction count} \; + \\ c_3 * \text{internal disjunction count} + c_4 * \text{range count} \; + \; c_5 * \text{equal count} \; + \\ c_6 * \text{less or greater count} \; + \; c_7 * \text{not equal count} \end{aligned} \tag{7-3}$$

Because examples can be represented in the form (7-4), it is possible to apply the formula (7-3) to compute complexity of the input examples, or simply *input complexity*, as (7-5).

$$(x_1 = v_1, x_2 = v_2, \ldots, x_k = v_k) \tag{7-4}$$

$$ICX = (|P| - 1) * c_2 \; + \; |P| * (k - 1) * c_1 \; + \; |P| * k * c_5 \tag{7-5}$$

Here, |P| is the number of positive examples of a concept and k is the number of input attributes. Based on the input complexity it is possible to define the *simplicity gain* as (7-6) and *normalized simplicity gain* as (7-7).

$$SG = ICX - CX \tag{7-6}$$

$$NSG = (ICX - CX) \; / \; ICX \tag{7-7}$$

Complexity of a cover learned by AQ21 will not be larger than input complexity thus the significance gain is always a positive number. The normalized significance gain is a

number from the range [0, 1]. Multiplication of the normalized significance gain by 100, is also a number from the range [0, 100]. Finally the formula (7-2) can be extended into (7-8) and used as a fitness function.

$$\text{Fitness}(X) = \text{Accuracy}(X)^{a} \times \text{Precision}(X)^{b} \times (100 \times \text{NSG})^{c} \qquad (7\text{-}8)$$

Here, a, b, and c are numbers in range [0, 1]. In the experimental results presented in Section 7.1.4, their values are a=1, b=1, and c=0.5. Using this setting of coefficients the fitness function takes values in the range [0, 100 000).

7.1.3 Medical Datasets

In this study AQ21's parameters are optimized to achieve the best performance on three medical datasets consisting of groups of patients with metabolic syndrome, patients' vitality scores, and relationships between diseases and lifestyles. These datasets represent different types of learning problems, with different numbers of classes, examples, attributes, etc.

The first dataset consists of measurements of different parameters aggregated over different groups of patients, and is referred to as the *metabolic syndrome dataset*. It consists of 20 examples drawn from 4 classes. These classes represent three diseases from the metabolic syndrome spectrum, namely, non-alcoholic fatty liver disease (NAFLD), simple steatosis (SS), and nonalcoholic steatohepatitis (NASH). It also includes a healthy status, represented by a control group serving as a contrast set for learning. NAFLD is the most general condition that comprises both SS and NASH cases,

which means that values of the output attribute form a hierarchy. Each example consists of mean values of different medical parameters measured over a group of patients. Such an aggregated data are not protected by patients' privacy laws, so they are easily available for the study. The total number of 152 input attributes in the dataset has been reduced to 20 most common, due to the fact that different measurements were recorded in different studies (most of the attributes were available only in single studies). The dataset used in this study has been collected from articles published in medical journals such as Hepatology, Obesity Research, International Journal of Obesity and some others (Wojtusiak et al., 2007).

The second dataset consists of measurements of different parameters of a group of patients and their *vitality scores*. The vitality score is a measure of patients' performance computed from answers to the SF-36 form (e.g., see http://www.sf-36.org). Because self-reporting based on the form SF-36 is not convenient for patients, in is important to be able to predict the value of vitality score based on other measurements, preferred by patients. The dataset consists of 43 patients, and more than 50 input attributes from which 11 attributes were selected. This selection was based on expert's decision about their relevance to the problem. All of the selected attributes are numeric. The continuous output attribute representing the vitality score has been discretized into three classes: *low* for values below 40, medium for values between 40 and 55, and high for values above 55. These ranges were based on the expert's decision and consisted of 11, 18, and 14 examples respectively.

The third dataset consists of *lifestyles* and diseases of non-smoking males, aged 50-65. The study employed a database from the American Cancer Society that contained 73,553 records of responses of patients to questions regarding their lifestyles and diseases. Each patient was described in terms of 32 attributes: 7 lifestyle attributes (2 Boolean, 2 numeric, and 3 rank), and 25 Boolean attributes representing diseases. From the original set of examples, 200 examples were randomly chosen for the presented experiments. The problem considered here was to determine rules for classifying prostate cysts based on lifestyles and presence of other diseases.

7.1.4 Results

When applied to the *metabolic syndrome* dataset AQ21 with default parameters gives in average on the 5-fold cross validation 75% predictive accuracy, with precision 100% and complexity 12 (0.99 normalized simplicity gain). The value corresponds to the fitness value 74,624.06. After LEM3 optimization of parameters on the same dataset AQ21 achieved 95% predictive accuracy, 100% precision and complexity 9, which is a significant improvement that gives the fitness value 94,523.81. An interesting result is that out of six LEM3 executions the program achieved this result twice – both times with enabled automatic improvement of representation space by constructive induction.

When applied to the *vitality score* dataset AQ21 with default parameters and 5-fold cross validation achieved averaged predictive accuracy 40%, precision 100%, and complexity 65 (normalized simplicity gain is 87%), which correspond to fitness value of 37,309.52. The fitness of the best obtained result is 77,149.21 giving over two times improvement.

The fitness is computed from the obtained average predictive accuracy 80%, precision 100%, and complexity 33.8 (93% simplicity gain).

An example plot illustrating LEM3's progress on optimizing the AQ21's parameters for the vitality score dataset is presented in Figure 7-19. It shows a singe program's execution with population size 50, number of children 30, and automated improvement of representation space instantiated by rejection.

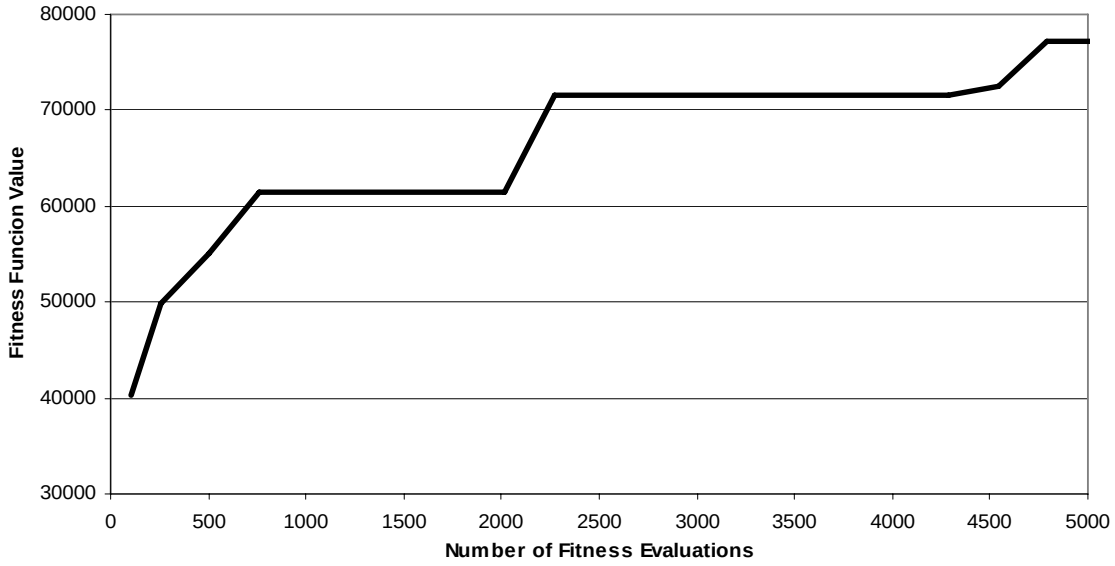


Figure 7-19: Increasing value of the fitness function when optimizing AQ21's parameters for the vitality score dataset.

