UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Scuola di Dottorato in Ingegneria dell'Informazione
Dottorato di Ricerca in Ingegneria Informatica ed Automatica

A.D. MCCXXIV

Comunita Europea
Fondo Sociale Europeo

An Approach to Pattern Recognition by Evolutionary Computation

FRANCESCO FONTANELLA

Tesi di Dottorato di Ricerca

Novembre 2005

UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

A.D. MCCXXIV

Scuola di Dottorato in Ingegneria dell'Informazione
Dottorato di Ricerca in Ingegneria Informatica ed Automatica

Comunita Europea
Fondo Sociale Europeo

Pattern Recognition by Evolutionary Computation

FRANCESCO FONTANELLA

Tesi di Dottorato di Ricerca

(XVIII ciclo)

Novembre 2005

Il Tutore
Prof. L.P. Cordella

Il Coordinatore del Dottorato
Prof. L. P. Cordella

Il Co-Tutore
Prof C. De Stefano

Dipartimento di Informatica e Sistemistica

Contents

Abstract **viii**

Acknowledgments **ix**

1 Introduction **1**

1.1 Pattern Recognition 1

1.1.1 Representation 2

1.1.2 Classification 4

1.1.3 Prototype Generation 4

1.2 Pattern Recognition Systems 5

1.3 The Evolutionary Computation Paradigm 7

1.3.1 Fitness 7

1.3.2 Selection 8

1.3.3 Solution Encoding 8

1.3.4 Operators 10

1.4 A Reader's Guide to the Thesis 10

2 Overview of Pattern Recognition **12**

2.1 Classification Algorithms 12

2.1.1 k-nearest neighbor 13

2.1.2 Artificial Neural Networks 14

2.1.3 Decision Trees 17

2.2 Classification Applications 18

2.2.1 Fingerprint Classification 19

2.2.2 Speaker Identification 19

2.2.3 Medical Applications 20

2.3 Clustering Algorithms 22

2.3.1 K-means 22

2.3.2 Hierarchical Clustering 24

2.3.3 Self-Organizing Maps 27

2.4 Clustering Applications 29

2.4.1 Image Segmentation . . . 29
2.4.2 Object Recognition . . . 30
2.4.3 Character Recognition . . . 31
2.4.4 Information Retrieval . . . 32
2.4.5 Data Mining . . . 33

3 Evolutionary Computation 34
3.1 Advantages of Evolutionary Computation . . . 34
3.2 The Evolutionary Algorithm Paradigm . . . 35
3.2.1 Solution Encoding . . . 39
3.2.2 Operators . . . 40
3.2.3 Selection Methods . . . 42
3.2.4 Fitness . . . 47
3.3 Genetic Algorithms . . . 52
3.3.1 Recombination . . . 54
3.3.2 Mutation . . . 54
3.4 Genetic Programming . . . 55
3.4.1 Representation of GP Individuals . . . 57
3.4.2 Initialization of a GP Population . . . 58
3.4.3 Recombination . . . 60
3.4.4 Mutation . . . 61
3.5 Grammatically–based Genetic Programming . . . 62
3.5.1 An Example of Application of CFG-GP . . . 63
3.6 Applications of EC–based Algorithms . . . 66

4 A Framework for Classification 69
4.1 Related and Previous Work . . . 69
4.2 An EC–based Framework for Classification problems . . . 71
4.2.1 Solution Encoding . . . 73
4.2.2 Prototype Labeling . . . 74
4.2.3 Fitness Evaluation . . . 77
4.2.4 Operators . . . 79
4.3 A CFG-GP Version of the EC-framework . . . 79
4.4 The LVQ Version of the EC-framework . . . 82

5 A New EC–based Method for Evolving Graphs 85
5.1 Previous and Related Work . . . 85
5.2 A New Data Structure for Graph Encoding . . . 87
5.3 A Formal Definition of Multilist . . . 89
5.3.1 Some Definitions on the Multilists . . . 90
5.3.2 The Cut Operation . . . 92

5.3.3 The Merge Operation 93
5.3.4 Further Operations on the Multilists 94
5.4 Graph Encoding by Multilists 94
5.4.1 The Cut Operation 96
5.4.2 The Merge Operation 97
5.5 Operators 98
5.5.1 Crossover 98
5.5.2 Mutation operator 102
5.6 Testing the approach 105

6 Experimental Results **108**
6.1 The CFG-GP Version of the Framework 108
6.1.1 Data sets 109
6.1.2 Data Normalization 110
6.1.3 Parameter Settings 110
6.1.4 Results 111
6.1.5 Comparison Findings 114
6.2 The LVQ Version of the Framework 115
6.2.1 Synthetically Generated Data 115
6.2.2 NIST Database 116
6.2.3 Satellite Images 119
6.3 Evolutionary Graph Generation 123
6.3.1 Parameter Settings 124
6.3.2 Results 125

7 Conclusions and Future Work **130**
7.1 Contributions 131
7.1.1 A Framework for Prototype Generation 131
7.1.2 Evolving Graphs 132
7.2 Conclusions 132
7.2.1 Framework for Prototype Generation 133
7.2.2 Evolving Graphs 134
7.3 Future Work 136
7.3.1 Framework for Prototype Generation 136
7.3.2 Evolving Graphs 137

Appendix A **139**

Appendix B **139**

Bibliography **144**

List of Figures

1.1 Patterns in a 2-D space 3
1.2 A pattern recognition system 6

2.1 Scheme of a neuron. 13
2.2 General scheme of a multilayer neural network. 14
2.3 A structure of a typical decision tree. 17
2.4 Decision trees 18
2.5 An example of clustering 23
2.6 K-means clustering 24
2.7 An example of dendrogram 25
2.8 An example of Venn diagram 26
2.9 Distribution of neurons in a SOM 28
2.10 Texture clustering. 29
2.11 Clusters of digits. 32

3.1 Flow chart of the EC schema. 37
3.2 Roulette wheel selection. 43
3.3 Tournament selection properties 46
3.4 Selection methods comparison 48
3.5 Dependence of loss of diversity on selection intensity. 48
3.6 En example of fitness landscape 49
3.7 GA crossover 54
3.8 GA mutation 55
3.9 GP tree 57
3.10 GP crossover 60
3.11 An example of mutation application. 61
3.12 GP tree for 6-multiplexer 62
3.13 A CFG tree for the 6-multiplexer 64
3.14 An example of CFG-GP crossover 66

4.1 Framework solution encoding 72
4.2 Two different ways of writing the letter “A”. 73

4.3 The outline of the framework 74
4.4 Framework dynamic labeling 75
4.5 A labeling comparison 78
4.6 Framework recombination operator 80
4.7 An example of CFG-GP solution encoding 81
4.8 An LVQ individual 82
4.9 The LVQ mutation operator 84

5.1 Examples of ML's 1 88
5.2 Example of incomplete multilist 91
5.3 Example of redundant multilist 91
5.4 Example of totally redundant multilist 91
5.5 The cut operation 92
5.6 The merge operation 93
5.7 The reduction operation 94
5.8 The completion operation 96
5.9 Two encoded graphs 97
5.10 Graphs from the merge operation 99
5.11 Crossover offspring 1 100
5.12 Crossover offspring 2 101
5.13 Crossover offspring 3 102
5.14 The reduction operation 2 103
5.15 The reduction operation 3 103
5.16 The null completion 103
5.17 Mutation 104
5.18 An instance of the wireless LAN design problem 106

6.1 BUPA recognition rates 113
6.2 Vehicle recognition rates 113
6.3 Synthetic data 1 117
6.4 Synthetic data 2 117
6.5 Synthetic data 3 118
6.6 Synthetic data 4 118
6.7 Classification map of the forest cover image 121
6.8 Classification map of the land cover image 121
6.9 Land cover Recognition rate 122
6.10 Access points plot 127
6.11 Wire lengths plot 127
6.12 Best 10 128
6.13 Best 100 128
6.14 Best 300 129

6.15 Best 500 129

7.1 An example of cut graph. 141

List of Tables

3.1 GP symbols for 6-multiplexer 62
3.2 A context free grammar for the 6-multiplexer problem. 64

4.1 The grammar for logical expressions 82

6.1 Data sets 109
6.2 CFG-GP evolutionary parameters 111
6.3 CFG-GP comparison 112
6.4 NIST parameters 116
6.5 NIST recognition rates 116
6.6 Satellite image parameters 119
6.7 Forest cover results 120
6.8 Land cover results 120
6.9 Evolutionary parameters for graph generation 124

Abstract

Evolutionary Computation (EC) has been inspired by the natural phenomena of evolution. It provides a quite general heuristic, exploiting a few basic concepts: reproduction of individuals, variation phenomena that affect the likelihood of survival of individuals and inheritance of parents features by offspring. EC has been widely used in recent years to effectively solve hard, non linear and very complex problems.

Amongst other things, EC–based algorithms have also been used to tackle classification problems. Classification is a process according to which an object is attributed to one of a finite set of classes or, in other words, it is recognized as belonging to a set of equal or similar entities, identified by a *label*. The main aspect of classification usually concerns the generation of prototypes to be used to recognize unknown patterns. The role of prototypes is that of representing patterns belonging to the different classes defined within a given problem. For most of the problems of practical interest, the generation of such prototypes is very difficult, since a prototype must be able to represent patterns belonging to the same class, which may be significantly dissimilar to each other. They must also be able to discriminate between patterns belonging to classes different from the one that they represent. Moreover, a prototype should contain the minimum amount of information required to satisfy the requirements mentioned above.

The research presented in this thesis has led to the definition of an EC–based framework to be used for prototype generation. The defined framework does not provide for the use of any particular kind of prototypes. In fact, it can generate any kind of prototype once an encoding scheme for the used prototypes has been defined. The generality of the framework can be exploited to develop many applications. The framework has been employed to implement two specific applications for prototype generation.
The developed applications have been tested on several data sets and the results compared with those obtained by other approaches previously presented in the literature.

Acknowledgments

Several people have made important contributions to the research work I carried out during the three years of my doctorate activity summarized in this Thesis.
I would like to thank, first of all, my tutor Professor Luigi Pietro Cordella. During the three years I benefited hugely from his experience and the precious advice he gave me.
An important part of the work presented in this Thesis originates from the long and useful discussions I had with my co-tutor, Professor Claudio De Stefano. I owe him a debt of gratitude. I would also like to thank prof. Angelo Marcelli for the encouragement he gave me for a part of the work presented here.
I'm grateful to Antonio Della Cioppa and Ivanoe De Falco, who initiated me in the evolutionary computation concepts.
I also want to thank prof. Giuseppe Trautteur, I am indebted to him.

Combining a PhD courses with having a family is a very hard job. I thank Teresa, my wife, for her constant support and understanding.
This thesis is dedicated to my mother, Orsola Bonino. Her gift to me is the way I look at the world, without this gift this dissertation would never have happened.

To my mother,
the origin of all of this

Chapter 1

Introduction

The term *Pattern recognition* denotes a wide class of activities, including the human mental processes which transform the sensory activities into perceptive experiences. Pattern recognition can be defined as the act of tacking in raw data and taking actions based on the "category" of the patterns recognized in the data [DHS01]. In this work the term will be used to meaning the recognition process automatically performed by a computer. This activity is often denoted as *automatic* or *mechanical* pattern recognition.

1.1 Pattern Recognition

Since the beginning of computer science the problem of recognizing patterns in raw data obtained from real world has been faced. This process is of great interest in many application fields: for example, optical character recognition (OCR) is of great deal for automatic processing of documents. Human faces recognition is gaining a lot of interest in security issues. The term *pattern* will be meant a diversified set of information to be used for representing the object to be recognized. This kind of information is generally obtained by means of sensors acquiring descriptions of objects from the real world.

The pattern recognition process is mainly characterized by three aspects:

- *representation* or *description* of the patterns to be recognized;
- *classification*, i.e. the act of recognizing the "category" to which the patterns provided to the recognizing system belong to;
- *prototyping*, i.e. the mechanism used for generating the *prototypes* or *models*. Prototypes are used for representing the different categories to be recognized.

Note that both classification and prototyping strongly depend on the representation chosen to describe the pattern to be recognized. In the following subsections each of this aspects will be detailed.

1.1.1 Representation

In the pattern recognition field the problem of representation is that of processing the information provided by the sensors in such a way that it is reduced to the minimum needed for the recognition task, eliminating that part of information that is unnecessary, which in some cases may even misleads the recognition process. On the other hand such processing must preserve the information useful to perform the recognition task.

A “good” representation is one in which patterns demanding the same action have representations somehow “close” to each other, yet “far” from those that demand different actions. Moreover, such kind of representation should allow one to easily represent prototypes describing different categories.

In some cases the patterns to be recognized are represented as vectors of real-valued numbers. This approach to the representation is called *statistical*. An alternative approach to the representation is called *structural*. In the latter approach a pattern is described in term of its component parts their relationships. These approaches are described below.

The Statistical Approach

The first approach used in the pattern recognition field for describing a patterns has been that of representing it as a vector consisting of measures of some of its characteristics or *features*. Following this approach, the patterns may be seen as points in the *feature space*, a theoretical n-dimensional space where n is the number of features chosen for describing the patterns. The values assumed by the features of a pattern are the coordinates of the point representing the pattern in that space. The statistical approach is essentially based on the idea that if the features are appropriately chosen, patterns belonging to the same class are represented in the feature space by near points according to a suitable chosen metric. From this standpoint, the problem of pattern recognition becomes that of partitioning the feature space in different regions, each one containing points belonging to the same class. It is worth noting that the number of such regions can be greater than the number of classes actually present in the problem. Since in many cases a class may consist of several set of patterns, in which patterns belonging to the same group are similar, whereas those ones belonging to different sets are quite

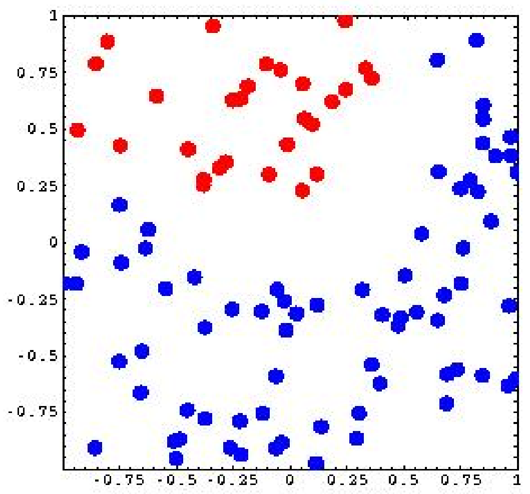


Figure 1.1: Patterns represented in a 2-D feature space. Each pattern is represented using a vector of two features, thus it is possible to represent the features in a 2-D dimensional Cartesian space.

unlike. Usually these sets of patterns are said to be *sub-classes* of the class their patterns belong to.

The crucial aspect of the statistical approach is choosing the best set of features to be used for describing the pattern. This set has to provide the system with the minimum quantity of information necessary for distinguishing patterns belonging to different classes. But this task in many cases may be very hard. *Feature selection* techniques have been devised that allow one to choose the most distinctive features among those available. However, the problem of choosing a first set of candidate features remains.

The Structural Approach

Since the beginning of the 60's a different approaches to pattern recognition have been devised. It is based on the idea that a complex object can be described by means of the simpler parts, called *components* or *primitives*, of which it is made of. For example, a technical drawing can be represented in terms of lines and symbols. This decomposition process can be recursively applied to the decomposed object: the symbols can be on their turn described as sets of segments, arcs, etc. For an effective description of an object it is not sufficient to describe its components, since a crucial role is played by the relationships, or simply *relations*, existing among these simpler parts. Thus, according to this approach an object is described by its structure, whence the name *structural approach.*

The main drawbacks of the structural approach concern both the classification process and prototypes generation, as they are based on more complex methods than that available for vectorial description. Particularly, classification requires a comparison method between structural description providing a similarity measure. The definition of such methods it is not at all a simple task.

1.1.2 Classification

Classification is a process according to which an entity is attributed to one of a finite set of *classes* or, in other words, it is recognized as belonging to a set of equal or similar entities, possibly identified by a name. Classification can also be alternatively defined as the process that, given a set of prototypes, assigns a pattern to one of them. The effect of such a assignment is that of dividing the patterns in classes[1].

In many cases, classification is based on a "distance" function which takes in input two descriptions of patterns and gives as output a value, usually normalized in the range $[0.0, 1.0]$. This value measures the similarity of the patterns given in input, or equivalently the distance between the patterns. Given such a function and a set of prototypes, the assignment of an incoming pattern to one of the classes is performed by measuring the distance between the pattern and each of the prototypes. Then, the pattern is assigned to the same class of its nearest prototype. Note that such approach can be applied to any kind of description used to represent the patterns.

The degree of difficulty of a classification problem strongly depends on the variability in the feature values for patterns in the same class relative to the difference between feature values for patterns in different classes. The variability of feature values for patterns in the same class may be due both to the intrinsic complexity of the problem and to *noise.* Noise can be defined in very general terms: any property of the sensed pattern which is not due to the true characteristics of the pattern but instead to the randomness in the world or the sensors.

1.1.3 Prototype Generation

The third main aspect of pattern recognition is the mechanism employed for generating prototypes, quite often denoted by the term *prototyping.* A

[1]In the following this term will substitute the term "categories" used so far.

prototype can be defined as a model on which all (or at least a part of[2]) the patterns of a given class can be traced back to. The function of a prototype is that of representing all the patterns that can be traced back to it. A prototype may be seen as a way to reduce the information needed for describing a much larger quantity of information: that represented by all the patterns it describes. Moreover, a prototype musts have another fundamental requisite: its representation ability has to be *discriminative*, i.e. it has to be able to represent the patterns of just a single class.

The concept of prototype is quite general. In many cases it is a pattern. But in some cases it may be represented in a different way from the patterns. In the *syntactic* approach to pattern recognition, for example, logical rules can be used to decide to which class a given prototype has to be assigned. In such a case, the set of rules making up a prototype specifies the requirements met by the patterns that it represents.

A set of prototypes may be built either exploiting the a priori knowledge of the problem or deducing it on the basis of a set of patterns provided to the system. This deductive process may be seen as a form of *learning* and it is usually called *training* whereas the set of patterns provided is called *training set*. Some general methods can be employed for training a classifier:

Supervised a teacher provides a class label for each pattern in the training set. The goal of the supervised training is yielding a set of labeled prototypes that minimizes the *error* achieved by the system on the provided training set. This error indicates the percentage of misclassified training patterns[3].