When applied to the *lifestyles* dataset AQ21 with default parameters and 5-fold cross validation achieved averaged predictive accuracy 57%, precision 99.02%, and complexity 610.4 (normalized simplicity gain is 95%), which correspond to fitness value of 55,012.27. The fitness of the best result found by LEM3 is 79,639.4 (average predictive accuracy 82%, average precision 97.12%, and average complexity 59.40).

7.1.5 Conclusions

The application of LEM3 to optimization of parameters of a complex system, AQ21, gave excellent practical results. LEM3 was able to improve performance, measured in terms of hypotheses' accuracy and simplicity, achieved by the AQ21 system on all three medical datasets. These are important results because:

- The optimization problem consisted of multitype attributes to which many methods cannot be applied.
- The best obtained results were achieved using LEM3 with automated improvement of representation space.
- These results confirm LEM3's applicability to optimization of very complex systems, the example of which is the optimization of AQ21's parameters. Complex systems are rarely described using only numeric attributes, which makes LEM3 particularly suitable for this type of tasks.

7.2 Application to Finding the Best Discretization of Numeric Attributes

Discretization is an abstraction of real values of the domain of a numeric attribute (e.g. ratio, interval) into a finite number of intervals. Proper discretization of numeric attributes is one of the most important steps of data preparation process. Using pure continuous variables in AQ-based learning often leads to overfitting, that is, learning hypotheses which accurately describe training data, but poorly generalize to new examples. On the other hand, too high abstraction into to very few intervals, may lead to loss of important information and introduce ambiguity to a dataset.

Another rationale for discretizing numeric attributes is increase of comprehensibility of learned knowledge. Conditions that include high precision numeric attributes are harder to memorize and understand by humans, while conditions that include not so precise, discretized, attributes tend to be preferable. In many cases discrete values can be additionally replaced with symbolic labels such as *low*, *medium*, *high*, *very high*, to further improve understandability. Because comprehensibility of knowledge is hard to measure numerically (e.g. because it is subjective), it is not directly measured in this study. It is only assumed that people prefer discretized attributes with fewer values.

Most methods described in the literature, such as ChiMerge (Kerber, 1992) and Chi2 (Tay and Shen, 2002), are used to discretize only one variable at once (Dougherty, Kohavi and Sahami, 1995). Some methods use the approach considered in this study that assumes that all numeric attributes are discretized at the same time (e.g., Bay, 2001). An overview and experimental comparison of different discretization methods used in machine learning is presented in (Dougherty, Kohavi and Sahami, 1995). Also, recently introduced field of *granular computing* includes methods for discretizing numeric variables by so called *granulation of variables* (e.g. Bargiela and Pedrycz, 2002).

7.2.1 Optimization Objective

The objective of the presented optimization is to find the best discretization of numeric attributes in a given dataset. This can be formulated as finding the smallest possible number of discrete intervals that give the most accurate and simplest learning results. This can be measured using cross-validation as described in Section 7.1 by finding the

best discretization that maximizes the measure (7-8). For large datasets, however, the fitness evaluation may be very computationally expensive.

Another possibility is to maximize total quality of all numerical attributes being discretized. The quality of a single attribute can be measured using statistical methods such as gain ratio (Quinlan, 1993) or PROMISE (Baim, 1982). The total quality of all numerical attributes (7-9) is a sum of qualities of single attributes.

$$Quality(X_1..X_n) = \sum_{i=1}^{n} Quality(X_i) \tag{7-9}$$

Assuming that quality a single attribute is normalized into [0, 1] range as for example in the case of promise measure, the fitness function takes values between 0 and n, where n is the number of numerical attributes in the given dataset. In this study PROMISE is used to evaluate quality of attributes.

7.2.2 Representation Space

The previous section described the optimization objective (fitness function) for the problem of discretization of numeric attributes. This section discusses representation of candidate solutions (representation space) for that problem.

Suppose that X_1, …X_n are numeric attributes that need to be discretized, and $D(X_1)$, …, $D(X_n)$ are their domains defined by intervals $[lb_1, ub_1]$, … $[lb_n, ub_n]$, respectively. By applying discretization process, each attribute X_i is transformed into its discretized form Xd_i with domain given by (7-10).

$$D(Xd_i) = \{ I_i^1, I_i^2, \ldots, I_i^{ki} \} \tag{7-10}$$

where I_i^j are disjoint intervals given by (7-11) whose union is $D(X_i) = [lb_i, ub_i]$.

$$I_i^j = [lb_i^j, ub_i^j] \tag{7-11}$$

Assuming that the intervals are ordered, we have (7-12), (7-13) and (7-14).

$$ub_i^j = lb_i^{j+1} \quad \text{for } j=1..ki-1 \tag{7-12}$$

$$lb_i^1 = lb_i \tag{7-13}$$

$$ub_i^{ki} = ub_i \tag{7-14}$$

The equations (7-10) to (7-14) imply that to represent discretization of a numeric attribute X_i it is sufficient to store values $lb_i^2, \ldots lb_i^{ki}$, that are borderlines between intervals.

To represent different discretizations in the optimization process, each of the numbers lb_i^j can be represented in the learnable evolution model by one numeric attribute LB_i^j. The attributes LB_i^j must satisfy (7-15) and (7-16).

$$D(LB_i^j) = D(X_i) \tag{7-15}$$

$$lb_i < LB_i^2 < \ldots < LB_i^{ki} < ub_i \tag{7-16}$$

LEM3 implementation of the learnable evolution model requires that all candidate solutions are represented by the same numbers of attributes. This requirement is present because of representation of examples used for learning by the AQ21 system used by LEM3 for hypotheses formulation. Because of that each attribute X_i has an additional constant K_i such that $k_i \leq K_i$ for all candidate solutions. Such a constant can be set for

example to a maximum acceptable number of intervals. Finally, discretization of one attribute can be represented by K_i numbers (7-17).

$$k_i, lb_i^2, \ldots lb_i^{ki}, u_i^{ki+1}, \ldots, u_i^{Ki} \quad (7\text{-}17)$$

The values of $u_i^{ki+1}, \ldots, u_i^{Ki}$ are ignored during evolution and hypotheses formulation. Additionally, when AQ21 learning is applied all values u_i^j are replaced with not applicable meta-value that indicates that the actual values do not exist (Michalski and Wojtusiak, 2005).

Representation of complete candidate solutions is given by (7-18). It includes representation (7-17) of all original numeric attributes being discretized. The total number of attributes used to represent discretizations is $K_1 + K_2 + \ldots K_n$.

$$k_1, lb_1^2, \ldots lb_1^{k1}, u_1^{k1+1}, \ldots, u_1^{K1}, \ldots k_n, lb_n^2, \ldots lb_n^{kn}, u_n^{kn+1}, \ldots, u_n^{Kn} \quad (7\text{-}18)$$

7.2.3 Constraints

Single-objective optimization of discretization of numeric attributes requires definition of several constraints. These constraints are partially caused by the form of representation space described in the previous section, and partially represent desired properties of the discretizations.

Table 7-9: Constraints in the problem of finding optimal discretization.

Constraint	Explanation
$LB_i^2 < \ldots < LB_i^{ki}$, $i = 1..n$	Intervals in discretization must not be overlapping. This constraint is needed because of the representation space used in the optimization. Degree of violation of this constraint is the ratio of the number violated inequalities to the total number of inequalities.
Discretization does not introduce ambiguity	When abstracting numerical values, examples belonging to different classes may become indistinguishable, therefore ambiguity is introduced to the learning process. Such a situation is a violation of this constraint. Degree of the violation of this constraint is the ratio of the total number of examples to the number of examples that become ambiguous.
$\max(\lvert D(DX_i)\rvert) \leq \min(\lvert D(DX_i)\rvert)+\tau$ Sizes of domains of discretized attributes do not differ more than by τ	Avoiding violation of the previous constraint may lead to a situation when some attributes are discretized into a very small number of intervals, and others may have many intervals that are sufficient to distinguish between training examples. To avoid this situation the number of intervals for all discretized attributes must be within a given threshold.

7.2.4 Results

This section describes a summary of experimental results of applying the learnable evolution model to automatic discretization of continuous attributes. In these experiments two previously described medical datasets are used, namely, metabolic syndrome and vitality score. The results are compared with ChiMerge (Kerber, 1992) method executed with different numbers of ranges. The ChiMerge algorithm is implemented in the AQ21 system, thus it is easy to use and compare with results obtained by LEM3.

In the original metabolic syndrome dataset there is no ambiguity (identical examples belonging to different classes), and the total PROMISE of all attributes is 7.75. In the presented experiments LEM3 was started with no known feasible solutions (see Section 4.7). LEM3 obtained result with no ambiguous examples that needed only two intervals for sixteen attributes and three intervals for four attributes in the dataset consisting of twenty attributes total. This result is in contrast with the result obtained by ChiMerge that when finding three intervals per attribute, introduced two ambiguous examples. The total promise of discretized attributes found by LEM3 is also higher as shown in Table 7-10.

Table 7-10: Results of finding the best discretization by LEM3 and ChiMerge on the metabolic syndrome dataset.

Method	Fitness	Ambiguity
Original data	7.75	0
ChiMerge with 2 intervals per attribute	10.964	3
ChiMerge with 3 intervals per attribute	10.597	2
ChiMerge with 4 intervals per attribute	10.2945	0
ChiMerge with 5 intervals per attribute	10.1406	0
ChiMerge with 6 intervals per attribute	9.5049	0
LEM3, no more than 3 intervals per attribute	**12.1609**	0

Table 7-11: Results of finding the best discretization by LEM3 and ChiMerge on the vitality score dataset.

Method	Fitness	Ambiguity
Original data	3.7991	0
Manually discretized	4.02792	0
ChiMerge with 2 intervals per attribute	3.89843	4
ChiMerge with 3 intervals per attribute	4.29631	0
ChiMerge with 4 intervals per attribute	4.34179	0
ChiMerge with 5 intervals per attribute	4.68878	0
ChiMerge with 6 intervals per attribute	4.7587	0
LEM3, no more than 4 intervals per attribute	**5.55234**	0

In the original vitality score dataset there is no ambiguity neither when continuous data are used, nor when the data are manually discretized by an expert. The fitness value (the total PROMISE of all attributes) corresponding to the original continuous data is 3.7991, and for the manually discretized data it is 4.02792. The comparison of results found by the ChiMerge algorithm and LEM3 is presented in Table 7-11.

The presented results were obtained using the LEF-based method for starting with no feasible solutions implemented in LEM3 and described in Section 4.7. The program converged to feasible solutions (i.e. correct discretizations that don't introduce ambiguity), while optimizing the given quality measure.

7.2.5 Conclusions

Application of the learnable evolution model to automatic discretization of numeric attributes shows that it is able to find simple and high performing discrete attributes. Due to the use of constraints, the new attributes do not introduce ambiguity into data when replacing the original continuous attributes. On both metabolic syndrome and vitality score datasets LEM3 found better discretizations than those found by the ChiMerge algorithm. For the vitality score dataset the discretization found by LEM3 also have higher quality than the discretizations manually created by an expert.

CHAPTER 8 CONCLUSIONS

This dissertation presents a new methodology for handling constrained optimization problems and for automatically improving representation spaces in the learnable evolution model. Previous research has shown LEM's excellent performance and applicability to many real world problems. While the fact that most of the real world optimization problems are constrained is well known, the double relation between handling constraints and improvement of the representation space has not been previously investigated. The methods for handling specific types of instantiable constraints can be applied to instantiate hypotheses learned in modified representation spaces in order to improve program's efficiency. On the other hand, improvements of the representation spaces may help to better capture feasible candidate solutions, and reduce the number of infeasible solutions generated. This is particularly important when evaluation of constraints is a time consuming process.

Key components of the described methodology have been implemented in an experimental system LEM3. Experimental evaluation on selected widely known testing problems revealed high potential of the implemented methodology and indicated new areas of research. Also application of the methodology to two real world problems indicated excellent results and provided more evidence about its importance.