Unsupervised no information is provided on the actual label of the patterns in the training set. The system automatically forms clusters of the input patterns. The clustering depend on a similarity function provided to the system. The role of this function is that of measuring how much two patterns are similar. This kind of training is also called *clustering*.

1.2 Pattern Recognition Systems

In the classification process performed by a pattern recognition system three different operations can be distinguished: preprocessing, feature extraction

[2]As discussed in Section 1.1.1 in many cases a single prototype may be unable to represent all the patterns of the corresponding class.

[3]A training pattern is said to be misclassified if the system attributes to it a label different from that provided by the teacher.

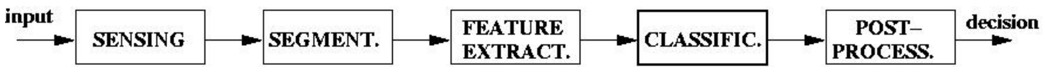


Figure 1.2: A pattern recognition system may be partitioned into components such as the ones shown above.

and classification. Figure 1.2 shows a more detailed diagram of the components of a typical pattern recognition system. A sensor converts physical inputs into signal data. In the segmentation step sensed objects are isolated each other and separated from the background. The feature extractor measures the object properties to be used for classification. The classifier uses the extracted features for labeling the sensed objects. Finally a post-processor takes into account other considerations, such as the effect of contexts and the costs of errors, in order to decide on the appropriate action. In the following each of these operations is described:

Sensing The input to a pattern recognition system is typically a transducer, e.g. a camera to be used for acquiring images or a microphone employed for acquiring sound. The difficulty of obtaining data from the external world strongly depends on the characteristics and limitations of the transducer (its bandwidth, resolution, sensitivity, distortion, signal- to-noise ratio, latency, etc.) and on the characteristics of the environment in which the data lie.

Segmentation and Grouping Segmentation is one of the deepest problems in pattern recognition. The purpose of segmentation is identifying the objects to be recognized into the raw data provided by the sensors. In computer vision, for example, this purpose is usually performed by partitioning the acquired image in a number of regions, so that each region corresponds to a recognizable object.

Closely related to the problem of pattern recognition is the problem of recognizing or grouping together the various parts of a composite object. For example, in OCR when the letter **E** is encountered it is crucial for the system to put together the strokes that form that letter.

Feature Extraction The goal of a feature extractor is to characterize the objects to be recognized by measurements whose values are likely very similar for those in the same class and very different for those in different ones. This leads to the idea of seeking for *distinguishing features* that are *invariant* to irrelevant transformations of the input. For example, the absolute position of an object identified in the acquired scene is irrelevant to the category of that object and thus the representation to be used should be insensitive to its absolute position.

Classification As mentioned above the task of a classifier is that of attributing the incoming descriptions provided by the feature extractor to one of the classes actually defined in the classification problem faced.

Post Processing Usually, a classifier is used in order to undertake an action depending on the category of the analyzed object. To each action a *cost* is associated . The post-processor uses the output of the classifier to decide on the action to be undertaken. Conceptually, the simplest measure of classifier performance is the classification error rate, i.e. the percentage of new patterns that are assigned to a wrong category. Thus, it is common to seek for minimum error-rate classification. However, it may be much better to recommend actions that will minimize the total expected cost, which is called the *risk*.

In order to improve the system performance, the post-processor can also be used to exploit the *context*, i.e. input-dependent information other than from the target pattern itself. For example, in optical character recognition a vocabulary can be used to help recognizing words.

1.3 The Evolutionary Computation Paradigm

Evolutionary Computation (EC) is a generic term used to indicate a class of population-based optimization algorithms that use mechanisms inspired by natural evolution, such as selection, reproduction, mutation and recombination, called *operators*. Candidate solutions to the optimization problem play the role of individuals in a population, and the cost function, usually called the *fitness* function, determines the environment within which the solutions "live". The evolution of the population takes place after the repeated application of the operators.
The basic elements of the EC paradigm are briefly described below:

1.3.1 Fitness

The fitness function measures the "goodness" of an individual as solution of the problem to be solved. In practice, such a function contains the whole available knowledge on the problem. This information is then exploited by the selection mechanism in order to choose the individuals, i.e. solutions, that will undergo genetic manipulation, performed by the operators, for producing the new population. Looking at this fact from a search space standpoint, the fitness function provides information for locating regions containing good solutions. As a consequence, the definition of an appropriate fitness function for the problem faced, is a key issue in the designing phase of any EC–based algorithm.

1.3.2 Selection

The selection mechanism, together with the genetic operators, represent those parts of an EC–based system responsible for the stochastic search in the solution space. In the evolutionary computation jargon, the individuals chosen by the selection mechanism are called *parents*, whereas those obtained after the manipulation carried out by the genetic operators are called *offspring*. The role of the selection mechanism is that of exploiting the information acquired so far in order to find better solutions. Such exploitation takes place favoring those individuals having a better fitness with respect to those with a worse fitness. Nevertheless, these are not completely excluded from the process generating the new population.

1.3.3 Solution Encoding

An important aspect of every EC–based algorithm is represented by the way the solutions are encoded as individuals in the population. The choice of a particular way for encoding the solution strongly affects the definition of the operators, whose role is just that of generating new solutions (individuals) from those the operators are applied to. Often in the EC literature, the structure used for encoding the individuals, which is manipulated by the recombination and mutation operators, is called *chromosome* or *genotype*.

Several EC schemes have been developed since the first of these schemes was devised in the 60' of the last century. These schemes differ each other just for the way the solutions are encoded as individuals in the population. In the following the main EC–based schemes and the corresponding encoding used will be described:

Genetic Algorithms

Genetic Algorithms (GAs) were first introduced by Holland [Hol92]. In GAs, solutions are encoded as bitstrings. Such kind of encoding can be used for representing a wide class of solutions. For example, a chromosome 120 bits long and such that each series of 12 consecutive bits encodes a normalized value in the range $[0.0, 1.0]$ could be used for representing the parameters that describe an airplane wing with particular properties.

As regards the operators, GAs have both a recombination and a mutation operator. The recombination operator is called *crossover*. Several versions of this operator have been devised. However, the simplest one divides each bitstring into two pieces and exchanges the first section of each bitstring. As regards mutation, instead, a common GA mutation strategy is that each bit is flipped to the opposite value, with some low probability, .

Genetic Programming

The field of program induction, using a tree-structured approach, was first clearly defined by Koza [Koz92]. In this approach, named Genetic Programming (GP), solutions are evolved in the form of Lisp programs in an evolutionary way that extends the concepts of the fixed–length representations used in GA. The evolved structures are specified as a combination of functions (arity > 0) and terminals (0–arity functions) which are combined in order to form Lisp programs. In order to apply GP to a specific problem, a specific set of functions $F = \{f_1, f_2, \ldots, f_n\}$ and also a specific set of terminals $T = \{t_1, t_2, \ldots, t_n\}$ have to be defined.

Also GP has both a recombination and a mutation operator. The recombination operator also called crossover creates two new individuals (i.e. programs) by swapping randomly chosen sub–trees between the two parent programs. The mutation operator, instead, modifies a single individual, modifying a randomly chosen sub-tree, with a newly generated one.

Evolutionary Programming

Evolutionary Programming (EP) has been developed by Fogel [FOW66] for evolving finite state machines (FSM) to be used to recognize or reject strings correctly according to some target regular language. The EP field has changed dramatically since its introduction and now the two main differences between EP and GA are that EP uses only mutation (while GAs also use crossover) and that in EP there is no restriction on a particular type of encoding to be used (as opposed to the bitstring representation so common in GA). The encoding to be chosen musts have just the property of representing FSM and of being suitable for a mutation operator, able to explore the FSM search space.

Evolutionary strategies

Evolution Strategies (ESs) were developed by Rechenberg and Schwefel at the beginnings of 70's of the last century [Rec73]. The motivation of this study was, from the beginning, to solve engineering design problems. In fact, Rechenberg and Schwefel developed ESs in order to conduct successive wing tunnel experiments for aerodynamic shape optimization. In ESs, solutions are usually encoded as vectors of floating point numbers rather than bitstrings.

In ESs, is used as recombination operator an operator called *intermediate recombination* in such a way that the vectors of two parents are averaged together, element by element, to form a new offspring. Mutation, instead, is performed by adding a random value from a Gaussian distribution (Gaussian mutation) to each element of an individuals vector, in order to create a new offspring.

1.3.4 Operators

Genetic operators explore new areas of the solution space searching for new and possibly better, if , solutions. The search is usually performed randomly, by modifying one or more individuals given in input. They are characterized by a quantity p which indicates the probability value according to which an operator is applied to one ore more individuals given in input. Hence, there is a probability $(1 - p)$ that the input individuals are passed unchanged in output. Note that as higher is p, as higher is the exploration of the search space performed by that operator.

Genetic operators can be grouped in two main classes, *recombination* and *mutation*. The first class consists of those operators that take as input two or more individuals and give as output the same, or even lesser, number of individuals. The effect of recombination operators is that of swapping parts of the individuals given in input. As regards mutation operators, instead, they take in input a single individual and yield as output a new individual obtained by randomly modifying the input one. These operators randomly explore the neighbor of the solution represented by the input individual.

1.4 A Reader's Guide to the Thesis

Chapter 2 introduces the basic concepts of supervised and unsupervised classification, as well as some of the most commonly used algorithms. Furthermore, some examples of application are also described.

Chapter 3 contains a survey on Evolutionary Computation, describing the basic terminology and methods. Particularly, genetic programming, genetic algorithms, breeder genetic algorithms and a new approach, devised for evolving graphs are illustrated.

In Chapter 4 the main contribute of this Thesis is presented: an Evolutionary Computation framework to be used to generate prototypes for any classifier. The proposed framework is quite general and can be employed any time a way for encoding the prototypes used by a given prototype can be defined. The more remarkable feature of this approach is likely its ability to automatically find the needed number of prototypes, without any addition of knowledge by the user. The only knowledge system exploits is that contained in the set of labeled patterns used in the training phase.
In this chapter are also reported two specific applications of the framework. The first one uses derivation trees in order to represent prototypes, consisting of logical expressions; the second one, in the second one, instead, prototypes are feature vectors.

Chapter 5 presents a new EC–based method for evolving graphs.

In Chapter 6 the experiments performed in order to evaluate the effectiveness of both the approaches proposed are reported. Specifically, for the framework devised, the behavior of both the applications described in the chapter 4 on several data sets have been investigated, performing many experiments. The method devised for evolving graphs has been tested on a hard non–linear optimization involving the design of wireless networks.
Moreover, for both the methods investigated, the comparison of the obtained results with those obtained by other approaches on the same problems, are also reported.

Finally, in Chapter 7 the conclusions and some ideas on the future work of the research presented in this Thesis are reported.

Chapter 2

An Overview of Pattern Recognition Algorithms

In this chapter an overview of pattern recognition algorithms is given. These algorithms can be divided in two main classes: classification and clustering. As we have seen in the previous chapter, a classification algorithm is able to recognizing the "category" to which a pattern belongs to. As concerns clustering algorithms, instead, they are able to divide the analyzed data into groups whose members are similar to each other and dissimilar to those belonging to the other groups, according to similarity criteria provided by the user.

Some applications of both classes of algorithms are also described in this chapter. In the last decades, classification algorithms have been increasingly used in a large variety of applications . Among the others it is possible mention: optical character recognition, medical applications, object recognition and document analysis. Some applications in the medical and biometric signals domains will be illustrated. Also clustering algorithms have been finding a large variety of applications in the last years. Here, applications in the data mining and image analysis domain will be described.

2.1 Classification Algorithms

In the most common and well-known classification algorithms, a pattern is described by a feature vector, in which each component is a real number.

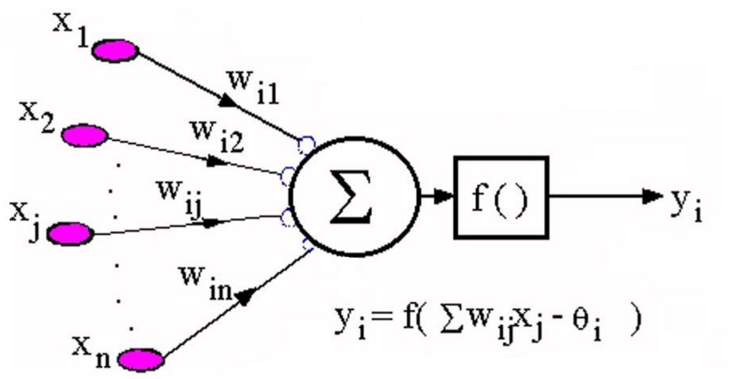


Figure 2.1: Scheme of a neuron.

2.1.1 k-nearest neighbor

The k-nearest neighbor decision rule (k-NN) is a widely used classification algorithm in statistical pattern recognition, which uses as information about the classes to be distinguished a set of labeled pattern feature vectors, i.e. each marked with the class it belongs to, this data set is usually called *training set*.
When an unknown vector has to be classified, its k closest neighbors are found from among all the patterns in the training set, and the class label is decided based on a majority rule. In order to avoid ties on class overlap regions, a odd value for k is chosen[1]. This rule is simple and elegant, yet the error rate is small in practice. In theory, it is known that, as the number of prototype patterns increases, the error rate asymptotically gets close to the optimal Bayes error rate and actually tends to it when k is increased. Because of this fact the k-NN rule has become a standard comparison method against which new classifiers are compared. When $k = 1$ the k–NN decision rule is called Nearest Neighbor (NN).

The major problem of using the k-NN decision rule is the computational complexity caused by the large number of distance computations required. For realistic pattern space dimensions, it is hard to find any variation of the rule that would be significantly lighter than the brute force method, in which all the distances between the unknown pattern vector and the training vectors are computed. Therefore, various modifications of the k-NN classifiers have been presented, often based on editing or pruning techniques by which the number of patterns may be decreased without losing accuracy.

[1]Note that this choice can avoid the occurrence of ties only if the number of classes present among the k-neighbors is less than three.

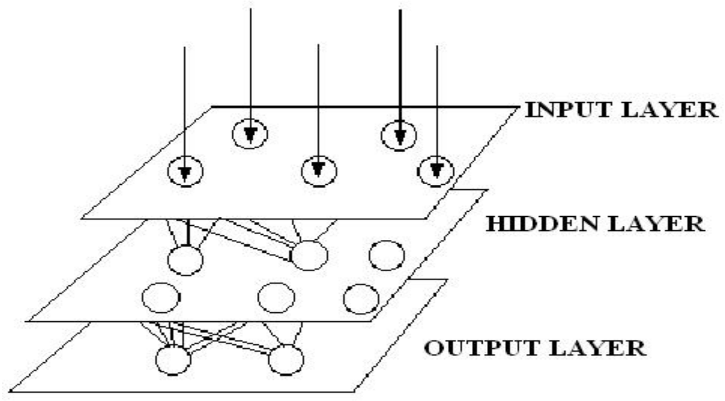


Figure 2.2: General scheme of a multilayer neural network.

2.1.2 Artificial Neural Networks

An Artificial Neural Network (ANN) is a system composed by multiple layers of simple processing elements called neurons (Figure 2.1). Each neuron is linked to certain of its neighbors with varying coefficients of connectivity (Figure 2.2). Each input is multiplied by a connection weight. In the simplest case, these products are simply summed, fed through a transfer function to generate a result, and then output. Even though all artificial neural networks are constructed from this basic building block, the fundamentals may vary in the blocks, so there are differences. Basically, all artificial neural networks have a similar topological structure. Some of the neurons interface the real world to receive inputs and other neurons provide the real world with the network outputs. All the rest of the neurons are hidden. As shown in figure 2.2, the neurons are grouped into layers. The input layer consists of neurons that receive inputs form the external environment. The output layer consists of neurons that communicate the output of the system to the user or external environment. There are usually a number of hidden layers between these two layers. When the input layer receives the input its neurons produce outputs, which in turn become inputs to the next layer of the system. The process continues until the output layer is invoked, providing its output to the external environment. There are different types of inter-layer connections (i.e., connections used between layers):

Fully connected Each neuron on the first layer is connected to every neuron on the second layer.

Partially connected A neuron on the first layer must not be necessarily connected to all the neurons on the second layer.

Feed forward The neurons on the first layer send their output to the neurons on the second layer, but they do not receive any input back from the neurons on the second layer.

Bi-directional There is another set of connections carrying the output of the neurons of the second layer into the neurons of the first layer.

Resonance The layers have bi-directional connections, and the neurons continue sending messages across the connections several times until a certain condition is achieved.

Recurrent The neurons within a layer are fully or partially connected among each other. As these neurons receive input from another layer, they must communicate their outputs among each other a number of times before they are allowed to send their outputs to another layer. Generally some conditions should be achieved among the neurons of a layer before they can send their outputs to another layer.

Neural networks are a kind of machine learning algorithms, because changing their connection weights (training) causes the network to learn the solution to a problem. The connection strength between neurons is stored as a weight-value for each specific connection. The system acquires new knowledge by adjusting these connection weights. The learning ability of a neural network is determined by its architecture and by the algorithmic method chosen for training. The most common training methods are:

Supervised This method requires a teacher that may be a training set of data or an observer who grades the performance of the network results.

Unsupervised The hidden neurons must find a way to organize themselves with no external aid. In this approach, no pattern outputs are provided to the network against which it can measure its predictive performance for a given vector of inputs.

A variety of learning laws are in common use. These laws are mathematical algorithms used to update the connection weights. Some of the major laws are described below:

Hebbs Rule This basic rule is: If a neuron receives an input from another neuron, and if both are highly active, the weight of the link between the neurons should be strengthened.

Hopfield Law This law is similar to the Hebbs Rule, with the exception that it specifies the magnitude of the strengthening or weakening. It states, "if the desired output and the input are both active or both inactive, increment the connection weight by the learning rate, otherwise decrement the weight by the learning rate."

Delta Rule It is one of the most commonly used. This rule is based on the idea of continuously modifying the strengths of the input connections to reduce the difference (delta) between the desired output value and the actual output of a neuron. This rule changes the connection weights by minimizing the mean squared error of the network. The error is propagated back into previous layers, one layer at a time. The process of back-propagating the network errors continues until the first layer is reached. This rule is also referred to as the Widrow-Hoff Learning Rule and the Least Mean Square Learning Rule.

Kohonen's Learning Law In this procedure, the neurons compete for the opportunity to learn, or to update their weights. The processing neuron with the largest output is declared the winner and has the capability of inhibiting its competitors as well as exciting its neighbors. Only the winner is permitted output, and only the winner plus its neighbors are allowed to update their connection.

The most successful applications of neural networks are in categorization and pattern recognition. Such a system classifies the object under investigation (e.g. an illness, a pattern, a picture, a chemical compound, a word, the financial profile of a customer, etc.) as one in a number of possible categories that, in return, may trigger the recommendation of an action such as a treatment plan or a financial plan. Image processing and pattern recognition form an important applicative area of neural networks, probably one of its most active research areas.

In particular, neural networks have been used for printed character and handwriting recognition. This area provides applications for banking, credit card processing and other financial services, where reading and correctly recognizing handwriting on documents is a crucial task. The pattern recognition capability of neural networks has been used to read handwriting in processing checks; the amount must normally be entered into the system by a human. Neural networks are also used in areas ranging from robotics, speech, signal processing, vision, character recognition to musical composition, detection of heart malfunction and epilepsy, fish detection and classification, optimization, and scheduling [Hay94].

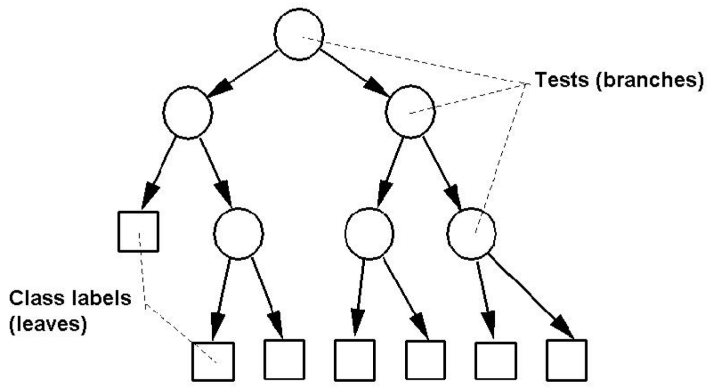


Figure 2.3: A structure of a typical decision tree.

2.1.3 Decision Trees

A *decision tree* classifier is a hierarchical structure where, at each level, a test is applied to one or more attribute values that may have one of two outcomes [Mit97]. The outcome may be a leaf, which allocates a class, or a decision node, which specifies a further test on the attribute values and forms a branch or subtree of the tree. Classification is performed by moving down along the tree until a leaf is reached. The structure of a decision tree classifier is shown in Figure 2.3.

The method for constructing a decision tree described by Quinlan [Qui93] is as follows: Given n classes denoted as $C_1, C_2, \dots C_n$, and a training set $\mathcal{D}_{tr}$, then

- if $\mathcal{D}_{tr}$ contains one or more objects all belonging to a single class C_j, then the decision tree is a leaf identifying class C_j.
- if $\mathcal{D}_{tr}$ contains no objects, the decision tree is a leaf determined from information other than $\mathcal{D}_{tr}$.
- if $\mathcal{D}_{tr}$ contains objects belonging to a mixture of classes, then a test is chosen, based on a single attribute, that has one or more mutually exclusive outcomes $\mathrm{Out}_1, \mathrm{Out}_2, \dots, \mathrm{Out}_n$. $\mathcal{D}_{tr}$ is partitioned into subsets $\mathcal{D}_{tr_1}, \mathcal{D}_{tr_2}, \dots, \mathcal{D}_{tr_n}$, where $\mathcal{D}_{tr_i}$ contains the objects in $\mathcal{D}_{tr}$ whose outcome of the chosen test is Out_i. The same method is applied recursively to each subset of training objects.

Quinlan's decision tree classifier, denoted in the literature as C4.5, uses tests based on a single attribute value. That is, decision boundaries are parallel to

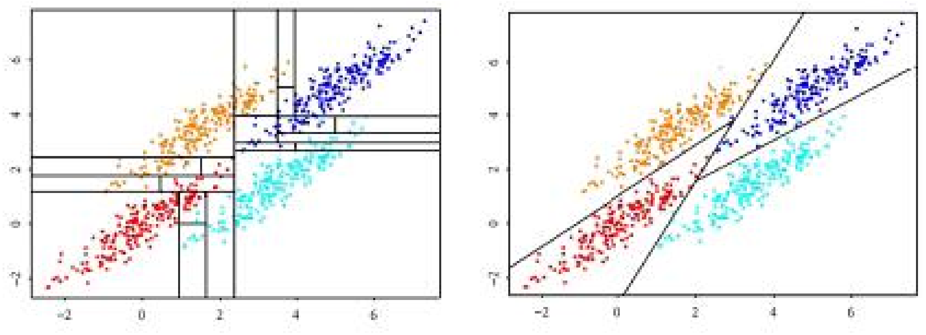

(a) (b)

Figure 2.4: Given axes that show the attribute values and colors corresponding to class labels (a) and (b), respectively axis-parallel and oblique decision boundaries.

the attribute axes, such as the decision regions shown in Figure 2.4(a)). Other tree classifiers may use more than one attribute value. For instance, oc1 in [MKS94], was designed specifically to produce decision trees with oblique (linear) decision boundaries like those shown in Figure 2.4(b). Oblique decision boundaries can be an advantage in examples such as that shown in Figure 2.4(b), where the natural class regions can be approximated using only 3 oblique decision boundaries compared to 19 axis-parallel boundaries.

2.2 Classification Applications

As already said, classification algorithms have been used in a large variety of domains in the last three decades. Among the huge number of explored domains, it is possible to mention: optical character recognition, medical applications, biometric signal recognition, object recognition, database indexing and document analysis. This Section analyzes applications belonging to a couple of fields that also today play an extremely relevant role: medical applications and biometric signal recognition. In fact, the relevance of computer-aided interpretation of medical examination is increasing with the diffusion of mass screening, while the detection of biometric signals is a currently explored field, mainly for people authentication.

Four applications are analyzed in the following. The first two applications deal with biometric signals: fingerprints recognition and of the speaker identification. The other two applications rise to the medical domain: the diagnosis of cardiac pathologies based on electrocardiogram analysis and the diagnosis of breast cancer based on the analysis of mammograms.

2.2.1 Fingerprint Classification

Large volumes of fingerprints are collected and stored every day in a wide range of applications, including forensic use, access control, and drivers license registration. Automatic identification based on fingerprints requires the input fingerprint to be matched with a large number of fingerprints stored in a database. Fingerprint classification is a technique used to assign a fingerprint to one of the several prespecified types already established in the literature [DDAS03]. Fingerprint classification can be viewed as a coarse-level matching of the fingerprints. An input fingerprint is first matched to one of the pre-specified types and then compared to a subset of the database corresponding to that fingerprint type. For example, if the fingerprint database is binned into five classes, and a classifier assigns a fingerprint to two possible classes (primary and secondary) with extremely high accuracy, then the identification system will only need to search two of the five bins, thus decreasing the search space. Unfortunately, only five major fingerprint categories have been identified, the distribution of fingerprints within these categories is not uniform, and there are many ambiguous fingerprints, whose exclusive membership cannot be reliably stated even by human experts. In fact, the definition of each fingerprint category is both complex and vague. A long-time experience is required for a human inspector to achieve a satisfactory level of performance in fingerprint classification. Therefore, in practice, fingerprint classification is not immune from errors and does not offer much selectivity for fingerprint searching in large databases. To overcome this problem, methods based on feature vectors have been proposed for indexing : fingerprints are not partitioned into non-overlapping classes, but each fingerprint is characterized by means of a numerical vector summarizing its main features.

2.2.2 Speaker Identification

Speaker identification is a special case of the more general problem of speaker recognition. In the case of speaker identification the goal is to determine which voice, in a group of known voices, best matches the input voice patterns. If the speaker to be identified is not required to pronounce a specific set of phrases the system is said to be text-independent.

The speaker identification problem has been addressed in the literature by representing the audio signal using different features, calculated in the frequency or in the time domain. Moreover, different classification paradigms have been employed, basically neural networks. One of the most frequently quoted systems is illustrated in [DR95] . It uses features evaluated in the frequency domain by the cepstral analysis. Different classification paradigms are proposed and the best results obtained with a Gaussian mixture model. Other authors have proposed the use of acoustic features directly obtainable from the time domain, such as pitch, speech rate, voice quality and temporal variation of the audio signal. This is the case of [MHT^{+}01], where a system exclusively dedicated to the recognition of the voiced segments in the audio track is proposed.

2.2.3 Medical Applications

A large number of medical classification problems have been faced using automatic methods. This Section describes two examples of classification problems faced using neural networks. The first application concerns electrocardiography automatic interpretation while the second is the aided diagnosis of breast cancer using thoracic x-rays.
Electrocardiography (ECG) has a basic role in cardiology since it consists of effective, simple, non-invasive, low-cost procedures for the diagnosis of cardiovascular disorders that have a high epidemiological incidence and are very relevant for their impact on patient life and on social costs. Pathological alterations observable by ECG can be divided into three main areas:

1. cardiac rhythm disturbances (or *arrhythmia*);
2. dysfunction of myocardial blood perfusion (or *cardiac ischemia*);
3. chronic alteration of the mechanical structure of the heart (for instance *left ventricular hypertrophy*).

The literature in this topic reports several approaches to classification. In general, past approaches, according to published results, seem to suffer from common drawbacks that depend on high sensitivity to noise and unreliability in dealing with new or ambiguous patterns [Tal83]. Artificial neural networks have often been proposed as tools for realizing classifiers that are able to deal even with nonlinear discrimination between classes and to accept incomplete or ambiguous input patterns. In [SM98], three different architectures have been proposed for carrying out the classifiers:

- An ANN is implemented to perform arrhythmia detection because of its capability to reject unknown or ambiguous patterns. For this purpose, two uncertainty criteria are introduced and evaluated.

- Both a static and a recurrent ANN approach are implemented in several architectures to detect ischemic episodes. While the first approach features an easier learning process, the second one is able to learn the input signal evolution even on a reduced training set.

- Recognition of chronic myocardial diseases requires a three step procedure. The parameters chosen for analyzing the ECG have been fuzzy processed by a layer of normalized radial basis functions and then have been analyzed by a neural network; finally, a pruning technique is applied to reduce the network size and to improve its generalization capability.

The second application considered here is early breast cancer detection [ZSQ02]. Breast cancer is one of the leading causes of cancer deaths among women. A woman has a 12% chance of developing breast cancer and a 3.5% chance of dying from this disease over her lifetime. Based on the recent news in this research area ("Study Backs New Mammography Technique", November 28, 2000 on Reuters), in the largest study up to date, researchers at Women's Diagnostic and Breast Health Center in Plano, TX screened 12,860 women for breast cancer utilizing computer-aided diagnosis (CAD) to interpret each mammogram. A total of 49 unsuspected cancers were detected, 32 by both CAD and the radiologist, nine by the radiologist alone and eight only with the help of CAD. This shows that by using CAD in the interpretation of mammograms, the number of cancers detected has increased by about 20%. Moreover, all eight additional cancers were in the early stages, when they are most easily treated. This shows the real significance of early detection of micro-calcification clusters (MCCs) in digital mammography.

Clinically, MCCs are described as the presence of small deposits of calcium usually arranged in a cluster. They can be found as some poorly defined masses, architectural distortions, asymmetrical structures or also as some developing density or isolated ducts. Such diverse descriptions of MCCs provide different approaches to detect MCCs in different CAD schemes. The reported methods include both statistical and non-statistical approaches [ZSQ96], binary decision trees, and back-propagation neural networks. Many approaches have been developed to face the problem. For instance a shift-invariant artificial neural network approach has been developed to reduce the false positive detection rate of MCCs by means of a rule-based CAD scheme [ZQC96]. The feature set used was mainly derived from the spatial domain using raw image data. An improved method was developed by using a mixed feature set neural network with a spectral entropy decision algorithm [ZSQ02], and the feature set derived from both spatial and morphology domains. It is usually very difficult to adjust the balance between network complexity and learning accuracy for most real world problems. If too much emphasis is put on network complexity, a poorly learned compact neural network will evolve. On the contrary, when too much emphasis is put on learning accuracy, large neural networks are usually obtained, which yield very good results on the training data, but they usually achieve unsatisfactory performances on unknown data. In [XY99], sizing a neural network by using evolutionary algorithms is discussed.

2.3 Clustering Algorithms

Clustering can be defined as a process that organizes objects into groups or *clusters*, whose members are similar in some way. A cluster is therefore a collection of objects which are "similar" between them and are "dissimilar" from the objects belonging to other clusters.

The goal of clustering is to determine the intrinsic grouping in a set of unlabeled data. But how to decide what constitutes a good clustering? It can be shown that there is no absolute best criterion which would be independent of the final aim of the clustering. Consequently, it is the user which must supply this criterion, in such a way that the result of the clustering will suit his needs. For instance, we could be interested in finding representatives for homogeneous groups (data reduction), in finding "natural clusters" and describe their unknown properties ("natural" data types), in finding useful and suitable groupings ("useful" data classes) or in finding unusual data objects (outlier detection).

A brief description of the most common and well-known clustering algorithms is provided in the following. In all the proposed methods each pattern is described by a feature vector, in which each component is a real number.

2.3.1 K-means

Given a population of patterns and a number K of groups or classes, the final goal of this algorithm is partitioning the given population in K different classes such that each class has a center which is the mean position of all the patterns in that class, and each pattern is in the class whose center is closest to it. The inputs of the algorithm are: the number of pattern feature vectors, the needed number of classes K, and K means chosen at random.
After the inputs are provided, a loop starts until a termination condition is met. This loop consists of two steps. In the first step, each pattern is assigned to a class such that the Euclidian distance from this pattern to the

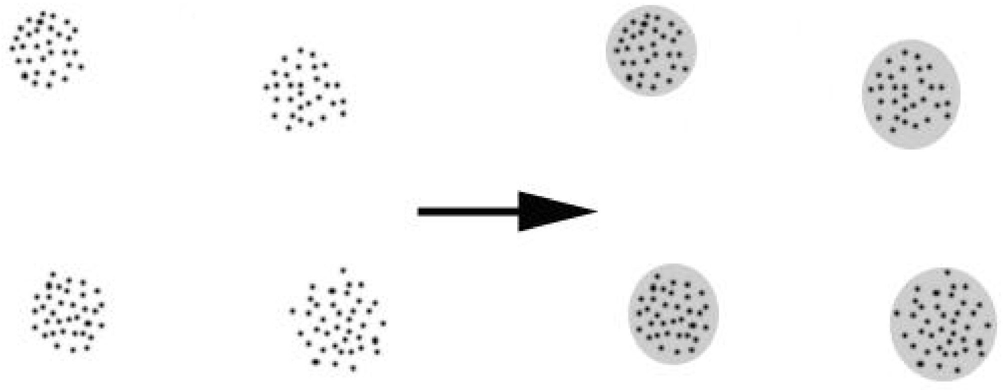


Figure 2.5: An example of clustering. In this case 4 clusters into which the data can be divided can be easily identified; the similarity criterion is distance: two or more objects belong to the same cluster if they are "close" according to a given distance (in this case geometrical distance). This is called *distance-based clustering.*

center of that class is minimized; in the second step, the mean of each class is calculated, based on the patterns belonging to that class. The quality of the result of a K-means clustering is characterized by two parameters:

- *Compacteness*: low within class variance;

- *Isolation*: high distance between class centers;

Theoretically, K-means should terminate when no more patterns are changing classes (see Figure 2.5). There are proofs of termination for K-means. These rely on the fact that both steps of K-means reduce the variance. Thus, eventually, there is no move to make that will further reduce the variance. Running to completion (i.e. no pattern changes its class) may require a large number of iterations. In many cases, the algorithm is terminated when one of the following criteria is met:

- A number of patterns is fixed. If a smaller number of patterns change class, the algorithm terminates;

- After a fixed number of iterations, the algorithm terminates;

One of the main problems of this algorithm is that the classification result depends on the initialization of the algorithm. If the K initial random means are changed, it is possible that also the final result will be different. In

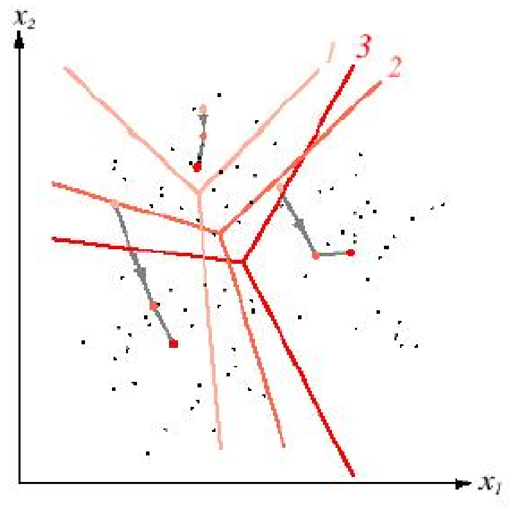


Figure 2.6: Trajectories for the centroids of the K-means clustering procedure applied to two-dimensional data. The final tessellation is also shown. The centroids correspond to the centers of the cells. In this case, convergence is obtained in three iterations.

the original K-means algorithm the initial means are computed randomly. Nevertheless, random computation can be substituted with more efficient procedures that usually are application-dependent.

2.3.2 Hierarchical Clustering

Hierarchical clustering is a bottom-up clustering method where clusters have subclusters, which in turn have sub-clusters, and so on. The classic example of this kind of clustering is species taxonomy. Agglomerative hierarchical clustering starts with every single pattern in a single cluster. Then, at each successive iteration, it merges the closest pair of clusters by satisfying some similarity criteria, until all the data is in one single cluster.
The hierarchy within the final cluster has the following properties:

- clusters generated in early stages are nested in those generated in later stages.

- clusters with different sizes in the tree can be valuable for discovery.