8.1 Contributions of the Dissertation

The presented work has contributed to theoretical and practical aspects of the problems of handling constrained optimization problems and automatically improving representation spaces in the learnable evolution model. Specifically, the most important contributions include:

- Classification of constraints into four classes based on the difficulty of handling them in the learnable evolution model. The most important distinction is made between *instantiable* and *general* constraints. This distinction is made by the presence of an efficient method for solving them in the instantiation process.
- Design and implementation of methods for handling instantiable constraints of types 1-3, including these given in the form or ordered conditions [ATT rel EXPR]. Although these types of constraint are very limited, and few optimization problems may have constraints in these forms, they are important for instantiation of conditions with constructed attributes.
- Design and implementation of three methods for handling general constraints in the learnable evolution model. The methods are specifically designed to work with the learnable evolution model, and are based on trimming rules hypothesized from high performing candidate solutions, approximation of the feasible area using machine learning, and using infeasible solutions as a contrast set for learning.
- Design and implementation of methods for automatically improving the representation space in LEM. Two methods based on data-driven constructive induction were included in the presented study. The former creates new attributes in

a general form, and the latter creates only attributes that can be treated as instantiable constraints.

- Design of methods for instantiating in the modified spaces. The methods are based on the fact that conditions that include constructed attributes can be treated as constraints.
- Two real world applications are presented in this dissertation. The first one concerns optimization of parameters of complex systems, and the second concerns finding the best discretization of numeric attributes. The former application uses the AQ21 machine learning system as an example of a complex system with 24 controllable parameters and several constraints. It is applied to three medical datasets. The latter application seeks the best discretization of all numeric attributes on two medical datasets.

8.2 Future Work

Many aspects of the presented research require additional studies on the methodology, implementation, and application levels. Several assumptions have been made to simplify the process of implementation and application.

Among the most important methodological unresolved issues are:

- The methodology for handling constrained optimization problems was tested on widely known problems with relatively small numbers of constraints. Additional study is needed to adjust these methods for problems with numbers of constraints

on the order of hundreds or more. This can be done by combining the presented approach with methods already available in the literature.

- Methods for constructing new attributes presented in Chapter 6 are designed for numerical attributes only, while one of the strengths of the learnable evolution model is its applicability to problems defined using multitype attributes. An important extension of the work is automatic construction of attributes of different types such as nominal, structured, ordinal, and count.
- The presented work uses data-driven constructive induction to improve representation space. Although the method gives very good results for small representation spaces, its performance drastically decreases with increasing numbers of attributes. This is due to the complexity of a search process that involves testing different combinations of attributes. Other methods for constructing new attributes such as principal component analysis that can easily involve many attributes at the same time can be tested. Use of such methods, however require definition of efficient instantiation methods.

The presented methodology has been implemented in an experimental system LEM3. A significant amount of work is needed to transform LEM3 into a user-friendly program that does not require programming skills to operate. This includes creating a graphical interface, creating methods for easy specification of problems, removing unneeded parameters and procedures implemented for experimental purposes, and setting remaining parameters to carefully selected default values. Current efforts in this direction include integrating LEM3 with the VINLEN system (e.g. Kaufman et al, 2007).

REFERENCES

REFERENCES

Bäck, T., Fogel, D.B., and Michalewicz, Z., *Evolutionary Computation 1: Basic Algorithms and Operations*, Taylor & Francis, 2000.

Bäck, T., Fogel, D.B., and Michalewicz, Z., *Evolutionary Computation 2: Advanced Algorithms and Operations*, Taylor & Francis, 2000.

Baim, P., "The PROMISE Method for Selecting Most Relevant Attributes for Inductive Learning Systems," *Reports of the Intelligent Systems Group*, ISG 82-1, UIUCDCS-F-82-898, Department of Computer Science, University of Illinois, Urbana, September 1982.

Baluja, S., "Population Based Incremental Learning: A Method for Integrating Genetic Search Based Function Optimization and Competitve Learning," *Technical Report*, CMU-CS-94-163, Carnegie Mellon University, 1994.

Baluja, S. and Caruana, R., "Removing the Genetics from the Standard Genetic Algorithm," *Proceedings of the 12th International Conference on Machine Learning,* pp. 38-46, 1995.

Bargiela, A., and Pedrycz, W., *Granular Computing: An Introduction*, Springer, 2002.

Bay, S.D., "Multivariate Discretization for Set Mining," *Knowledge and Information Systems*, 3, pp. 491-512, 2001.

Bensusan, H., and Kuscu, I., "Constructive Induction using Genetic Programming," *Presented at Evolutionary Computing and Machine Learning Workshop at International Conference on Machine Learning*, ICML'96, 1996.

Bentley, J.B., and Corne, D.W. (eds.), *Creative Evolutionary Systems*, Morgan Kaufmann Publishers, 2002.

Bloedorn, E., "Multistrategy Constructive Induction," *Ph.D. Dissertation, Reports of the Machine Learning and Inference Laboratory*, MLI 96-7, School of Information Technology and Engineering, George Mason University, Fairfax, VA, 1996.

Bloedorn, E., and Michalski, R.S., "Constructive Induction from Data in AQ17-DCI: Further Experiments," *Reports of the Machine Learning and Inference Laboratory*, MLI

91-12, School of Information Technology and Engineering, George Mason University, Fairfax, VA, December, 1991.

Bloedorn, E., and Michalski, R.S., "The AQ17-DCI System for Data-Driven Constructive Induction and Its Application to the Analysis of World Economics," *Proceedings of the Ninth International Symposium on Methodologies for Intelligent Systems* (ISMIS-96), Zakopane, Poland, June 10-13, pp. 108-117, Springer-Verlag, 1996.

Bloedorn, E., and Michalski, R.S., "Data-Driven Constructive Induction," *IEEE Intelligent Systems, Special issue on Feature Transformation and Subset Selection*, pp. 30-37, March/April, 1998.

Bloedorn, E., Michalski, R.S., and Wnek, J., "Multistrategy Constructive Induction: AQ17-MCI," *Proceedings of the Second International Workshop on Multistrategy Learning* (MSL93), Harpers Ferry, WV, pp. 188-203, Morgan Kaufmann, May 26-29, 1993.

Bloedorn, E., Wnek, J., Michalski, R.S., and Kaufman, K., "AQ17 A Multistrategy Learning System The Method and Users Guide," *Reports of the Machine Learning and Inference Laboratory*, MLI 93-12, School of Information Technology and Engineering, George Mason University, Fairfax, VA, November, 1993.

Boyan, J., and Moore, A., "Learning Evaluation Functions to Improve Optimization by Local Search," *Journal of Machine Learning Research*, 1, pp. 77-112, 2000.

Cervone, G., "An Experimental Application of the Learnable Evolution Model to Selected Optimization Problems," *Master's Thesis, Department of Computer Science,* George Mason University, Fairfax, VA, November 1999.

Cervone, G., Kaufman, K., and Michalski, R.S., "Experimental Validations of the Learnable Evolution Model," *2000 Congress on Evolutionary Computation*, San Diego CA, pp 1064-1071, July 2000.

Cervone, G., Panait, L.A., and Michalski, R.S., "The Development of the AQ20 Learning System and Initial Experiments," *Proceedings of the Tenth International Symposium on Intelligent Information Systems*, Zakopane, Poland, June, Physica-Verlag, 2001.

Clark, P., and Niblett, T., "The CN2 Induction Algorithm," *Machine Learning*, 3, pp. 261-289, 1989.

Cohen, W., "Fast Effective Rule Induction," *Proceedings of the 12th International Conference on Machine Learning*, Tahoe City, CA, July 9-12, pp. 115-123, Morgan Kaufmann, 1995.

Coletti, M., Lash, T., Mandsager, C., Michalski, R.S., and Moustafa, R., "Comparing Performance of the Learnable Evolution Model and Genetic Algorithms on Problems in Digital Signal Filter Design," *Proceedings of the 1999 Genetic and Evolutionary Computation Conference* (GECCO), Orlando, July, 1999.

Darwin, C., *On the Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life*, J. Murray, Albemarle Street, London, 1859.

Deb, K., *Multi-objective Optimization Using Evolutionary Algorithms*, Wiley, 2001.

Domanski, P.A., Yashar, D., Kaufman K. and Michalski R.S., "An Optimized Design of Finned-Tube Evaporators Using the Learnable Evolution Model," *International Journal of Heating, Ventilating, Air-Conditioning and Refrigerating Research*, 10, pp. 201-211, April, 2004.

Dougherty, J., Kohavi, R., and Sahami, M., "Supervised and Unsupervised Discretization of Continuous Features," *Proceedings of the Twelfth International Conference on Machine Learning*, July 9-12, Tahoe City, CA, pp. 194-202, Morgan-Kaufman, 1995.

Falkenhainer, B., "ABACUS: Adding Domain Constraints to Quantitative Scientific Discovery," *Reports of the Intelligent Systems Group*, ISG 84-7, UIUCDCS-F-84-927, Department of Computer Science, University of Illinois, Urbana, November , 1984.

Falkenhainer, B., and Michalski, R.S., "Integrating Quantitative and Qualitative Discovery in the ABACUS System," In *Machine Learning: An Artificial Intelligence Approach, Vol. III*, Y. Kodratoff and R.S. Michalski (Eds.), San Mateo, CA, pp. 153-190, Morgan Kaufmann Publishers, June, 1990.

Floudas, C.A., *Handbook of Test Problems in Local and Global Optimization*, Kluwer Academic Publishers, 1999.

Floudas C.A., and Pardalos, P.M., *A Collection of Test Problems for Constrained Global Optimization Algorithms*, Lecture Notes in Computer Science, 455, Springer, 1987.

Fogel, L.J., *Intelligence through Simulated Evolution: Forty Years of Evolutionary Programming*, Wiley, 1999.

Fogel, L.J., Owens, A.J., and Walsh, M.J., *Artificial Intelligence through Simulated Evolution*, Wiley, 1966.

Gen, M. and Cheng R., *Genetic Algorithms & Engineering Optimization*, John Wiley & Sons, 2000.

Gentle, J.E., *Elements of Computational Statistics*, Springer, 2002.

Giacobini, M., Brabazon, A., Cagoni, S., Di Caro, G.A., Drechsler, R., Farooq, M., Fink, A., Lutton, E., Machado, P., Minner, S., O'Neill, M.; Romero, J., Rothlauf, F., Squillero, G., Takagi, H., Uyar, A.S., and Yang, S. (Eds.), *Applications of Evolutionary Computing: EvoWorkshops 2007: EvoCOMNET, EvoFIN, EvoIASP, EvoINTERACTION, EvoMUSART, EvoSTOC, and EvoTransLog*, Valencia, Spain, April 11-13, Lecture Notes in Computer Science, Springer, 2007.

Grefenstette, J., Gopal, R., Rosimaita, B., and Gucht, D. V. , "Genetic Algorithms for the Traveling Salesman Problem," *Proceedings of an International Conference on Genetic*

Algorithms and their Applications, July 24-24, Pittsburgh, PA, pp. 160-168, Carnegie Mellon Publishers, 1985.

Hart, W.E., Krasnogor, N., and Smith, J.E. (EDS), *Recent Advances in Memetic Algorithms*, Springer, 1994.

Hiller, F.S., and Lieberman, G.J. *Introduction to Operations Research*, Eight Edition, McGraw-Hill, 2004.

Himmelblau, D.M., *Applied Nonlinear Programming*, Mc-Graw-Hill, 1972.

Holland, J.H., "Outline for a Logical Theory of Adaptive Systems," *Journal of the ACM*, 9, pp. 279-314, 1962.

Holland, J.H., *Adaptation in Natural Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, University of Michigan Press, 1975.

Hotelling, H., "Analysis of a complex of statistical variables into principal components," *Journal of Educational Psychology*, 24:417-441,498-520, 1933.

Jensen, R.M., Veloso M., and Bryant, R.E., "Fault Tolerant Planning: Toward Probabilistic Uncertainty Models in Symbolic Non-Deterministic Planning," *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*, ICAPS, Whistler, British Columbia, Canada, June 3–7, pp. 335-344, AAAI Press, 2004.

Jones, D.R., and Beltramo, M.A., "Solving Partitioning Problems with Genetic Algorithms," *International Conference on Genetic Algorithms*, pp. 442-449, 1991.

Jourdan, L., Corne, D., Savic, D., and Walters, G., "Preliminary Investigation of the 'Learnable Evolution Model' for Faster/Better Multiobjective Water Systems Design," *Proceedings of The Third International Conference on Evolutionary Multi-Criterion Optimization*, EMO'05, Guanajuato, México, March 9-11, Springer, 2005.

Kaufman, K., "INLEN: A Methodology and Integrated System for Knowledge Discovery in Databases," *Ph.D. Dissertation*, School of Information Technology and Engineering, Reports of the Machine Learning and Inference Laboratory, MLI 97-15, George Mason University, Fairfax, VA, November, 1997.

Kaufman, K., and Michalski, R.S., "ISHED1: Applying the LEM Methodology to Heat Exchanger Design," *Reports of the Machine Learning and Inference Laboratory*, MLI 00-2, George Mason University, Fairfax, VA, 2000a.

Kaufman, K., and Michalski, R.S., "The AQ18 System for Machine Learning and Data Mining System: An Implementation and User's Guide," *Reports of the Machine Learning and Inference Laboratory*, MLI 00-3, George Mason University, Fairfax, VA, 2000b.

Kaufman, K., and Michalski, R.S., "Applying Learnable Evolution Model to Heat Exchanger Design," *Proceedings of the Seventeenth National Conference on Artificial*

Intelligence (AAAI-2000) and *Twelfth Annual Conference on Innovative Applications of Artificial Intelligence* (IAAI-2000), Austin, TX, pp. 1014-1019, 2000c.