After the input patterns are provided, each pattern is assigned to a separate cluster. Firstly, all pair-wise distances between clusters are evaluated, then a distance matrix is constructed using the distance values. After that, a loop

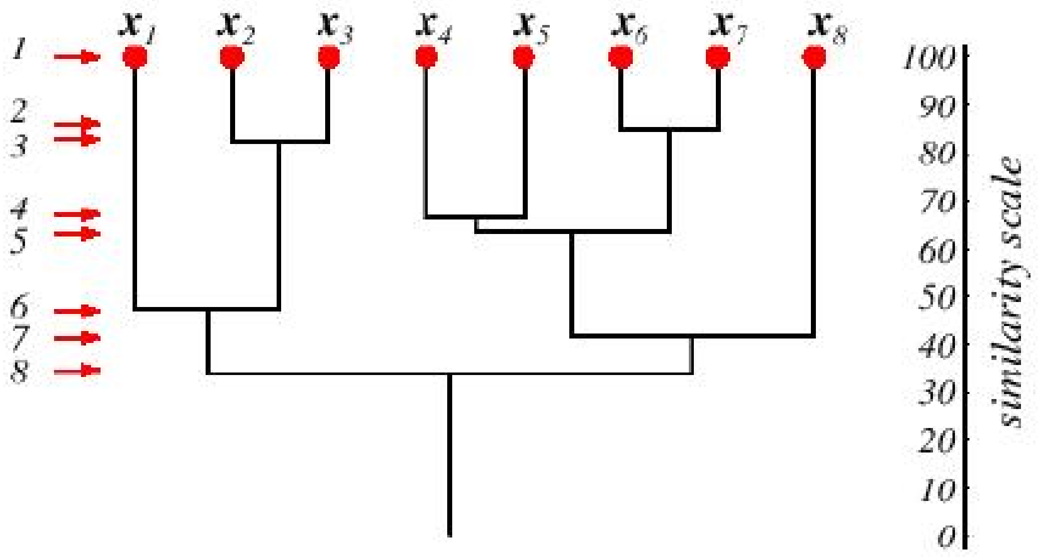


Figure 2.7: A dendrogram can represent the results of hierarchical clustering algorithms. The vertical axis shows a generalized measure of similarity among clusters. Here, at level 1 all eight points lie in singleton clusters; each point in a cluster is highly similar to itself, of course. Points x_6 and x_7 happen to be the most similar, and are merged at level 2, and so on.

starts until a termination condition is met. During each iteration of this loop, the pair of clusters with the shortest distance is looked for. The selected pair is removed from the matrix and the two clusters are merged. The distance among the new cluster and all the others are computed and the matrix is updated.

In the original version of the algorithm the loops end when the distance matrix is composed of only one element. The most natural representation of hierarchical clustering is a corresponding tree, called *dendrogram*, which shows how the patterns are grouped. Figure 2.7 shows a dendrogram for a simple problem involving eight patterns. Level 1 shows the eight patterns as singleton clusters. At level 2 patterns x_6 and x_7 have been grouped to form a cluster, and they stay together at all subsequent levels. Usually, in dendrogram representations, similarity scale are drawn in order to give a similarity measure among the grouped clusters.

Another representation for hierarchical clustering is based on sets, in which each level of a cluster may contain sets that are subclusters, as shown in Figure 2.8. The main disadvantage is that no provision can be made for a relocation of objects that may have been incorrectly grouped at an early

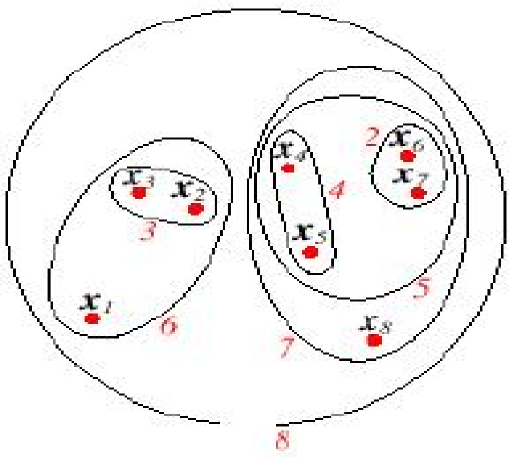


Figure 2.8: A set or Venn diagram representation of twodimensional data reveals the hierarchical structure but not the quantitative distances between clusters. The levels are numbered in red.

stage. Moreover, the use of different distance metrics for measuring distances between clusters may generate different results. The more common metrics used for the computation of cluster distance are detailed below.

Nearest Neighbor (minimum algorithm).

Let X_i and X_j respectively be the i–th and the j–th cluster; the distance function used for measuring the distance between two clusters is:

$$d_{\min}(X_i, X_j) = \min_{\substack{\overline{x} \in X_i \\ \overline{x}' \in X_i}} \|\overline{x} - \overline{x}'\|$$

When $d_{\min}(\cdot, \cdot)$ is used to measure the distance between subsets, the nearest-neighbor nodes determine the nearest subsets. In other words, the merging of X_i and X_j corresponds to the fusion of two clusters having the two closest elements. This kind of metrics roughly tends to produce “elongated” clusters.

Farthest Neighbor (maximum algorithm).

In this case the distance function used for measuring the distance between two clusters is:

$$d_{\min}(X_i, X_j) = \max_{\substack{\overline{x} \in X_i \\ \overline{x}' \in X_i}} \|\overline{x} - \overline{x}'\|$$

When $d_{\max}(\cdot,\cdot)$ is used to measure the distance between subsets, the distance is determined by the most distant nodes in the two clusters. This kind of metrics tends to merge the clusters minimizing the increment of the diameter of the clusters themselves, thus obtaining clusters compact and roughly equal in size.

Other Metrics.

The minimum and maximum measures represent two extremes in measuring the distance between clusters, and in some problems can lead to unsatisfactory results. The use of averaging is a possible way to improve the quality of clustering, and

$$d_{avg}(X_i, X_j) = \frac{1}{|X_i| \cdot |X_i|} \sum_{\overline{x} \in X_i} \sum_{\overline{x}' \in X_i} \|\overline{x} - \overline{x}'\|$$

$$d_{mean}(X_i, X_j) = \|mean(X_i) - mean(X_j)\|$$

can be more convenient metrics in the general case.

2.3.3 Self-Organizing Maps

A Self-Organizing Map (SOM) is a neural network algorithm introduced by Kohonen [Koh82, Koh89] in 1982. It is used to categorize and interpret large, high-dimensional data sets.
Briefly, it operates mapping n-dimensional data points that are similar to each other onto nearby regions of a q-dimensional space; q is much smaller than n (usually $q = 2$). The map is an array of nodes (also called neurons), usually twodimensional, but also of higher order; it is often laid out in a rectangular lattice. Each node has an associated reference vector of the same size as the input feature vector. The input vectors are compared to these reference vectors. A basic description of the SOM algorithm is given in the following:
At the beginning, the reference vectors of all the nodes are randomly initialized. Then a loop starts until a termination condition is met. This loop consists of the following steps:

- An input vector is randomly selected from the input set.

- Using a metrics (e.g., Euclidean) the input vector is compared to every reference vector.

- The node whose reference vector is the best match is chosen as the winning node for that particular input vector.

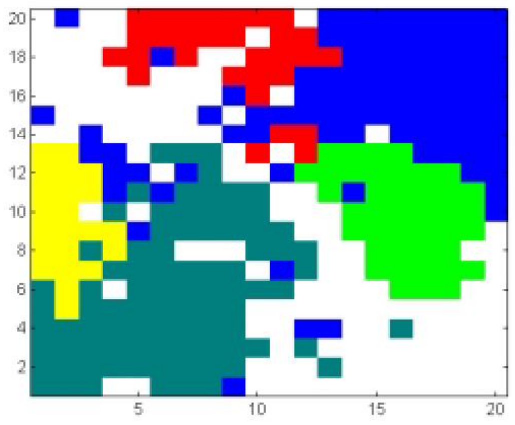


Figure 2.9: Distribution of neurons in a SOM. Different colors indicate the appurtenance to different classes.

- The neighboring nodes (i.e., nodes which are topographically close in the array) to the winning node are then updated by a certain amount so that they are more similar to the input vector.

The purpose of the last step is to make both the winning node and the nodes in the winning neighborhood responding more favorably to inputs similar to the input vector. This is how topologically close regions of the output map gain an affinity for clusters of similar data vectors. The loop is repeated for all the input vectors. This is called an epoch, or a time-step. There are different views on how many time-steps the algorithm should be repeated (training). Experimentally it has been seen that the optimal number depends on the application, but usually the number of epochs varies in the range $[1000 \;\; 100,000]$.

During the training process, the neurons assume a configuration determined from the input data (see Figure 2.9). After the training is terminated, the SOM can be used as a classifier: when an input pattern is submitted to the SOM, only the most similar neuron will be activated and will provide its output. If a set of input patterns, whose class is known, is submitted to the trained SOM, then it is possible to associate the neurons to the classes of input data.

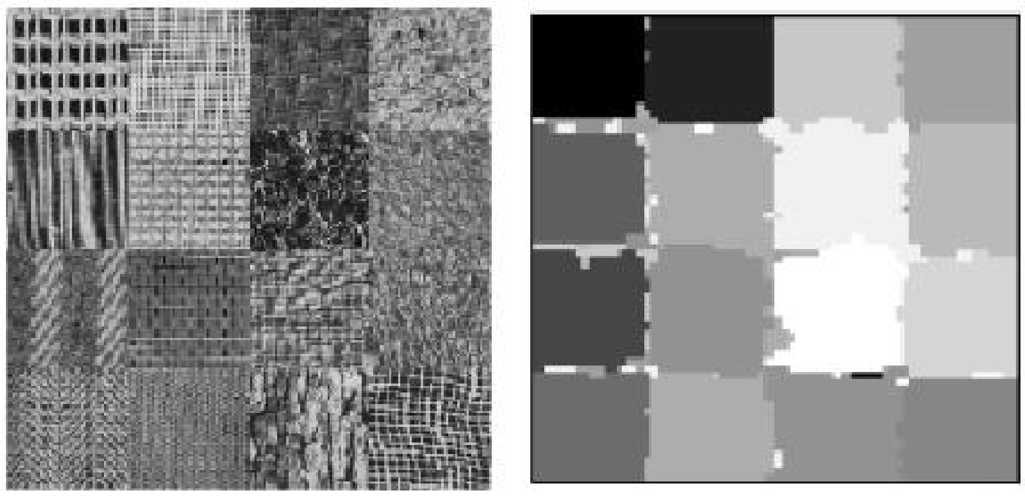


Figure 2.10: Texture clustering.

2.4 Clustering Applications

Clustering algorithms have been used in a large variety of applications. In this section, four applications where clustering has been employed as an essential step are described. These areas are: image segmentation, object and character recognition, document retrieval, and data mining.

2.4.1 Image Segmentation

Image Segmentation is a fundamental component in many computer vision applications, and can be addressed as a clustering problem [RK82]. The segmentation of the images presented to an image analysis system is critically dependent on the scene to be sensed, the imaging geometry, configuration, and sensor used to transduce the scene into a digital image, and ultimately on the desired output (goal) of the system. The applicability of the clustering methodology to the image segmentation problem was recognized over three decades ago, and the paradigms underlying the initial pioneering efforts are still in use today. A recurring theme is defining feature vectors at every pixel as composed of both functions of image intensity and functions of the pixel location itself. An image segmentation is typically defined as an exhaustive partitioning of an input image into regions, each of which is considered to be homogeneous with respect to some image property of interest (e.g., intensity, color, or texture). In this case a segment represents a region of the image and the segmentation problem consists in separating the image into a set of non-overlapping segments that cover the whole image.

2.4.2 Object Recognition

The use of clustering to group views of 3D objects for the purposes of object recognition in range data was described in [DJ95]. The term view refers to a range image of an object obtained from any arbitrary viewpoint. The system under consideration employed a viewpoint dependent (or view-centered) approach to the object recognition problem; each object to be recognized was represented in terms of a library of range images of that object. There are many possible views of a 3D object and one goal of that work was to avoid matching an unknown input view against each image of each object. A common theme in object recognition literature is indexing, wherein the unknown view is used to select a subset of views of a subset of the objects in the database for further comparison, and rejects all other views of the objects. One of the approaches to indexing employs the notion of view classes; a view class is the set of qualitatively similar views of an object. In that work, the view classes were identified by clustering; the rest of this subsection briefly outlines the technique. Object views were grouped into classes based on the similarity of shape spectral features. Each input image of an object viewed in isolation yields a feature vector which characterizes that view. The feature vector contains the first ten central moments of a normalized shape spectral distribution of an object view. The shape spectrum of an object view is obtained from its range data by constructing a histogram of shape index values (which are related to surface curvature values) and accumulating all the object pixels that fall into each bin. By normalizing the spectrum with respect to the total object area, the scale differences that may exist between different objects are removed. Given a set of object representations that describes m views of the i–th object, the goal is to derive a partition of the views. Each cluster contains those views of the i-th object that have been deemed similar based on the dissimilarity between the corresponding moment features of the shape spectra of the views. A database is used, containing 3,200 range images of 10 different sculpted objects with 320 views per object. The range images were synthesized from 320 possible viewpoints of the objects. The views of each object are clustered, based on the dissimilarity measure between their moment vectors using a hierarchical clustering scheme. The resulting clustering demonstrates that the views of each object fall into several distinguishable clusters. The barycenter (also called centroid) of each of these clusters was determined by computing the mean of the moment vectors of the views falling into the cluster. In [DJ95] it has been demonstrated that this clustering-based view grouping procedure facilitates object matching in terms of both classification accuracy and the number of matches necessary for correct classification of test views. Object views are grouped into compact and homogeneous view clusters, thus demonstrating the power of the cluster-based scheme for view organization and efficient object matching.

2.4.3 Character Recognition

Clustering was employed in [SA98] to identify lexemes in handwritten text for the purposes of writer-independent handwriting recognition. The success of a handwriting recognition system is vitally dependent on its acceptance by potential users. Writer-dependent systems provide a higher level of recognition accuracy than writer-independent systems, but require a large amount of training data. A writer-independent system, on the other hand, must be able to recognize a wide variety of writing styles in order to satisfy an individual user. As the variability of the writing styles that must be captured by a system increases, it becomes more and more difficult to discriminate between different classes due to the amount of overlap in the feature space. One solution to this problem is to separate the data from these disparate writing styles for each class into different subclasses, known as lexemes. These lexemes represent portions of the data which are more easily separated from the data of classes other than the one which the lexeme belongs to. In [SA98], handwriting is captured by digitizing the position of the pen and the state of the pen point (up or down) at a constant sampling rate. After some resampling, normalization, and smoothing, each stroke of the pen is represented as a variable-length string of points. A metric based on elastic template matching and dynamic programming is defined to calculate the distance between two strokes. Using the distances calculated in this way, a proximity matrix is constructed for each class of digits (i.e., 0 through 9, see 2.11). Each matrix measures the intraclass distances for a particular digit class. Digits in a particular class are clustered in an attempt to find a small number of prototypes.

Clustering is done using the K-means algorithm for different values of K. The value maximizing the interclass distance and minimizing the intraclass distance is assumed to be the best. As expected, the mean squared error (MSE) decreases monotonically as a function of K. The optimal value of K is chosen by identifying a knee in the plot of MSE vs. K. When representing a cluster of digits by a single prototype, the best on-line recognition results were obtained by using the digit that is closest to that cluster center. Using this scheme, a correct recognition rate of 99.33% was obtained.

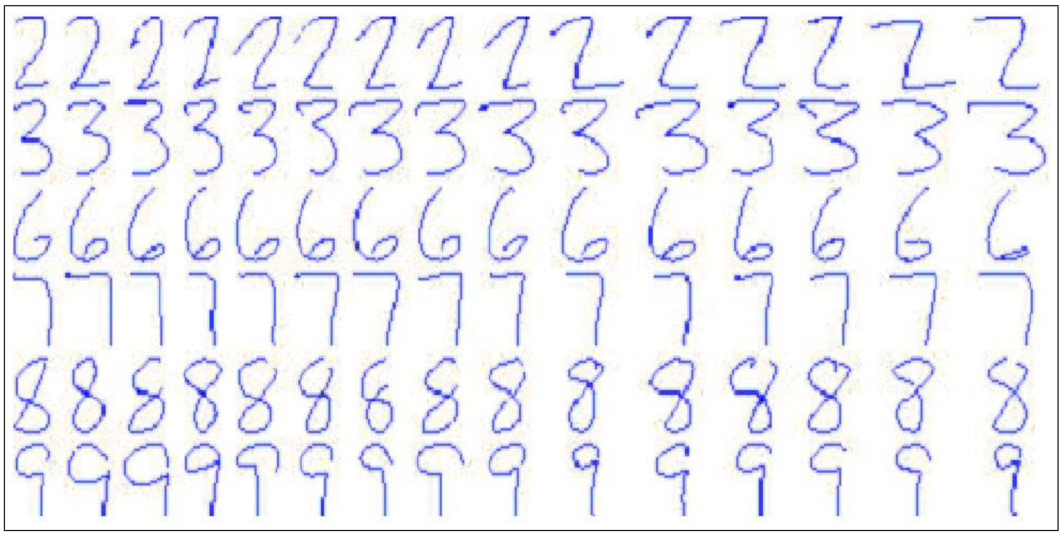


Figure 2.11: Clusters of digits.

2.4.4 Information Retrieval

Information retrieval (IR) is concerned with automatic storage and retrieval of documents [Ras92]. Many university libraries use IR systems to provide access to books, journals, and other documents. Libraries use the Library of Congress Classification (LCC) scheme for efficient storage and retrieval of books. The LCC scheme consists of classes labeled from A to Z, which are used to characterize books belonging to different subjects. There are several problems associated with the classification of books using the LCC scheme. Some of these are listed below:

- LCC number alone may not be able to retrieve all the relevant books: the classification numbers assigned to the books do not contain sufficient information regarding all the topics covered in a book.
- There is an inherent problem in assigning LCC numbers to books in a rapidly developing area. Multiple labels for books dealing with the same topic will force their placement on different stacks in a library.
- Assigning a number to a new book is a difficult problem. A book may deal with topics corresponding to two or more LCC numbers, and therefore, assigning a unique number to such a book is difficult.

[MJ95] describes a knowledge-based clustering scheme to group representations of books, which are obtained using the ACMCR (Association for

Computing Machinery Computing Reviews) classification tree. This tree is used by the authors contributing to various ACM publications to provide keywords in the form of ACMCR category labels. It consists of 11 nodes at the first level and each node has a label that is a string of one or more symbols. For example, I515 is the label of a fourth-level node in the tree. The clustering problem can be stated as follows. Given a collection of books, a set of clusters is needed. A hierarchical clustering algorithm has been used and a threshold value has been chosen such that the largest gap in the dendrogram is the one between the levels at which six and seven clusters are formed. An examination of the subject areas of the books in these clusters revealed that the clusters obtained are indeed meaningful.