Kaufman, K., Michalski, R.S., Pietrzykowski, J., and Wojtusiak, J., "An Integrated Multi-task Inductive Database VINLEN: An Initial Implementation and Early Results," *Proceedings of The 5th International Workshop on Knowledge Discovery in Inductive Databases*, KDID'06, in conjunction with ECML/PKDD, Berlin, Germany, September 18, pp. 116-133, Springer, 2007.

Kerber, R., "ChiMerge: Discretization of Numeric Attributes," *Proceedings of the Tenth National Conference on Artificial Intelligence*, July 12-16, San Jose, CA, pp. 123-128, AAAI Press, 1992.

Kovalov, M.Y., Ng, C.T., and Cheng T.C.E., "Fixed Interval Scheduling: Models, Applications, Computational Complexity and Algorithms," *European Journal of Operations Research*, 178, pp. 331-342, 2007.

Koza, J.R., *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.

Koziel, S., and Michalewicz, Z., "Evolutionary Algorithms, Homomorphous Mappings, and Constrained Parameter Optimization," *Evolutionary Computation*, 7, pp.19-44, 1999.

Krawiec K., "Genetic Programming-based Construction of Features for Machine Learning and Knowledge Discovery Tasks," *Genetic Programming and Evolvable Machines*, 3, pp. 329–343, 2002.

Larrañaga, P. and Lozano, J. (eds.), *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*, Kluwer Academic Publishers, 2002.

Liang, J.J., Runarsson, T.P., Mezura-Montes, E., Clerc, M., Suganthan, P.N., Coello Coello, C.A., and Deb, K., "Problem Definitions and Evaluation Criteria for the CEC 2006 Special Session on Constrained Real-Parameter Optimization," *Technical Report*, Nanyang Technical University, Singapore, 2005.

Liang, J.J., and Suganthan P.N., "Dynamic Multi-Swarm Particle Swarm Optimizer with a Novel Constraint-Handling Mechanism," *Proceedings of IEEE Congress on Evolutionary Computation*, CEC 2006, 16-21 July 2006**.**

Liu, H., and Motoda, H. (eds.), *Feature Transformation and Subset Selection,* Special Issue of the *IEEE Intelligent Systems*, 1998.

Liu, H., Stine, R., and Auslender, L. (eds.), *Workshop on Feature Selection for Data Mining: Interfacing Machine Learning and Statistics*, Newport Beach, CA, April, 2005.

Llora, X., and Goldberg, D. E., "Wise Breeding GA via Machine Learning Techniques for Function Optimization," *Proceedings of Genetic and Evolutionary Computation Conference*, GECC0 2003, pp. 1172-1183, Chicago, Il, July 12-16, 2003.

Mackworth, A.K, "Consistency in Networks of Relations," *Artificial Intelligence*, 8, pp. 99-118, 1977.

Markovich, S., and Rosenstein, D., "Feature Generation Using General Constructor Functions," *Machine Learning*, 49, pp. 59-98, 2002.

Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs,* Springer Verlag, 3rd edition, 1996.

Michalewicz, Z., Chapters 6-11 in Bäck, T., Fogel, D.B., and Michalewicz, Z., *Evolutionary Computation 2*, pp. 38-74, Taylor & Francis, 2000.

Michalewicz, Z., and Schoenauer M., "Evolutionary Algorithms for Constrained Parameter Optimization Problems," *Evolutionary Computation* 4, 1, pp. 1-32, 1996.

Michalewicz, Z., and Nazhiyath, G., "Genocop III: A Co-evolutionary Algorithm for Numerical Optimization Problems with Nonlinear Constraints," *Proceedings of the 2nd IEEE International Conference on Evolutionary Computation*, Vol.2, Perth, Australia, pp. 647-651, 1995.

Michalski, R. S., "On the Quasi-Minimal Solution of the General Covering Problem," *Proceedings of the V International Symposium on Information Processing* (FCIP 69), Vol. A3 , Yugoslavia, Bled, pp. 125-128, October 8-11, 1969.

Michalski, R.S., "A Variable-Valued Logic System as Applied to Picture Description and Recognition," in F. Nake, and A. Rosenfeld (eds.), *Graphic Languages*, North-Holland Publishing Co., 1972.

Michalski, R.S. "Graphical Minimization of Normal Expressions of Logic Functions using Tables of the Veitch-Karnaugh Type," *Journal of the Institute of Automatic Control*, 52, 1978.

Michalski, R.S., "A Theory and Methodology of Inductive Learning," *Machine Learning: An Artificial Intelligence Approach*, R.S. Michalski, T. J. Carbonell and T. M. Mitchell (eds.), pp. 83-134, TIOGA Publishing Co., Palo Alto, 1983.

Michalski, R.S., "Learnable Evolution: Combining Symbolic and Evolutionary Learning," *Proceedings of the Fourth International Workshop on Multistrategy Learning* (MSL'98), Desenzano del Garda, Italy, pp. 14-20, June 11-13, 1998.

Michalski, R. S., "LEARNABLE EVOLUTION MODEL: Evolutionary Processes Guided by Machine Learning," *Machine Learning*, 38, pp. 9-40, 2000.

Michalski, R.S., "ATTRIBUTIONAL CALCULUS: A Logic and Representation Language for Natural Induction," *Reports of the Machine Learning and Inference Laboratory*, MLI 04-2, George Mason University, Fairfax, VA, April, 2004a.

Michalski, R.S., "Generating Alternative Hypotheses in AQ Learning," *Reports of the Machine Learning and Inference Laboratory*, MLI 04-6, George Mason University, Fairfax, VA, December, 2004b.

Michalski, R.S., and Cervone, G., "Adaptive Anchoring Discretization for Learnable Evolution Model: The ANCHOR Method," *Reports of the Machine Learning and Inference Laboratory*, MLI 01-3, George Mason University, Fairfax, VA, 2001.

Michalski, R.S., and Kaufman, K., "The AQ19 System for Machine Learning and Pattern Discovery: A General Description and User's Guide," *Reports of the Machine Learning and Inference Laboratory*, MLI 01-2, George Mason University, Fairfax, VA, 2001a.

Michalski, R.S., and Kaufman, K., "Learning Patterns in Noisy Data: The AQ Approach," In G. Paliouras, V. Karkaletsis and C. Spyropoulos, (Eds.) *Machine Learning and its Applications*, Springer-Verlag, pp. 22-38. 2001b.

Michalski, R.S., and Kaufman, K., "INTELLIGENT EVOLUTIONARY DESIGN: A New Approach to Optimizing Complex Engineering Systems and its Application to Designing Heat Exchangers," *International Journal of Intelligent Systems*, Volume 21, Issue 12, 2006.

Michalski, R.S., Kaufman, K., Pietrzykowski, J., Sniezynski, B. and Wojtusiak, J., "Learning User Models for Computer Intrusion Detection: Preliminary Results from Natural Induction Approach," *Reports of the Machine Learning and Inference Laboratory*, MLI 05-3, George Mason University, Fairfax, VA, November, 2005.