2.4.5 Data Mining

Searching for useful nuggets of information among huge amounts of data has become known as the field of data mining. Data mining can be applied to relational, transactional, and spatial databases, as well as large stores of unstructured data such as the World Wide Web. Data mining is an exploratory activity, so clustering methods are well suited for it. Clustering is often an important initial step of many in the process of data mining. Some of the data mining approaches which use clustering are database segmentation, predictive modeling, and visualization of large databases.

Segmentation. Clustering methods are used in data mining to segment databases into homogeneous groups. This can serve purposes of data compression (working with clusters rather than with individual items), or of identifying characteristics of subpopulations which can be targeted for specific purposes (e.g., marketing aimed at senior citizens).

Predictive Modeling. Statistical methods of data analysis usually involve hypothesis testing of a model the analyst already has in mind. Data mining can aid the user in discovering potential hypotheses prior to using the statistical tool. Predictive modeling uses clustering to group items, then infers rules to characterize the groups and suggest models. For example, magazine subscribers can be clustered based on a number of factors (age, sex, income, etc.), then the resulting groups can be characterized in an attempt to find a model which will distinguish those subscribers that will renew their subscriptions from those that will not.

Visualization. Clusters in large databases can be used for visualization, in order to aid human analysts in identifying groups and subgroups that have similar characteristics.

Chapter 3

Evolutionary Computation

In nature, evolution is mostly determined by natural selection caused by the competition among different individuals for the resources available. In this competition, the fittest individuals are more likely to survive and propagate their genetic material trough future generations. This natural phenomena has been largely studied by computer scientists since the last years of the 50's of the last century [Fra57, Fri59, FDN59]. Those scientist sensed that so as natural evolution has been able to evolve highly complex structures, e.g. plants and animals, algorithms simulating this natural process could be devised in order to solve problems requiring complex and hard to find solutions. The result of this insight has been a new computation paradigm, largely used to implement several different algorithms, able to find adequate solutions for many problems.

In this chapter the basic concepts of the evolutionary computation paradigm are presented. In the first section the main advantages of EC–based algorithms while compared to other global optimization techniques are discussed. In the second section the basic concepts of the evolutionary computation paradigm are introduced, while the next two section are devoted to the description of the two main classes of EC–based algorithms: Genetic Algorithms and Genetic Programming. In section 3.5 a particular type of Genetic Programming, based on the concept of context-free grammar, is described. Finally, some applications that use EC–based algorithms are listed.

3.1 Advantages of Evolutionary Computation

EC–based algorithms have been successively used in a wide range of applications: optimization task, image analysis, data mining, etc. In order explain, at least in part, this success in the following some of the advantages of using EC–based algorithms, while compared to other global optimization techniques [Fog99], are listed:

1. The performances of any EC–based algorithm are representation independent, in contrast to other numerical techniques, which might be applicable only for continuous values or other constrained sets.

2. The EC paradigm offers a framework such that it is comparably easy to incorporate the available knowledge about the problem. The possibility of incorporating such information in an easy way, allows to better focus the evolutionary search, yielding a more efficient exploration of the state space of possible solutions.

3. EC–based algorithms can also be combined with more traditional optimization techniques. This may be as simple as the use of a gradient minimization after primary search with an evolutionary algorithm (e.g. fine tuning of weights of an evolutionary neural network) or it may involve simultaneous application of other algorithms (e.g. hybridizing with simulated annealing or Tabu search to improve the efficiency of basic evolutionary search).

4. The evaluation of each solution can be handled in parallel and only selection (which requires at least pair-wise competition) requires some serial processing. On the contrary, implicit parallelism is not possible in many global optimization algorithms like simulated annealing and Tabu search.

5. Traditional optimization methods are not robust with respect to the dynamic changes in the problem of the environment and often require a complete restart in order to provide a solution (e.g. dynamic programming). In contrast, evolutionary algorithms can be used to adapt solutions to changing circumstance.

6. Perhaps, the greatest advantage of evolutionary algorithms comes from the ability to address problems for which the available knowledge is inadequate to develop solving strategies that give satisfactory solutions. In these cases EC provide a general heuristic that may be able to find satisfactory solutions.

3.2 The Evolutionary Algorithm Paradigm

The term *natural evolution* is generally used to indicate the process that has transformed the community of living beings populating the earth from a set of simple unicellular organisms to a huge variety of living species, each integrated in the surrounding environment. The laws which guided such a tremendous development through the geological era, until the creation of the most complex species ever known, the human one, are essentially still unknown. Several attempts for explaining this extraordinary and astonishing natural phenomena have been done in the past. The most accredited scientific theory on the evolution of species is, still nowadays, due to Darwin [Dar59]:

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

The great merit of Charles Darwin has been that of identifying a small set of essential elements to rule evolution by natural selection: reproduction of individuals, variation phenomena that affect the likelihood of survival of individuals, inheritance of many of the parents' features by offspring in reproduction and the presence of a finite amount of resources causing competition for survival between individuals.

These simple features – reproduction, likelihood of survival, variation, inheritance and competition – are the bricks that build the simple model of evolution employed by computer scientists to define a new computational scheme. This model is able to solve difficult problems, so as NP-hard optimization problems or machine learning problems, in which solutions are represented by complex models able to represent objects belonging to different classes. This natural phenomena inspired computational schema is known as *evolutionary computation* (EC).

Most of the terms used in the EC jargon have been borrowed by biology. The description of a living being is written in a series of chromosomes: most multi-celled living beings have many chromosomes in their DNA (RNA in the cases of some organisms). In EC the genetic code for an *individual* to be evolved is simply called its *chromosome* (singular). Each chromosome is composed of many genes. The different possible states of these genes, in biology, are referred to as the alleles. The field of EC has collapsed this distinction and simply describes the chromosome under evolution as comprised of many alleles.

There is an important distinction between the genetic code of an organism and the organism itself. Though both of them are evolved simultaneously and are inexorably linked, in some cases there may be important reasons for this distinction to exist in EC. In biology, the encoding of an organism

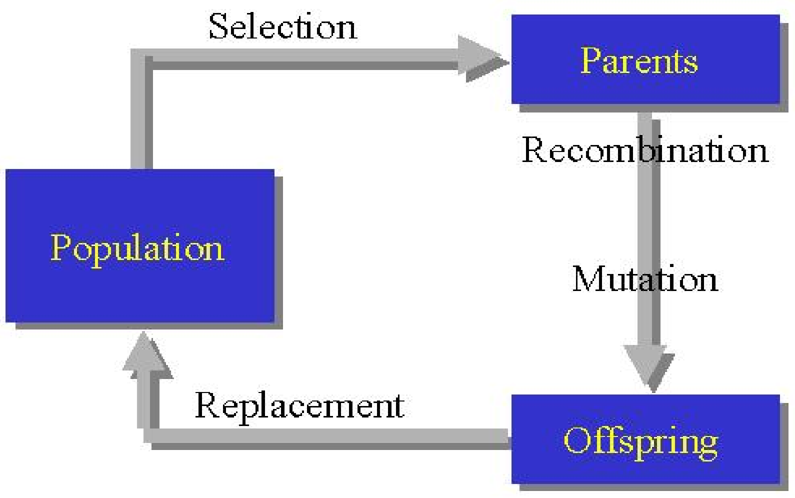


Figure 3.1: Flow chart of the EC schema.

(the chromosomes) is referred to as the *genotype* of that organism. The physical realization of the organism is referred to as the *phenotype* of that organism. The same terms (genotype and phenotype) are also used in EC to refer to the encoding used to represent the solutions of the problem faced and the solutions themselves. However, in some kinds of EC–based algorithm genotype and phenotype are identical. In biology, the *fitness* of an organism is the ability of that individual to live long enough to produce genetically viable offspring. In EC the fitness of an evolving individual is generally the ability of the phenotype of that individual to meet the specifications set by that specific EC system.

More specifically, EC–based algorithms find solutions for a given problem by generating a *population* of *individuals*, i.e. a set of tentative solutions. Then the "goodness" of each individual as solution of the problem is evaluated by means of a fitness function containing all the knowledge on the problem to be solved. Finally, a new population is generated by selecting individuals in the current population and modifying them by variation *operators*, in order to generate new and better, if possible, individuals. This process is repeated until one or more conditions are not satisfied. Formally, given a population p_t at time t, an evolutionary algorithm applies variation operators v on the population. Variations are applied according to a selection method s, where individuals compete to be selected according to their fitness. A population p_{t+1} at time $t + 1$ is found by:

$$p_{t+1} = v(s(p_t))$$

The variation operators provide new solutions modifying the existing ones. In evolutionary algorithms, these operators usually consist of *recombination* and *mutation* operators. In the EC jargon the individuals chosen by the selection method and given as input to the variation operators are called *parents*, whereas the individuals generated by these operators as output are called *offspring.* The use of a population of solutions, together with the variation operators and a selection mechanism, implementing competition among individuals, provides an effective strategy to explore the solution space of the problem to be solved. Note that only those solutions which are reachable by the operators, i.e. those obtainable by iteratively applying the operators to the individuals of the initial population (p_0), could be visited during the evolutionary process or *run.*

Due to the generality of the computational schema just described, many EC–based algorithms have been proposed since the first appearance of this natural phenomena inspired computational scheme and probably many others will be proposed in the future. Given a problem to be solved an evolutionary algorithm can be devised for that problem defining the following four elements:

- A *solution encoding*, i.e. a data structure, have to be defined in order to encode the solutions of the problem. The definition of such a structure should make easy the definition of variation operators.
- Variation *operators* that produce offspring by modifying the selected parent individuals. The operators implements the concept of inheritance through stochastic variation and are strictly related to the data structure employed to represent the solutions of the problem at hand;
- A *fitness function* that evaluates each individual and assigns to it a score, or *fitness value.*
- A *Selection method* that implements a choice mechanism that favors individuals with higher fitness.

In the following these key elements of any EC–based algorithm are described.

3.2.1 Solution Encoding

According to the Encyclopedia Britannica a code can be defined as a system of symbols and rules used to define a transformation establishing correspondences between the elements belonging to two different domains. In nature, long sequences of DNA (Deoxyribonucleic acid) molecules encode the instructions specifying the biological development of all cellular forms of life. DNA sequences contain all the information characterizing the individuals and its ability to survive in the environment. For example, in humans this can range from the hair color to the ability to roll the tongue and to any hereditary diseases. This encoding allows the inheritance of the traits from both the parents to the offspring. Moreover, mutations occurred in the DNA code, are immediately transmitted to the encoded organism. This mechanism gives the nature the chance of exploring new possibilities of life, fitter to the environment.

In the EC field the difference between the solution (phenotype in the EC jargon) and its encoding is not always clear because in some cases they are identical. However, in many cases the role of the solution encoding is that of defining simpler and often more effective operator.
As mentioned in the introduction in the EC field the different representations employed for encoding solutions and the corresponding operators are used in order to categorize the four main branches in which the EC field can be divided: genetic algorithms use a bit-string and two-parent crossover, evolutionary strategies use a real-valued vector and Gaussian mutation, evolutionary programming employs a finite-state machine and mutation operators, and genetic programming uses a computer program or executable structure and two-parent crossover. These classifications represent common or initial implementations, but many implementations use components from different branches and make the classifications less accurate.

Besides the four main branches distinguishable by their different encoding, many specific encoding have been developed in order to solve particular problems. Among the others, grouping problems[1] and graph generation problems have induced the development of specific encoding. As regards grouping problems, Falkenauer [Fal98] proposed a new encoding method specifically devised for this kind of problems. For the graph generation problems, new

[1]In a grouping problem a set of objects has to be partitioned in subsets according to some constraints. An example of grouping problem is the bin packing problem: it consists in placing n objects, each with a weight $\omega_i > 0$, in the minimum number of bins such that the total weight of the objects in each bin does not exceed the bin's capacity. All the bins have the same capacity c. This type of problems belongs to the class of the NP-hard problem.

methods involving specific representation have been proposed in the fields of molecular design [GLW98] and electrical circuit design [CAH$^+$02], using a direct encoding of the evolving graph. Different specific encoding schemes have been also used for evolving artificial neural networks [Yao99]. Moreover, in chapter 5 a new EC–based method able to evolve graphs of variable size is presented.

3.2.2 Operators

The role of the variation (*genetic*) operators from a search standpoint is that of exploring new areas of the solution space searching for new and better, if possible, solutions. The search is usually performed by randomly modifying[2] the individual(s) given in input. Operators are usually characterized by a quantity p which indicate the probability according to which an operator is applied to one ore more individuals. Hence, there is a probability $(1 - p)$ that such individuals remain unchanged. Note that the higher p, the higher the exploration of the search space.

Genetic operators can be grouped in two main classes, *recombination* and *mutation*. The first class groups those operators that take as input two or more individuals and give as output the same, or even lower, number of individuals. The effect of recombination operators is that of swapping or combining parts of the individuals given in input. The effects of this kind of operators strongly depend on the similarity degree of the individuals it is applied on [Sha01]. In fact, more similar are the input individuals lesser is the degree of diversity between input and output individuals.
As regards mutation operators, instead, they take in input a single individual and yield as output a new individual obtained randomly modifying the input one. The effect of this operator is that of a random exploration of the neighbor of the input individual.
These two classes of operators are described in the following.

Recombination

In the EC field, recombination operators have received extensive exploration since the presentation of the pioneer work of John Holland in the 1970s on genetic algorithms [Hol92]. The main idea behind this kind of operators was that of taking sub-parts from individuals that supply different sub-solutions and combining them in order to improve the quality of the solutions.

[2]Note that also eurhystic operators can be defined. This kind of operators do not modify the individual randomly, but use eurhystic knowledge in order to generate better solutions.

The idea of information exchange among individuals in a population by means of recombination operator, in a broad sense, can be regarded as modeling sex. In biology, there are four main explanations proposed for the importance of the role played by sexual reproduction in the process of evolution [Hol00]:

- Provides long (random) jumps in the space of possibilities, thus providing a way off of local maxima.
- Repairs mutational damage by sequestering deleterious mutations in some offspring while leaving other offspring free of them.
- Provides persistent variation that enables organisms to escape adaptive targeting by viruses, bacteria, and parasites.
- Recombines building blocks, hence allowing the discovery of new better solutions made by the rearranged blocks.

The importance and the effectiveness of the recombination operator, is founded on the theoretical foundation of GA's formulated by Holland [Hol92], based on the concept of *schemata*, which in the case of GA's are sub-structures of bit strings. This analysis suggests that selection increasingly focuses the search on subsets of the search space with estimated above-average fitness, whose solutions share one or more of this sub-structure, i.e. schemata. This analysis forms the basis for the fundamental theorem of Genetic Algorithms, namely, the *Schema Theorem* [Hol92]: "Short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations of a genetic algorithm"[3]. A consequence of this theorem is another key idea of the theoretical foundations of the EC: these algorithms explore the search space by short, low-order, high-fit schemata which, subsequently, are recombined to form even more highly fit higher-order schemata. This statement is well known as the *Building Block Hypothesis* [Gol89, Hol92]. The ability to produce fitter and fitter partial solutions by combining building blocks is believed to be a primary source of the search power of any EC–based algorithm. Thus, crossover has been considered to be the primary search operator that distinguishes EC-scheme based algorithms from most other search algorithms.

[3]The concept of schemata introduced by Holland has been extended to others EC-based algorithm, e.g. genetic programming. Furthermore, these extensions have induced the demonstration of the schemata theory also for other classes of EC–based algorithms, e.g. genetic programming [PM03a, PM03b].

Mutation

In biology, mutations are permanent, sometimes transmissible changes to the genetic material (DNA or RNA) of a cell. Mutations can be caused by copying errors in the genetic material during the cell division, by exposure to radiation, chemicals, or also by viruses. Mutations are considered the driving force of evolution, where less favorable (or deleterious) mutations are removed from the gene pool by natural selection, while more favorable (or beneficial) ones tend to accumulate.

In the EC field, the role of the mutation is that of generating new solutions, modifying the genetic material of single individuals. In this way, mutation helps to maintain the genetic diversity in the new population of individuals. A classic and simple example of a mutation operator is that of the GA's: each bit of the bit-string, representing an individual, is changed according to a probability p_m. Generally this probability is set to $1/L$ where L is the length of the string. This value ensure that, on the average, only one bit is mutated in a single string.

3.2.3 Selection Methods

Selection methods are based on stochastic mechanisms that allow one to select in a probabilistic manner the individuals to be chosen as parents for producing offspring in the next population. The role of selection is that of exploiting the information acquired so far in order to find better solutions. Such exploitation takes place favoring individuals having better fitness. Nevertheless, individuals having lower fitness are not completely excluded from the choice process.

Selection methods are characterized by some quantities that essentially measure the degree of exploitation of the information given by those individuals that have a good fitness. If $P(t+1)$ is the population generated from the population $P(t)$ by a given selection method σ, some quantities can be defined:

Selection intensity

If $\overline{f}_{P(t+1)}$ if the expected average fitness value of $P(t+1)$ and $\overline{f}_{P(t)}$ is the expected average fitness value of $P(t)$ the selection intensity is given by

$$S_\sigma = \frac{\overline{f}_{P(t+1)}}{\overline{f}_{P(t)}}$$

Selection variance

The selection variance is the expected variance of the fitness distribution of the population $P(t+1)$.

Loss of diversity
Proportion of individuals of the population $P(t)$ that have not been selected at all by σ and then are not present in $P(t+1)$.

From a search perspective, the selection intensity measures the degree of exploitation of the information available (the fitness values of the individuals in the current population). The loss of diversity, instead, measures the loss of information due to the selection process. These quantities are directly proportional: the higher the selection intensity, the higher the loss of diversity. Note that if too high values are used for such quantities, the system tends to perform like a greedy heuristic. On the contrary, if too small values are used the system behavior is similar to that of random search algorithms. In the following the most used selection methods are described.