Michalski, R.S., and Larson, J., "AQVAL/1 (AQ7) User's Guide and Program Description," *Report No. 731*, Department of Computer Science, University of Illinois, Urbana, June 1975.

Michalski, R.S., and Larson, J., "Selection of Most Representative Training Examples and Incremental Generation of VL1 Hypotheses: The Underlying Methodology and the Description of Programs ESEL and AQ11," *Report No. 867*, Department of Computer Science, University of Illinois, Urbana, May 1978.

Michalski, R.S., and Larson, J., "Incremental Generation of VL1 Hypotheses: The Underlying Methodology and the Description of Program AQ11," *Reports of the Intelligent Systems Group*, ISG 83-5, UIUCDCS-F-83-905, Department of Computer Science, University of Illinois, Urbana, January 1983.

Michalski, R.S., and Wojtusiak, J., "Reasoning with Meta-values in AQ Learning," *Reports of the Machine Learning and Inference Laboratory*, MLI 05-1, George Mason University, Fairfax, VA, June, 2005.

Michalski, R.S., and Wojtusiak, J., "Semantic and Syntactic Attribute Types in AQ Learning," *Reports of the Machine Learning and Inference Laboratory*, MLI 07-1, George Mason University, Fairfax, VA, 2007.

Michalski, R.S., and Zhang, Q., "Initial Experiments with the LEM1 Learnable Evolution Model: An Application to Function Optimization and Evolvable Hardware," *Reports of the Machine Learning and Inference Laboratory*, MLI 99-4, George Mason University, Fairfax, VA, May 1999.

Moscato, P., "On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms", *Caltech Concurrent Comput*ation *Program*, C3P Report 826, 1989.

Muharram, M., and Smith, G. D., "Evolutionary Constructive Induction," *IEEE Transactions on Knowledge and Data Engineering*, 17, pp. 1518-1528, 2005.

Mühlenbein, H., and Paaß, G., "From Recombination of Genes to the Estimation of Distributions I. Binary Parameters," *Proceedings of The 4th International Conference on Parallel Problem Solving from Nature*, Berlin, Germany, September 22-26, 1996.

Oyama, A., "Wing Design Using Evolutionary Algorithms," *Ph.D. Thesis*, Department of Aeronautics and Space Engineering of Tohoku University, 2000.

Quinlan, J.R., "Induction of Decision Trees," *Machine Learning*, 1, pp. 81-106, 1986.

Quinlan, J.R., *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1993.

Rardin, R.L., *Optimization in Operations Research*, Prentice Hall, 1997.

Rasheed, K.M., "GADO: A Genetic Algorithm for Continuous Design Optimization," *Ph.D. Thesis*, The State University of New Jersey, 1998.

Rechenberg, I., "Cybernetic Solution Path of an Experimental Problem," *Royal Aircraft Establishment*, Library Translation No. 1122, 1965.

Reynolds, R.G., "An Introduction to Cultural Algorithms," *Proceedings of the Third Annual Conference on Evolutionary Programming*, World Scientific Publishing, River Edge, NJ, 1994.

Reynolds, R.G., and Peng, B., "Cultural Algorithms: Modeling of How Cultures Learn to Solve Problems," *16th IEEE International Conference on Tools with Artificial Intelligence* (ICTAI'04), 15-17 November, Boca Raton, Fl, pp. 166-172, 2004.

Reynolds, R.G., and Zhu, S., "Knowledge-Based Function Optimization Using Fuzzy Cultural Algorithms with Evolutionary Programming," *IEEE Transactions on Systems*, Man, and Cybernetics, 31, pp. 1-18, 2001.

Reinke, R., "Knowledge Acquisition and Refinement Tools for the ADVISE META-EXPERT System," *Reports of the Intelligent Systems Group*, ISG 84-6, UIUCDCS-F-84-926, Department of Computer Science, University of Illinois, Urbana, 1984.

Riley, P., and Veloso, M., "Coach planning with opponent models for distributed execution," *Autonomous Agents and Multi-Agent Systems*, 13, pp. 293-325 , 2006.

Rothlauf, F., Branke, J., Cagnoni, S., Costa, E., Cotta, C., Drechsler, R., Lutton, E., Machado, P., Moore, J.H., Romero, J., Smith, G.D., Squillero, G., and Takagi, H. (Eds.), *Applications of Evolutionary Computing, EvoWorkshops 2006: EvoBIO, EvoCOMNET, EvoHOT, EvoIASP, EvoINTERACTION, EvoMUSART, and EvoSTOC*, Budapest, Hungary, April 10-12, 2006.

Saleem, S., and Reynolds, R.G., "Function Optimization with Cultural Algorithms in Dynamic Environments," *Proceedings of the Workshop on Particle Swarm Optimization 2001*, Purdue School of Engineering and Technology, Indianapolis, IN, 2001.

Schwefel H-P. "Kybernetische Evolution als Strategie der exprimentellen Forschung in der Strömungstechnik," *Master's thesis*, Technical University of Berlin, 1965.

Sebag, M., and Schoenauer, M., "Controlling Crossover through Inductive Learning," *Proceedings of the 3rd Conference on Parallel Problems Solving from Nature*, PPSN '94, Lecture Notes in Computer Science, pp. 209-218, Springer, 1994.

Sebag, M., Schoenauer, M., and Ravisé, C., "Toward Civilized Evolution: Developing Inhibitions," *Proceedings of the 7th International Conference on Genetic Algorithms*, East Lansing, MI, USA, July 19-23, pp. 291-298, 1997a.

Sebag, M., Schoenauer, M., and Ravisé, C., "Inductive Learning of Mutation Step-size in Evolutionary Parameter Optimization," Proceedings of the 6th Annual Conference on Evolutionary Programming, pp. 247-261, 1997b.

Sniezynski, B., Szymacha, R., and Michalski, R. S, "Knowledge Visualization Using Optimized General Logic Diagrams," *Proceedings of the Intelligent Information Processing and Web Mining Conference*, IIPWM 05, Gdansk, Poland, June 13-16, 2005.

Storn, R., and Price, K., "Differential Evolution - a Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Spaces," *Journal of Global Optimization*, 11, pp. 341-359, 1997.

Price, K.V., Storn, R.M., and Lampinen, J.A., *Differential Evolution: A Practical Approach to Global Optimization*, Springer, 2005.

Surry, P.D., Radcliffe, N.J., and Boyd, I.D., "A Multi-objective Approach to Constrained Optimisation of Gas Supply Networks: The COMOGA Method," *Evolutionary Computing, AISB Workshop*, Lecture Notes in Computer Science, pp. 160-180, Springer, 1995.

Syswerda, G., "Schedule optimization using genetic algorithms," In *Handbook of Genetic Algorithms*, L. Davis (Ed.), Van Nostrand Reinhold, 1991.