Fitness Proportional or Roulette Wheel

Let N be the number of individuals belonging to a population P and $\{f_0, f_1, \ldots, f_{N-1}\}$ be the set of their fitness values. The fitness proportional selection method gives to each individual i in P the following probability of being selected:

$$p_i = \frac{f_i}{\sum\limits_{i=1}^{N-1} f_i}$$

A possible implementation of the method consists in mapping the individuals to N contiguous segments $\{s_0, s_1, \ldots, s_{N-1}\}$ of a line, such that the length of the segment s_i is proportional to f_i. Each time an individual has to be selected, a random number in the interval $[0, \sum_{i=0}^{N-1} s_i]$ is generated, the i–th individual whose corresponding segment s_i spans the number is chosen. This technique is analogous to a roulette wheel (whence its name), where each slice

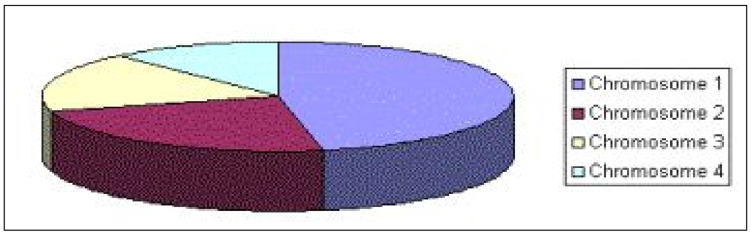


Figure 3.2: Roulette wheel selection.

is assigned to a different individual and the size of the slice is proportional to the fitness of the corresponding individual. In this metaphor the ball is represented by the random number (see figure 3.2).

This selection method, although widely used in many EC–based algorithms, has been criticized because of the importance given to the fitness values differences [BT96]: if differences between "good" and "bad" individuals in a population is high, it is possible that only the first ones will be selected in many copies, decreasing the population diversity. In this situation both the selection intensity and the loss of diversity tend to increase. On the contrary, when in the late phases of a run the individuals in the population tend to have similar fitness values, the selection intensity of this mechanism decreases dramatically. The main drawback of this selection method is that its selection parameters, e.g. selection intensity, strongly depends on the relative fitness values of the individuals to be selected and cannot be tuned from the outside.

Ranking selection

Ranking selection [GB89] was proposed to overcome the scaling problems of the proportional fitness selection (stagnation in the case where the selective pressure is too small or premature convergence where selection has caused the search to narrow down too quickly). It is based on the fitness order, into which individuals are sorted. The selection probability is then assigned to individuals as a function of their rank in the population. Mainly, linear ranking and exponential ranking are used: Let N be the number of individuals to be selected, i the position of an individual in the ordered population (worst fit individual has $i = 1$, whereas the fittest one has $i = N$) and s_p the selective pressure: according to the linear ranking, the probability of the i–th individual to be selected is given by the following formula:

$$p(i) = 2.0 - s_p + 2.0 \cdot (s_p - 1) \cdot (i - 1)/(N - 1)$$

Note that linear ranking allows one to vary the selective pressure in the range $[1.0, 2.0]$.

A non linear formula can be also used to compute the probability to select the i–th individual:

$$p(i) = \frac{c^{(N-i)}}{\sum\limits_{j=1}^{N} c^{(N-j)}}$$

The sum $\sum_{j=1}^{N} c^{N-j}$ normalizes the probabilities so as to ensure that:

$$\sum_{j=1}^{N} p_i = 1$$

According to this formula the probabilities of the ranked individuals are exponentially weighted. The base of the exponent c $(0 < c < 1)$ is the parameter of the method. The closer c to 1 the lower the "exponentially" of the selection method.

The main drawback of the ranking–based selection mechanisms is that it exaggerates the differences between closely clustered fitness values, so that the slightly better solutions can be chosen much more frequently than the slightly worse ones [Whi89].

Tournament Selection

In the tournament selection, a number T of individuals is randomly chosen from the population, and the best individual in this group is selected as parent. This process is repeated for as many times as the number of individuals to be selected.

Among the many selection mechanisms proposed since the beginning of the evolutionary computation [BT96], the tournament is increasingly being used because it is simple and efficient for both non-parallel and parallel architectures. Furthermore, it allows to adjust its selective pressure so as to adapt its performance for different domains. This adjustment can be done simply modifying the the number T of individuals involved in the tournament. In fact, its selective pressure is increased (decreased) by simply increasing (decreasing) the value of T. In figure 3.3 the relations between T and the selective pressure, the loss of diversity and the selection variance are shown. Note that these quantities depend only on the Tournament size T, but are independent of the population size N.

Truncation Selection

Differently from the selection methods modeling natural selection described above, truncation selection is an artificial selection method. It is used by breeders for large populations/mass selection [Bul80].

In truncation selection, individuals are sorted according to their fitness. The individuals to be selected as parents are randomly chosen among the first Tr best individuals. The value of Tr indicates the proportion of the population to be selected as parents and usually takes values in the range

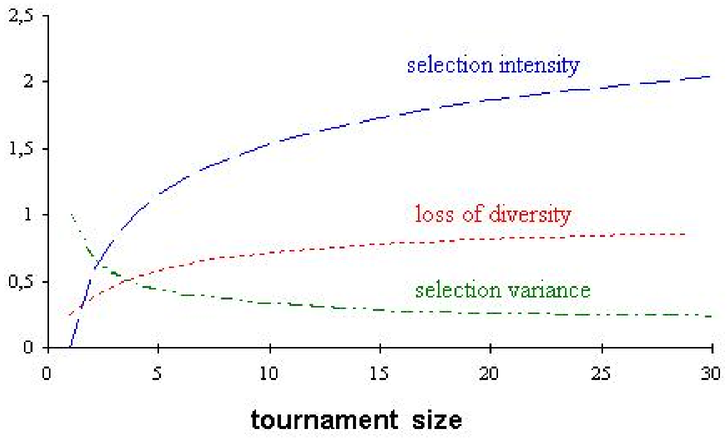


Figure 3.3: Relation between the tournament size T and properties of selection performed.

$[0.10, 0.50]$. Individuals below the truncation threshold do not produce offspring. Clearly, truncation selection allows one to easily control the loss of diversity which is equal to $(1.0 - Tr)$. On the contrary the relation between the selective pressure and Tr is much more complex.

Comparison of Selection Schemes

In [MSV95] an analysis of truncation selection on the ONEMAX function[4] can be found. In [BT96] this analysis is extended to tournament and linear ranking selection as well. For this function the number of generations needed to reach convergence with a simple genetic algorithm is proportional to $\sqrt{n}$ and inversely proportional to the selection intensity (the population should be large enough to converge to the optimum and the initial population should be generated at random). This would suggest a high selection intensity as best selection scheme. However, a high selection intensity leads to premature convergence and thus a poor quality of the solutions. From the analysis done in [BT96] appeared that the three selection methods behave similar assuming similar selection intensity. In figure 3.4 the relation between selection intensity and the corresponding parameter of the selection methods (selective pressure, truncation threshold and tournament size) is shown.

[4]This function takes a bitstring as input and gives as output the number of ones contained in input. As a consequence the optimal string is that containing all ones and the value of the ONEMAX function on it is n, where n is the number of bits in the input strings.

Note that with tournament selection only discrete values can be assigned and linear ranking selection allows only a smaller range for the selection intensity

The three selection methods have also been compared by considering the following parameters: loss of diversity (figure 3.5) and selection variance on the selection intensity. From the plot of figure 3.5, it can be seen that truncation selection leads to a much higher loss of diversity for the same selection intensity compared to ranking and tournament selection. This behavior is consequence of the fact that truncation selection is more likely to replace less fit individuals with fitter offspring, because all individuals below a certain fitness threshold cannot be selected. As regards ranking and tournament selection, they seem to behave similarly.

As concerns the differences in selection variance, for the same selection intensity truncation leads to a much smaller selection variance than ranking or tournament, which instead behaves similarly.In [BT96] it was also proved, that the fitness distribution for the population generated by ranking and tournament selection, respectively setting $s_p = 2$ and $T = 2$, is identical.

3.2.4 Fitness

As mentioned above fitness function measures the "goodness" of an individual as solution of the problem at hand. In practice, such a function contains the whole available knowledge on the problem. This information is exploited by the selection mechanism in order to choose the individuals that will undergo genetic manipulation for producing the new population. Looking at this fact from a search space standpoint, the fitness function provides information for locating regions containing good solutions. The fitness function can be seen as the "driver" of the search performed by any EC–based technique, that guides the search toward those regions of the solution space that seems to be very promising, according to the built-in knowledge of the fitness function. As a consequence the definition of an appropriate fitness function for the problem faced, is a key issue in the designing phase of any EC–based algorithm.

Another way to look at the fitness functions is in terms of *fitness landscape*, which shows the fitness value for each possible chromosome. Using a landscape metaphor in order to develop insight about the workings of a complex system originates from the work of Wright on genetics [Wri32]. A simple definition of fitness landscape in EC is the following: a fitness landscape is a plot where the points in the horizontal direction represent different individual genotypes in a search space and the points in the vertical direction represent the fitness value of each individual [LP02]. If genotypes can be visualized in two dimensions, the plot can be seen as a 3-D map, which may contains peaks and valley (see figure 3.6). From a fitness landscape perspective the

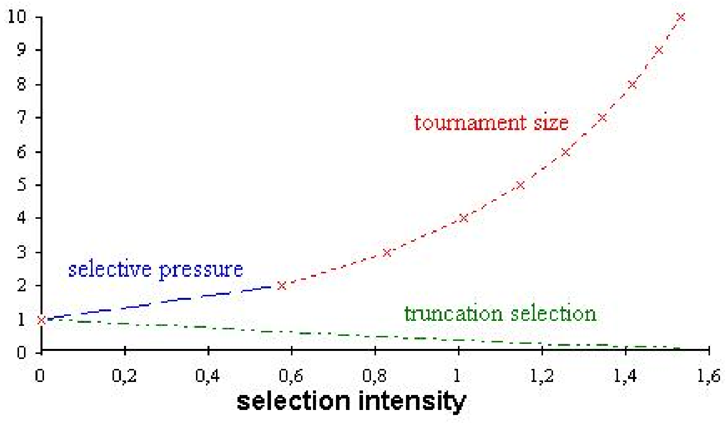


Figure 3.4: (a) Dependence of selection parameters on selection intensity for ranking selection (the blue line), tournament (red) and truncation (green).

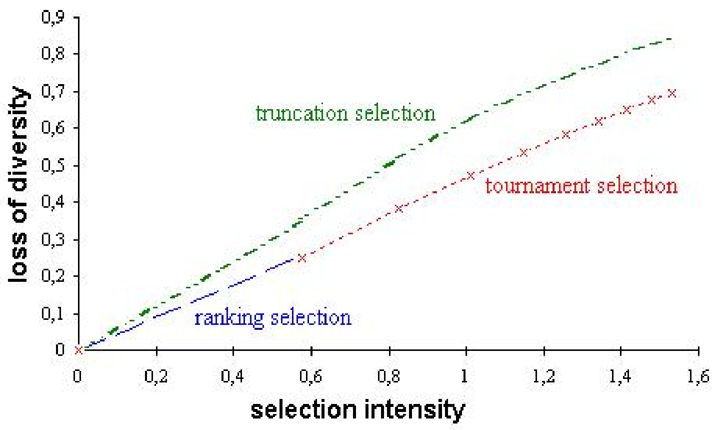


Figure 3.5: Dependence of loss of diversity on selection intensity.

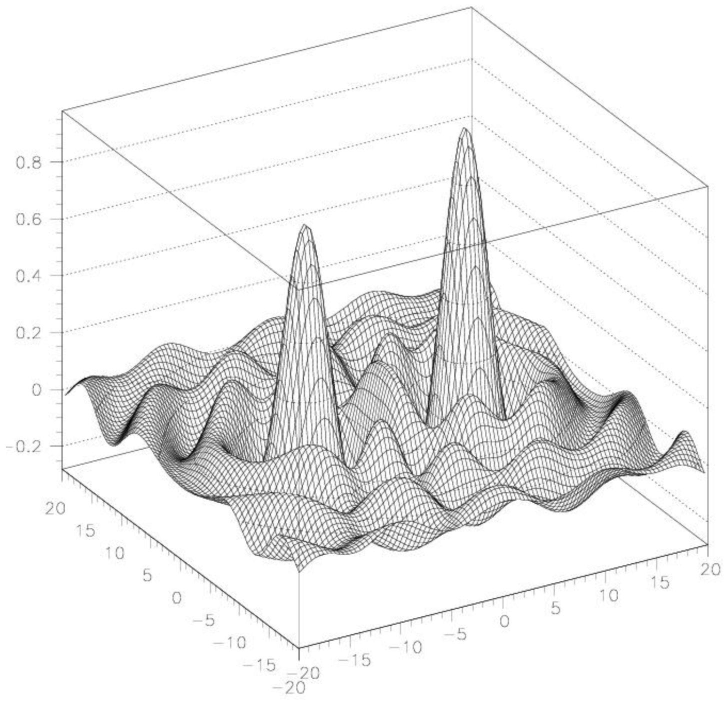


Figure 3.6: En example of fitness landscape. In which the phenotype is represented by a couple of integers $[-20, 20]$.

task of finding the best solution for a problem is equivalent to finding the highest peak (for maximization problem) and any technique used as problem solver can be seen as short-sighted explorer searching for it.

From the fitness landscape perspective, EC–based techniques can be imagined as operating via population of explorers, initially scattered at random across the fitness landscape. Explorers, i.e. individuals, that have found a relatively high fitness points are rewarded with a high probability of surviving and reproducing.

Fitness Landscape and Problem Difficulty

The fitness landscape metaphor can be helpful to understand the difficulty of a problem, i.e. the average amount of search needed to find the optimal solution for that problem. For example, imagine a very smooth and regular landscape with a single hill top. This is a typical fitness landscape of an easy problem, the most of the search strategies are able to find the top of the hill in a straightforward manner.

The opposite is true for a very rugged landscape, with many hills which are lower than the best one. The search of the global optimum in this case is a very hard task for every search strategy, as it is easy to be trapped on one of the many sub-optimal peaks present in the landscape, usually called *local optima*, while the highest peak is called *global optimum*.

Another kind of hard fitness landscape are those that contain totally flat areas, usually called *plateaus*. These might be situated at high or low fitness values, but in both cases there is no gradient and any search technique has no guidance as to where he should go next. At the extreme of this situation there is the *needle in a haystack* landscape: all the points in the landscape have the same fitness value, except one, the global optimum, that has a higher fitness than the others. In this extreme case, every search strategy can only act like a random searcher, moving at random through the landscape and hoping to fall by hazard on the global optimum (the needle).

Fitness Landscape Topology

Using the fitness landscape metaphor presents some difficulties: if the different genotypes of the search space can be represented in two dimensions, then it is possible to visualize the landscape by using a 3-D image like that of figure 3.6. But, usually, this is not the case in practice, since in the most of the problems of practical interest the number of variables involved is more than two and then the landscape cannot be visualized. Moreover, problems can be multi objective, i.e. the fitness can have several component, each represented on a different vertical axis. As a consequence, fitness landscapes are very difficult to be visualized in practice.

Furthermore, in order to understand the features of a fitness landscape, it is not enough to know which points are in the search space and what their fitness values are, but it is also necessary to know which point is a neighbor of which other point. It is clear that the definition of a *neighbor relationship* is a crucial step in the construction of a fitness landscape, since its main characteristics, like peaks, valleys, plateaus, etc. strongly depend on the neighborhood relationship and deeply change if this relationship is modified. This relationship is defined in terms of the genetic operators used to explore the search space: two individuals are assumed to be neighbors if it is possible to obtain the second one by the application of one step of a genetic operator to the first one. While the neighbor relationship can be easily defined for unary genetic operators like mutation, it is much more difficult define this relationship for binary or multi-parent operators like crossover. This difficult may prevent one from being able to plot the landscape even for one or two dimensional search spaces.

Note that given an operator O, a distance d can also be defined in terms of O: the distance between the points a and b in the search space $d(a, b)$ is equal to the number of applications of O needed to transform a in b. So doing, if S is the entire solution space, the neighborhood N of a point $x \in S$ can be defined as the set of points y of S reachable by one application of the operator O:

$$N(x) = \{y \in S : y = O(x)\}$$

The notion of neighborhood is essential for the definition of local optima: those points $x \in S$ for which (for maximization problems):

$$f(x) - f(y) \geq 0, \quad \forall y \in N(x)$$

A point x^* is a global optimum if it is the absolute maximum in the whole S:

$$f(x^*) - f(y) \geq 0, \quad \forall y \in S$$

Notice that global optima are well defined by the fitness function and do not need the definition of any structure on S. Other features of a landscape such as basins, barriers, or neutrality can be defined likewise in [Sta02]

An alternative way for representing the fitness landscape, that can overcome some of these problems, is by means of a graph, where the nodes represent the individual genotypes in the search space, the labels of the nodes are the fitness values of the corresponding genotype and the arcs define the neighborhood relationship on the basis of a unary genetic operator.

Examples of Fitness Function

Fitness evaluation may be as simple as computing a mathematical function or as complex as running an elaborate simulation. In order to communicate this variety in an effective way, in the following two practical examples of fitness evaluation are given.

Example 1
Suppose that one is planning to evolve a good airplane wing. To do this he designs a way of encoding many of the important aspects of standard airplane wings as parameters. This list of parameters will be the genotype of the airplane wing. With a specific list of parameters, one could take this list and create a physical realization of the wing (e.g., out of plastic) or a virtual realization of the wing (e.g., a CAD model). This realization is the phenotype of the evolving structure. Imagine that one is trying to discover a wing design that has a minimal drag to lift coefficient. If a virtual wind-tunnel is available, the process for determining the fitness of the chromosome c in the current population could work as follows:

- From the chromosome c create the phenotype it encodes (a CAD model of the wing).

- Insert this virtual wing into the virtual wind-tunnel and measure the drag α and lift β created by this wing under the air conditions of interest.

- Assign the fitness to be $f_c = \frac{\alpha}{\beta}$

In this example, the fitness defined allows one to evolve wings that have a good drag to lift coefficient, but other parameters, e.g. the wing weight are not optimized, since they have not been inserted in the fitness function.

airplane wings under evolution are said to be *encoded* by the bitstrings that represent them.