Takahama, T., and Sakai, S., "Constrained Optimization by the ε Constrained Differential Evolution with Gradient-Based Mutation and Feasible Elites," *Proceedings of IEEE Congress on Evolutionary Computation*, CEC 2006, 16-21 July, 2006.

Tay, F.E.H., and Shen, L., "A Modified Chi2 Algorithm for Discretization," *IEEE Transactions on Knowledge and Data Engineering*, 14, pp. 666-670, 2002.

Whitley, L.D., Starkweather, T., and Fuquay, D., "Scheduling Problem and Traveling Salesman: The Genetic Edge Recombination Operator," *Proceedings of the 3rd International Conference on Genetic Algorithms*, Fairfax, VA, pp. 133-140, June, 1989.

Wnek, J., "Hypothesis-driven Constructive Induction," *Ph.D. Dissertation*, School of Information Technology and Engineering, George Mason University, 1993.

Wnek, J., "DIAV 2.0 User Manual: Specification and Guide through the Diagrammatic Visualization System," *Reports of the Machine Learning and Inference Laboratory*, MLI 95-5, George Mason University, Fairfax, VA, 1995.

Wnek, J., and Michalski, R.S., "Hypothesis-Driven Constructive Induction in AQ17: A Method and Experiments," *Proceedings of the IJCAI-91 Workshop on Evaluating and Changing Representation in Machine Learning*, Sydney, Australia, August, 1991.

Wnek, J., and Michalski, R.S., "Hypothesis-driven Constructive Induction in AQ17-HCI: A Method and Experiments," *Machine Learning*, 14, 2, pp. 139-168, 1994.

Wnek, J., Kaufman, K., Bloedorn, E., and Michalski, R. S., "Inductive Learning System AQ15c: The Method and User's Guide," *Reports of the Machine Learning and Inference Laboratory*, MLI 95-4, George Mason University, Fairfax, VA, March 1995.

Wojtusiak, J., "AQ21 User's Guide," *Reports of the Machine Learning and Inference Laboratory*, MLI 04-3, George Mason University, Fairfax, VA, September, 2004a (updated in September, 2005).

Wojtusiak, J., "The LEM3 Implementation of Learnable Evolution Model: User's Guide," *Reports of the Machine Learning and Inference Laboratory*, MLI 04-5, George Mason University, Fairfax, VA, November, 2004b.

Wojtusiak, J., "Initial Study on Handling Constrained Optimization Problems in Learnable Evolution Model," *Proceedings of The Graduate Student Workshop at Genetic and Evolutionary Computation Conference*, GECCO 2006, Seattle, WA, July 8-12, 2006.

Wojtusiak, J., and Michalski, R.S., "The LEM3 System for Non-Darwinian Evolutionary Computation and Its Application to Complex Function Optimization," *Reports of the Machine Learning and Inference Laboratory*, MLI 05-2, George Mason University, Fairfax, VA, October, 2005.

Wojtusiak, J., and Michalski, R.S., "The LEM3 Implementation of Learnable Evolution Model and Its Testing on Complex Function Optimization Problems," *Proceedings of Genetic and Evolutionary Computation Conference*, GECCO 2006, Seattle, WA, July 8-12, 2006.

Wojtusiak, J., Michalski, R.S., Kaufman, K., and Pietrzykowski, J., "Multitype Pattern Discovery Via AQ21: A Brief Description of the Method and Its Novel Features," *Reports of the Machine Learning and Inference Laboratory*, MLI 06-2, George Mason University, Fairfax, VA, 2006a.

Wojtusiak, J., Michalski, R. S., Kaufman, K., and Pietrzykowski, J., "The AQ21 Natural Induction Program for Pattern Discovery: Initial Version and its Novel Features," *Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence*, Washington D.C., November 13-15, 2006b.

Wojtusiak, J., Michalski, R.S., Simanivanh, T., and Baranova A.V., "The Natural Induction System AQ21 and Its Application to Data Describing Patients with Metabolic Syndrome: Initial Results," *International Conference on Machine Learning and Applications*, ICMLA 2007, Cincinnati, Ohio, 13-15 December, 2007.

Zadeh, L.A. "Fuzzy sets," *Information and Control*, 8, 3, pp. 338-353, 1965.

CURRICULUM VITAE

Janusz Wojtusiak obtained with honors his Master's degree in Computer Science from the Jagiellonian University in 2001. In his master's thesis he investigated an application of artificial intelligence to biology.

After getting his Master's degree Janusz started Ph.D. studies in the Institute of Computer Science, Jagiellonian University. In 2002 he was invited by Professor Ryszard Michalski to be a visiting scientist in the George Mason University Machine Learning and Inference Laboratory. In 2003, he moved to the Computational Sciences and Informatics Ph.D. program, with concentration area Computational Intelligence and Knowledge Mining, in the George Mason University College of Science (formerly School of Computational Sciences), and received a research assistantship. In his Ph.D. dissertation, he investigates the learnable evolution model, an evolutionary computation method that uses machine learning to guide the evolution process, and its application to the optimization of very complex systems.

Janusz Wojtusiak has teaching experience from Jagiellonian University, where he taught laboratories on digital circuits, computer simulation, and methods of artificial intelligence. He also has experience in developing commercial software, which he gained while working for a private company between 1999 and 2002. Janusz is currently an assistant director and system manager in the George Mason University Machine Learning and Inference Laboratory (see http://www.mli.gmu.edu). He authored or co-authored over 20 research papers in machine learning, evolutionary computation and health informatics related fields. Janusz is a member of the Association for Advancement of Artificial Intelligence (AAAI), Institute of Electrical and Electronics Engineers (IEEE) Computer Society and Technical Committee on Intelligent Informatics (TCII), and Association of Computing Machinery (ACM) Special Interest Group for Genetic and Evolutionary Computation (SIGEVO). He also serves as a reviewer for international journals and conferences in these fields.

Janusz's research interests include artificial intelligence, machine learning, evolutionary computation, knowledge mining, and intrusion detection, with particular applications of these fields in health care. He is involved in various projects including learnable evolution model, inductive databases, AQ learning, learning user signatures, and several projects that involve analysis of different types of medical data. He is developing and applying the AQ21 and LEM3 systems, and partially involved in the development of the VINLEN system.

A publication of the *Machine Learning and Inference Laboratory*
George Mason University
Fairfax, VA 22030-4444 U.S.A.
http://www.mli.gmu.edu

Editor: Janusz Wojtusiak

The *Machine Learning and Inference (MLI) Laboratory Reports* are an official publication of the Machine Learning and Inference Laboratory, which has been published continuously since 1971 by R.S. Michalski's research group (until 1987, while the group was at the University of Illinois, they were called ISG (Intelligent Systems Group) Reports, or were part of the Department of Computer Science Reports).