Example 2
See now another example of fitness: suppose that one is unaware of the Taylor series expansion for $f(x) = e^x$ and he would like an approximation to e^x by means of a functions that contains the basic arithmetic operators $(+, -, *, /)$ plus the variable x and constants. Such an approximation function can be obtained using a GP algorithm. The fitness of a chromosome c that encodes the function $\mathcal{F}_c$ can be defined as follows:

- Pick N random values for x inside the range of interest (e.g., $[0, 1000]$);

- For value x_j $(1 \leq j \geq N)$, define the error for this x value as: $E_j = (e^{x_j} - \mathcal{F}_c(x_j))^2$;

- Compute the fitness of c as, $f_c = \frac{1}{N}\sum_{j=1}^{N} E_j$

This particular experiment has been performed and several first terms of the Taylor's series were successfully evolved [Koz92]. In this example, the fitness of each chromosome is an approximation to the chromosome's real "fitness", i.e. that measured on all the points in the given range, and that the fitness of chromosome c will be different if tested more than once. Moreover, N is a parameter of the fitness calculation: higher it is and greater is the accuracy of the measured fitness[5]. This situation occurs anytime that the calculation of the exact fitness for an individual is too expensive.

3.3 Genetic Algorithms

Genetic algorithms (GAs) are one of the best known class of EC-based algorithms. They have been invented by Holland in the middle of the 1970s

[5]Obviously, the price to pay for a greater accuracy is a greater computational cost for the fitness evaluation.

[Hol92]. GAs have been successfully applied to a wide variety of real-world problems, such as combinatorial optimization ones or learning tasks. In this class of algorithms, individuals are coded as fixed length strings of characters.

The first step in the design of a GA is the definition of the *alphabet* Σ, the set of the n allowed characters that can be used to construct the strings and the strings length L. The search space, i.e. the set of all possible individuals, is composed by n^L different strings. In the most of the case, that will be referred to as *canonical GA* in the following, is the one in which the alphabet consists of just the two symbols 0 and 1 ($\Sigma = \{0, 1\}$). In this case the size of the search space equals to 2^L.

The GA process starts with the generation of an initial population, consisting of a set of individuals randomly chosen in the search space. The size p of the population is usually fixed once for all at the beginning of the run. Note that usually $p << n^L$. After that the initial population has been created, the process enters in a loop, where the following steps are performed at each iteration (iterations of the GA process are usually called *generations*):

1. The fitness of each string is evaluated.

2. Selection is performed: one individual is selected from the current population and inserted in the *mating population*[6]. This selection process is repeated p times. At the end of this process the mating population contains p individuals.

3. Variation is performed: recombination and mutation are applied to the individuals in the mating population in order to create the *next* population.

The process just described ends when a *termination criterion* is fulfilled. The most commonly used termination criteria are:

- at least one individual in the current population has a satisfactory fitness value;

- a prefixed maximum number of generation has been reached.

As mentioned in section 3.2.3 several selection methods have been developed. These methods are independent from the way in which the solutions are encoded, any selection method can be used for performing the selection step (point 2). As regards the specific operators of the GAs they are described in the following.

[6]Note that the selected individuals are not removed from the population under selection.

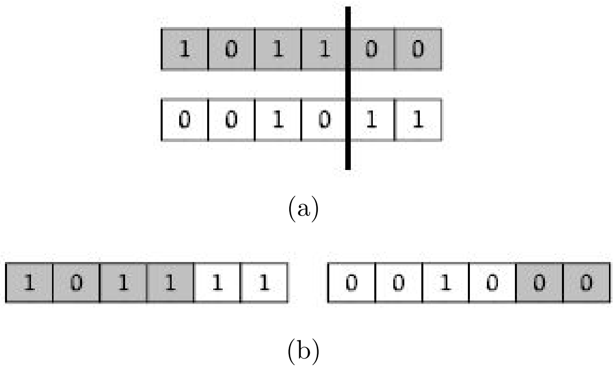


Figure 3.7: (a) Two GA individuals selected for crossover. The crossover point is equal to 4. (b) On the left the offspring obtained by combining the crossover fragment of the first parent with the remainder of the second one is shown, while the offspring obtained combining the crossover fragment of the second parent and the remainder of the first one is shown on the left.

3.3.1 Recombination

The recombination operator used in the GAs is usually called *crossover*. It is applied to couple of strings with a probability p_c and produce two offspring, each containing genetic material from each of their parents.

Many crossover algorithms have been proposed for GAs in the literature. The most used one is called *one point crossover* and its operation is illustrated in figure 3.7. This kind of crossover is performed first randomly choosing a number t in the range $[1, L-1]$. The number t identifies one of the $L-1$ interstitial locations lying between the positions of a string of length L. In the example of figure 3.7(a) the chosen interstitial location is the fourth one. This location becomes the *crossover point*. Then each parent is split at this crossover point into a *crossover fragment* and a *remainder*. The crossover fragment of the first individual of figure 3.7(a) is the string `1011` while the remainder is `00`. The crossover fragment of the first individual is then combined with the remainder of the second one while the crossover fragment of the second individual is combined with the remainder of the first one.

3.3.2 Mutation

The mutation operator modifies a sub-string of an individual, with a certain probability p_m. Also in the case of mutation many algorithms have been de-

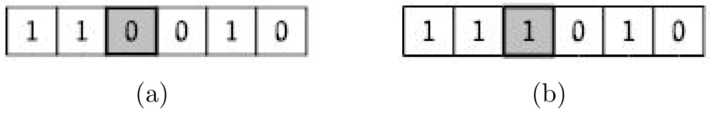


Figure 3.8: (a) The string to be mutated. In this case only the third position has been changed. (b) The individual obtained after mutation.

veloped. The most commonly used is called *point mutation* and its operation is illustrated in figure 3.8. Each position in the input string is chosen with a probability p_m and the character contained in that position is then replaced with another randomly chosen character. The optimal mutation rate for GAs have been extensively studied for the canonical GAs and has been found that the in the most of the cases the optimal rate is $1/L$ [Och05]. This mutation rate causes, on average, the change of just one bit for each individual.

3.4 Genetic Programming

GAs are capable of solving many problems and are simple enough to allow solid theoretical studies. Nevertheless, the fixed-length string representation of the solutions that characterizes them is difficult, unnatural and constraining for a wide set of applications. In these cases the most natural representation for a solution is a hierarchical computer program rather than a fixed-length character string. For example, fixed-length strings do not readily support the hierarchical organization of tasks into subtasks typical of computer programs, they do not provide any convenient way for incorporating iteration and recursion and so on. But above all, GAs representation schemes do not have any dynamic variability: the initial selection of string length limits in advance the number of internal states of the system and limits what the system can find out.

This lack of representation power is overcome by Genetic Programming [Koz92], which operates with very general hierarchical computer programs. Even though every programming language (e.g. Pascal, Fortran, C, etc.) is virtually capable of expressing and executing general computer programs, Koza chose the LISP (LISt Processing) language for encoding GP individuals. The reasons of this choice can be summarized in the following:

- Both instructions and data have the same form in LISP, so that it is possible and convenient to treat a computer program as data in the genetic population.

- This common form of programs and data is a parse tree and this fact allows one to decompose in a simple way a structure in substructures (subtrees) to be manipulated by the genetic operators.
- LISP facilitates the programming of structures whose size and shape change dynamically and the handling of hierarchical structures.
- Many programming tools are commercially available for LISP.

In other words GP, as originally defined by Koza, considers individuals as LISP-like tree structures. These structures are perfectly capable of capturing all the fundamental properties and features of modern programming languages. The GP using this representation will be called *tree-based* from now on. The tree-based representation of genomes, although is the oldest and most commonly used, is not the only one that has been employed in GP. In particular, in the last few years a growing attention has been given by researchers to linear genomes [BB01, RCO98] .

In synthesis, GP based algorithms breeds programs for solving problems by executing the following steps:

1. Generate an initial population of computer programs.
2. Iteratively perform the following steps until the termination criterion has been fulfilled:
   (a) Execute each program in the population and assign it a fitness value according to how well it solves the problem at hand.
   (b) Create a new population by applying the following operations:
      i. Probabilistically select a set of computer programs to be reproduced, on the basis of their fitness (selection step).
      ii. Copy some of the selected individuals, without modifying them into the new population (reproduction).
      iii. Create new computer programs by genetically recombining randomly chosen parts of two selected individuals (crossover)
      iv. Create new computer programs by substituting randomly chosen parts of some selected individuals with new randomly generated ones (mutation).
3. The best computer program appeared in any generation is designated as the result of the GP process and given in output as solution of the problem whose knowledge has been inserted in the fitness function used to evaluate the individual.

In the following sections each specific features of this process is detailed.

3.4.1 Representation of GP Individuals

The set of all the possible structures that GP–based system can generate is the set of all the possible trees that can be built recursively from a set of function symbols $\mathcal{F} = \{f_1, f_2 \ldots, f_n\}$ (used to label internal tree nodes) and a set of terminal symbols $\mathcal{T} = \{t_1, t_2 \ldots, t_m\}$ (used to label tree leaves). Each function in the set $\mathcal{F}$ takes a fixed number of arguments, specifying its *arity.* Functions may include arithmetic operations (+,-,*,etc.) mathematical functions (such as sin, cos, log, exp, etc.) conditional operations (e.g. `IF-THEN-ELSE`) and other domain-specific functions that may be defined. Each terminal symbol is typically either a variable or a constant, defined on the problem domain. In figure 3.9 an example of GP individual is shown.

Closure and Sufficiency of Function and Terminal Sets

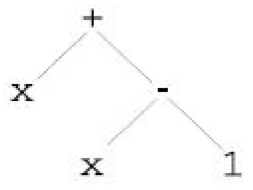
The function and terminal sets should be chosen so as to verify the requirements of *closure* and *sufficiency.*

The closure property requires that each of the functions chosen is able to accept as its arguments any value and data type that may possibly be returned by any function and any terminal. In other words each function should be well defined for any combination of arguments that it may encounter.

The reason why this property must be verified depends on the fact that programs must be executed by an interpreter in order to assign them a fitness value. The failure of the execution of a program due to a wrong argument for one or more of the functions in it, makes impossible its evaluation. The function and terminal sets of the example proposed in figure 3.9 satisfy this property.

The following ones, for instance, don't satisfy this property: $\mathcal{F} = \{+, -, /\}$ and $\mathcal{T} = \{x, 1, 0\}$. In fact, each evaluation on an expression containing the

Figure 3.9: A legal tree built by using the sets $\mathcal{F} = \{+, -\}$ and $\mathcal{T} = \{x, 1\}$. The tree encodes the expression $(+x(-x1))$
.



operation of division by zero would be lead to an unpredictable behavior. Usually, in any GP implementation the division operation is suitably modified in order to satisfy the closure property. The satisfaction of this property in real-life applications is not always so easy, because the use of many different data types could be necessary.

Another common example of this situation is a mix of boolean and numeric functions: function sets could be composed by boolean functions (AND, OR, etc.), arithmetic functions (+,−,∗,etc.), comparison functions (>,<,=, etc.) and conditional ones (IF THEN ELSE), etc. and one might want to evolve expression such us:

$$\text{IF } ((x > 10 * y)\text{AND}(y > 0)) \text{ THEN } z + y \text{ ELSE } z * x.$$

In such cases, the introduction of typed functions into the GP genome can help to force the closure property to be verified. For example, a particular GP version, called *strongly typed* GP (STGP), in which each node carries its type as well as the types it can call, thus forcing functions calling it to cast the argument into the appropriate type, has been introduced [Mon93a]. The use of typing makes much more efficient the initialization and variation phases with respect to a standard GP, as type checking allows the system to generate only individuals that respect the closure property. Furthermore, the effect of the type checking is that of reducing the search space which is likely to improve the search.

The sufficiency property requires that the set of terminals and that of the functions is capable of expressing a solution to the problem. For instance, the function and terminal sets of the example of figure 3.9 satisfy the sufficiency property if the problem at the hand is an arithmetic one. It does not verify this property if, for instance, if the problem faced is a logic one. For many domains, the requirements for sufficiency in the function and terminal sets are not clear and the definition of appropriate sets depends on the experience of the GP designer and his knowledge of the problem.

3.4.2 Initialization of a GP Population

The initialization of a GP population is the first step of the evolution process. It consists in the creation of the program structures that will be later evolved. The most common initialization methods in the tree-based GP are the *grow* method, the *full* method and the *ramped half-and-half* method [Koz92]. This methods will be explained in the following paragraphs, where the set of the function symbols composing the trees will be denotated by $\mathcal{F}$, the set of the terminal symbols by $\mathcal{T}$ and the maximum depth allowed for a tree by d.

Grow initialization

When the grow method is employed, each tree of the initial population is built using the following algorithm:

- a random symbol is chosen with uniform probability from $\mathcal{F}$ to be the root of the tree;

- let n be the arity of the function symbol chosen. Then n nodes are selected with uniform probability from the set $\mathcal{F} \bigcup \mathcal{T}$ to be its sons;

- for each function symbol between these n nodes, the method is recursively applied, i.e. its sons are selected from the set $\mathcal{F} \bigcup \mathcal{T}$, unless this symbol has a depth equal to $d - 1$. In the latter case, its sons are selected from $\mathcal{T}$.

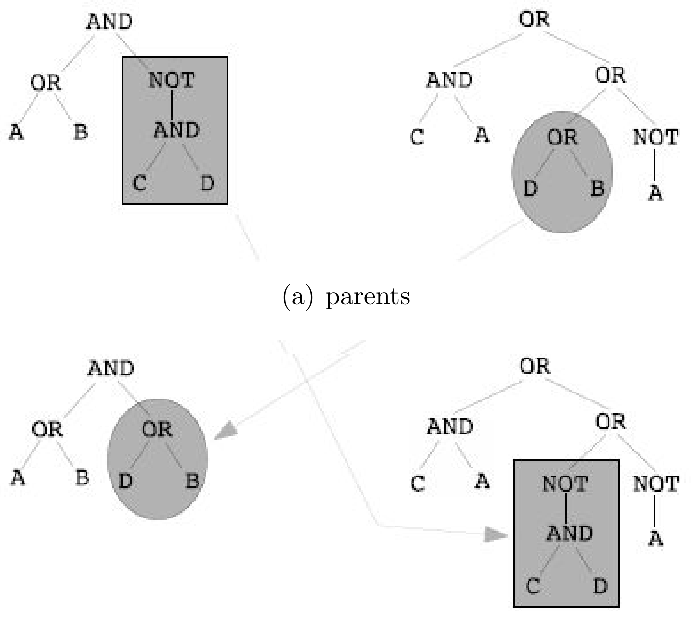
Summarizing, the root is chosen with uniform probability from $\mathcal{F}$, so that no tree composed by a single node is initially created. Nodes with depths between 1 and $d - 1$ are chosen with uniform probability from $\mathcal{F} \bigcup \mathcal{T}$, but once a branch contains a terminal node, that branch has ended even if the maximum depth d has not been reached. Finally, nodes at depth d are chosen from $\mathcal{T}$. Since the incidence of choosing terminals from $\mathcal{F} \bigcup \mathcal{T}$ is random throughout the initialization, trees initialized by using the grow method are likely to have irregular shape, i.e. to contain branches of various different lengths.

Full initialization

Instead of choosing nodes from $\mathcal{F} \bigcup \mathcal{T}$, the full method chooses only the function symbols until a node is at the maximum depth. Then it chooses only terminals. The result is that every branch of the tree goes to the full maximum depth.

Ramped half-and-half initialization

As first noted by Koza [Koz92], population initialized with the above two methods my be composed by trees that are too similar between them. The ramped half-and-half technique has been developed in order to enhance population diversity. Let d the maximum depth parameter, the population is divided equally among individuals to be initialized with tree having depths equal to 1, 2, ..., $d-1$, d. For each depth group, half of the trees are initialized with the full technique and half with the grow one.



(a) parents

(b) offspring

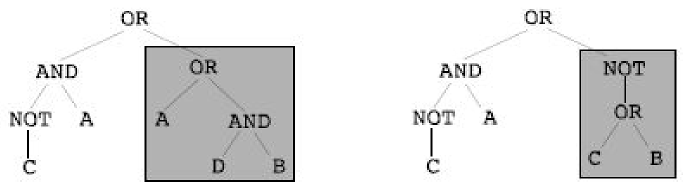
Figure 3.10: An example of crossover application. Crossover fragments are included into gray forms.

3.4.3 Recombination

The recombination operator of the GP, commonly called *crossover* so as in the GAs, creates variation in the population by producing new offspring that consist of parts taken from each parent.

Given two parents I_1 and I_2, GP crossover starts by independently choosing one random point in each parent (it will be called the crossover point for that parent). Usually the crossover point is chosen in such a way that the node lying underneath this point has the probability equal to 0.9 of being an internal node and a probability of 0.1 of being any node (internal or terminal). The crossover fragment for a particular parent is the subtree rooted at the node lying underneath the crossover point. The first offspring is produced by deleting the crossover fragment from I_1 and inserting the crossover fragment of I_2 at the crossover point of I_1. The second offspring is produced in a symmetric manner. In figure 3.10 an example of crossover application is shown.

The swapping of entire subtrees, regardless of the crossover points chosen, yields syntactically legal programs only if the closure property of the func-



(a) parent (b) offspring

Figure 3.11: An example of mutation application.

tions is satisfied. It is important to remark that in cases where a terminal and/or the root of one parent are located at the crossover point, generated offspring could have considerable depths. This may be one possible cause of the phenomenon of *bloat*, i.e. progressive growth of the code size of individuals in the population. For this reason, many variants of the standard GP crossover have been proposed in the literature. The most common ones assign different probabilities of being chosen as crossover point to the various parents'nodes, depending on the depth level they are situated. In particular, it is very common to assign low probability of being chosen as crossover points to the root and the leaves, in order to limit the influence of degenerative phenomena like that described above.

3.4.4 Mutation

The standard GP mutation, called *subtree mutation*, starts by choosing a point at random, with uniform probability distribution, within the input individual. This point is called the *mutation point*. Then, the subtree laying below the mutation point is removed and a new randomly generated subtree is inserted at that point. This operation so as in the case of standard crossover is controlled by a parameter that specifies the maximum depth allowed and limits the size of the newly created subtree that is to be inserted. Nevertheless, the depth of the generated offspring can be considerably larger than the one of the parent.

Also for the mutation many variants of the standard one have been developed. The most commonly used are the ones aimed at limiting the probability of choosing the root and/or the leaves of the parent as mutation points. A particular example of mutation variant is the *point mutation* [PL97], in which, concurrently, each node in the tree has a given probability to be changed with a random one of the same arity.

GPTerminals	a0 a1 d0 d1 d2 d3
GPNonterminals	and(2) or(2) not(1) if(3)

Table 3.1: The GP symbols for the 6-multiplexer problem. The numbers in the parentheses indicate the arity of the corresponding nonterminals symbols.

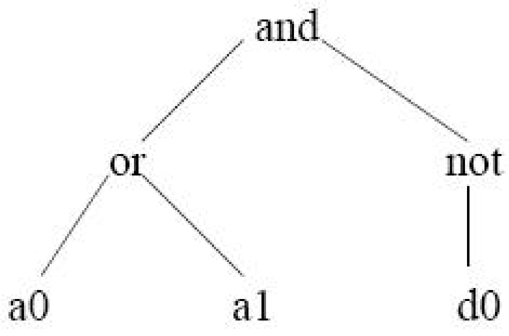


Figure 3.12: A GP tree for the symbols of the table above.

3.5 Grammatically–based Genetic Programming

As mentioned in section 3.4.1 the requirement of closure is hard to be satisfied in many cases, especially when several data types could be necessary. Several approaches have been proposed within the GP framework in order to overcome this drawbacks [Mon93b, Ros94]. Among the others, that proposed by Whigham [Whi96], who introduced the use of context free grammar (CFG), is one of the most effective. The class of context-free grammars is one of the four main classes distinguished by Chomsky [Cho56]. Of these grammars, the class of the context-free ones is the most popular, as the grammars belonging to this class are simple and applicable to many problems. For example, most modern programming languages have been defined using these grammars.

The use a context-free grammar allows the user to bias the initial GP structures, and automatically ensure that typing and syntax are maintained by manipulating the explicit derivation tree built using the grammar. This system will be referred to as *context-free grammar genetic programming*, or CFG-GP.

A context free grammar is a four-tuple (N, Σ, $\mathcal{P}$, S), where N is the non-

terminal[7] alphabet, Σ is the terminals alphabet, $\mathcal{P}$ is the set of productions and S is the designated start symbol. The productions are of the form
$x \rightarrow y$
where $x \in N$ and $y \in \{\Sigma \bigcup N\}^*$.
Productions of the form
$x \rightarrow y$
$x \rightarrow z$
may be expressed using the disjunctive symbol $|$, as
$x \rightarrow y|z$
The string separated by the symbol '|' on the right side of the rules will be denoted in the following as *clause*.

A derivation step represents the application of a production form of $\mathcal{P}$ to some nonterminal $A \in N$. In the following the symbol $\Rightarrow$ is used for representing the derivation step. For example, given the nonterminal A, the derivation step from A by applying the production rule $A \rightarrow \theta$ is represented as:

$$\alpha A \beta \overset{A \rightarrow \theta}{\Rightarrow} \alpha \theta \beta$$

where $\alpha, \beta, \theta \in \{N \bigcup \Sigma\}^*$ and $A \in N$.

Moreover, a derivation *rooted* in A, where $A \in N$, is defined as

$$A \overset{*}{\Rightarrow} \alpha \text{ where } \alpha \in \{N \cup \Sigma\}^*$$

Here $\overset{*}{\Rightarrow}$ represents zero or more derivation steps. Note that a series of derivation steps may be represented as a tree such as in figure 3.13.

3.5.1 An Example of Application of CFG-GP

In this section the 6-multiplexer[8] is used as example to describe the differences between traditional GP and CFG-GP. A full description of the 6-multiplexer can be found in [Koz92].

In table 3.1 the sets of symbols used by the GP for representing the 6-multiplexer problem is shown. In figure 3.12 a GP tree available by using this table is also shown. While in table 3.2 one of the many possible grammars that allows the creation of the same functional structures is shown. In figure

[7]Grammars traditionally use the definitions *terminal* and *nonterminal* to represent the atomic tokens and symbols to be replaced, respectively. GP have used these terms for distinguishing functions with a number of arguments greater than 0 and 0-arity functions or atomic values. In order to ensure to avoid confusion when discussing GP constructs the words GPterminals and GPnonterminals will be used.

[8]The 6-multiplexer is a simple boolean problem where two address lines are used to select between four possible data lines.

Symbols

$N = \{B, T\}$
$\Sigma = \{a0, a1, a2, d0, d1, d2, d3\}$

Production rules

1)	S	$\rightarrow$	B
2)	B	$\rightarrow$	$and\ B\ B \mid or\ B\ B \mid not\ B \mid if\ B\ B\ B \mid T$
3)	T	$\rightarrow$	$a0 \mid a1 \mid a2 \mid d0 \mid d1 \mid d2 \mid d3$

Table 3.2: A context free grammar for the 6-multiplexer problem.

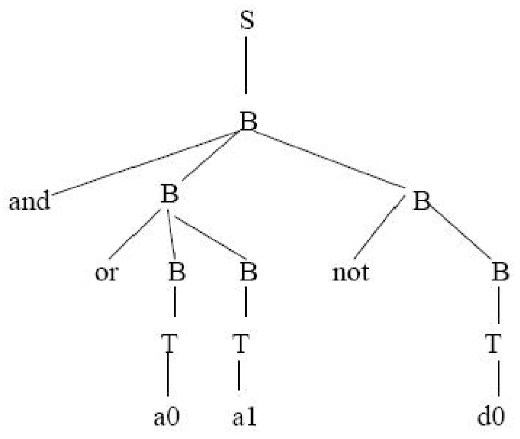


Figure 3.13: A tree created from the grammar of table 3.2.

3.13 a derivation tree generated by using this grammar is shown. Note that both the trees represent the function $and(or(a0, a1), not(d0))$.

In the following the creation of the initial population and the operators of the CFG-GP are described.

Creating the initial population

In the GP approach the individuals of the initial population can be created by using one of the method described in section 3.4.2. Generally the most used among the methods mentioned above is the ramped half-and-half.

In the CFG-GP, instead, the individuals in the initial population, encoded as derivation tree, are generated by starting with root node containing the symbol S, then a clause on the right side of the corresponding production rule is randomly chosen. After, a number of sons equal to the number of symbols in the chosen clause is assigned to the root node. Each son contains a different one of these symbols. The same process is recursively repeated for each nonterminal node in the tree (see figure 3.13). Note that the derivation tree so built represents the genotype of the individual. As regards the phenotype, it is represented by the string built by visiting the tree in depth first order and copying into this string the terminal symbols contained in the leaves.

In order to control the growth of the tree generated by a given grammar, to each clause on the right hand of a production rule is a associated a value in the range $[0.0, 1.0]$. This value indicates the probability that the corresponding clause is returned when that production rule is activated returns. These values are usually normalized. Specifically, the growth of trees is limited by given a much more high probability to those clauses containing only terminal symbols. For example, if one takes into account the rule

$$B \quad \rightarrow \quad \textit{and } B\ B \ \mid\ \textit{or } B\ B \ \mid\ \textit{not } B \ \mid\ \textit{if } B\ B\ B \ \mid\ T$$

a possible choice of the probabilities for the right side clauses could be the following:

$$0.10 \ \mid\ 0.10 \ \mid\ 0.10 \ \mid\ 0.10 \ \mid\ 0.60$$

Crossover

All the terminals in a derivation tree have at least one nonterminal above them in the program tree, so without loss of generality crossover points can be constrained to be located only on nonterminals nodes. The crossover operation maintains legal programs of the language defined by the grammar by ensuring that the same nonterminals are selected at each crossover point. Given two derivation trees ρ_1 and ρ_2 the crossover algorithm is:

1. Randomly choose a nonterminal $A \in \rho_1$;
2. Randomly select $A \in \rho_2$.
3. Swap the subtrees below these nonterminals;

Note that if the symbol A does not exist in ρ_2 the trees are left unchanged.

Mutation

The mutation operator is applied to the input tree by randomly choosing a nonterminal as the site for mutation. Then the subtree under the chosen nonterminal is deleted and a new subtree is generated according to the grammar productions. The chosen nonterminal is used as starting point.

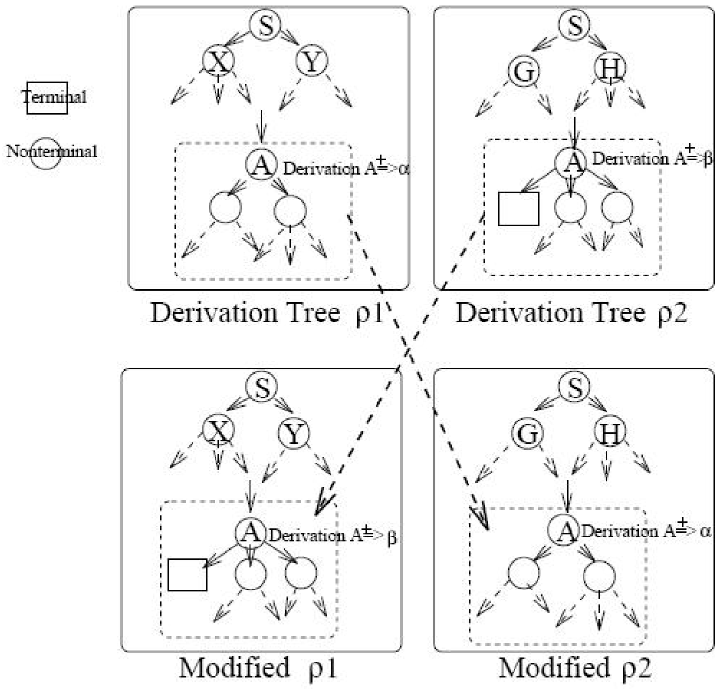


Figure 3.14: An example of CFG-GP crossover

3.6 Applications of EC–based Algorithms

EC–based algorithms are usually used when the problem to be solved is a NP–hard one and the knowledge available does not allow to develop a solving strategy that is specific for the problem, or when the strategies developed so far give unsatisfactory solutions. In these cases, the powerful heuristic provided by the EC paradigm, using the limited knowledge available on the problem may be able to find satisfactory solutions. In the following some applications that use EC–based algorithms are listed:

Combinatorial problems

There exists an extensive range of problems which can be formulated as obtaining the values for a vector of variables subject to some restrictions. The elements of this vector are denominated *decision-variables*, and their nature determines a classification of this kind of problems. Specifically, if decision-variables are required to be discrete (i.e. the solution is a set or sequence of integer numbers or any other discrete type), the problem is said to be *combinatorial.* The process of finding optimal solutions (maximizing or minimizing an objective function) for such a problem is called combinatorial optimization. These problems have been traditionally approached using exact techniques such as Branch and Bound [LW66]. These techniques ensure that the optimal solution is found but, unfortunately, their application is seriously limited as they cannot cope with the combinatorial explosion of the solutions, i.e. the exponential growth of the number of solutions as the number of decision-variables involved increases.

This kind of problems have been addressed quite successfully with GAs. A very common example of this kind of problems is the time-tabling of exams of classes in Universities, etc. At the Department of Artificial Intelligence, University of Edinburgh, time-tabling the MSc exams is now done using a GA–based system. Also the Job-Shop Scheduling Problem [9] is a very difficult NP-hard problem which, so far, seems best addressed by branch and bound search techniques. GAs have been successfully applied also to this problem [NY91].

Neural networks optimization

As mentioned in section 2.1.2 Artificial Neural Networks (ANNs) represent an AI important paradigm dealing with massively parallel information processing proposed as biological models for the human brain. Whenever an ANN is to be used it must first be designed. At present, any ANN design drags along an unstructured, heuristic and arbitrary path in order to reach “the better” structure and connectivity to be trained. Only the training methods are truly applied, but every ANN type seems to need a different own training mechanism. Usually the training mechanism is a some kind of hillclimbing, that is very closely related to (and so, dependent on) the problem being solved, the ANN type and/or the pattern set for it. The final result is a vast landscape of different multiparameter tuning procedures to be carried out for any individual problem and with no warranties for optimum results.

In order to overcome this lack of methodology EC-based methods have been used to design Artificial Neural Networks [Yao99]. Essentially EC-based algorithms have been used at two different levels: optimization of the connection weights or of the architecture. In the former case

[9] An instance of the job-shop scheduling problem consists of a set of n jobs and m machines. Each job consists of a sequence of n activities so there are $n * m$ activities in total. Each activity has a duration and requires a single machine for its entire duration. The activities within a single job all require a different machine. An activity must be scheduled before every activity following it in its job. Two activities cannot be scheduled at the same time if they both require the same machine. The objective is to find a schedule that minimizes the overall completion time of all the activities.

the architecture of the ANN is provided by the user and EC is used just to optimize the connection weights. In the latter case, instead, EC is employed to find the best configuration of the ANN, especially for minimizing its size, while the connection weights are adjusted by using specific algorithms, e.g. back-propagation, in order to minimize the error.

Computer Aided Design (CAD)

The term Computer-aided design (CAD) refers to the use of a wide range of computer-based tools that assist engineers, architects and other design professionals in their design activities. CAD is used to design and develop products, these can be goods used by end consumers or intermediate goods used in other products. CAD is also extensively used in the design of tools and machinery used in the manufacture of components.

In order to optimize the products designed by means of CAD tools, EC-based algorithms have been integrated in this software tools. An example of this use of EC in the CAD designing is represented by EnGENEous a CAD tool developed by General Electric [Gol66]. It is a hybrid system, combining numerical optimization tools and genetic algorithms. This tool has been used to improve the performance in the design Boeing 777's jet engines.

Chapter 4

An Evolutionary Framework for Classification Problems

In this chapter one of the main contributions of this thesis is presented. We have defined a general EC–based framework for generating prototypes in classification problems, which is largely independent of both the classification scheme and the technique used for representing the prototypes. Moreover, our approach does not require any specific knowledge on the classification problem to be solved and it is able to automatically find the actual number of prototypes needed to represent the patterns belonging to different classes. This ability derives from three key assumptions: (i) the encoding of *all* the prototypes searched in a *single* individual; (ii) each individual contains a *variable* number of prototypes; (ii) prototypes within an individual are not a priori labeled. It is useful to remark that the use of such a framework can be very helpful for the development of a recognition system because, independently of the specific classification scheme used, the definition of the prototypes strongly affect the performance of the whole system.

In section 4.1 the previous and related work for classification problems in the EC field is briefly illustrated. While in section 4.2 the framework devised is detailed. In section 4.3 the CFG-GP version of the proposed general framework is presented. Finally, in section 4.4 another version that uses real-valued vectors is presented.

4.1 Related and Previous Work

Thanks to their ability to solve hard problems characterized by complex and high dimensional search spaces, EC-based algorithms have been successfully used to solve classification problems. Specifically, GAs have been applied to evolve sets of rules for more than two decades. These rules predict the class of a pattern specifying some values of the pattern attributes. This methodology forms a machine learning paradigm called *learning classifier systems* (LCS) [LSW00]. In this approach each individual encodes one or more rules of the form IF-THEN: the rule antecedent (the IF part) contains a combination of conditions on some attribute values, while the rule consequent (the THEN part) expresses the class predicted by the rule. GAs for rule discovery can be divided into two main classes, called Michigan and Pittsburgh, based on how rules are encoded by individuals In the Michigan [GN95, GS93] approach each individual encodes a *single* prediction rule, whereas in the Pittsburgh approach [PGP97, Jan93, DSGJ93] each individual encodes a whole *set* of prediction rules.

The Pittsburgh approach was originally devised to solve single–class problems and then only the antecedent part of a rule is encoded. Patterns that match one or more rules are classified as a positive examples of the class, whereas patterns that do not match any rule are classified as a negative examples. In order to tackle multi–class problems, they have been extended by introducing multiple populations so that each population is dedicated to learn rules for a specific class. Note that it is also possible that a pattern is matched by more rules belonging to different classes or by any rules of any class. Unfortunately, this problem has not been addressed in many of the systems based on the Pittsburgh approach.
Also the Michigan approach was developed to cope with problems for a single–class only, as well. In case of multi–class problems, these algorithms are run once for each class, where each run evolves a set of rules for a specified class. So as in the Pittsburgh approach, it is possible that an instance is matched by several rules, each predicting a different class, or it is also possible that an instance is matched by none of any rule predicting any class. In order to overcome this problem affecting both the approaches, some hybrid Michigan/Pittsburgh methods have also been proposed [Hek97].

As mentioned above, genetic algorithms have widely been used to cope with classification problems, but only recently some attempts have been done solve such problems using GP [ABL02, RDC00, DDT02, KPMA00]. In [RDC00], GP has been used to evolve equations involving simple arithmetic operators and feature variables, for hyper-spectral image classification. In [ABL02], GP has also been employed for image classification problems, adding exponential functions, conditional functions and constants to the simple arithmetic operators. In [CLH02] GP has been used to generate discriminant functions which carry out arithmetic functions with fuzzy attributes for a classification problem. In [MVFN01] GP has been used to evolve populations of fuzzy rule sets, whereas a simple evolutionary algorithm was employed to evolve the membership function definitions. The populations involved are allowed to co-evolve in such a way that both rule sets and membership functions can adapt each other. In [KPMA00], an interesting method which considers a c-class problem as a set of c two-class problems has been introduced. When the expression for a particular class is searched, that class is considered as target, while the other ones are merged and treated as a single undesired class. In this way, c expressions can be obtained performing c runs. These expressions can then be used concurrently for discriminating the c different classes of the problem at hand. In all the above quoted approaches, the number c of classes to be dealt with is used to divide the data set at hand in exactly c parts. Thus, these approaches do not take into account the existence of subclasses within one or more of the classes in the analyzed data set. In the EC field, classification tasks have been faced by considering them as multimodal optimization problems: a single prototype is seen as one of the several solutions to be searched for [HGD94, HG96]. In this kind of approach the individuals representing the candidate prototypes are concurrently evolved in the same population and *niching* methods [GR87, DG89] are used to form and maintain the searched solutions. These methods consider a good solution (i.e. prototype) and its neighborhood in the solution space as a *niche*. The aim of these algorithms is that of creating groups of individuals, usually called *subpopulation*, each one occupying a different niche. A subpopulation has the task of finding out the best solution within the niche it occupies. Among niching methods, fitness sharing are the best known and the most widely used [Hor97]. In this approach, a distance measure is defined and the individuals having distances among them lower than a given value (niche radius) are considered as belonging to the same niche. Such individuals are forced to share a given amount of fitness. In practice, the sharing is obtained by reducing the fitness of all the individuals within the niche according to their number. The effective use of the fitness sharing in real world problems, however, is severely limited because it requires a high degree of knowledge about the fitness function landscape, while for the most of such problems such knowledge is hard to achieve.

4.2 An EC–based Framework for Classification problems

In this section an EC–based framework for generating prototypes is presented. This framework is general and has been defined without specifying the form of the prototypes to be used. As a consequence, it can be applied

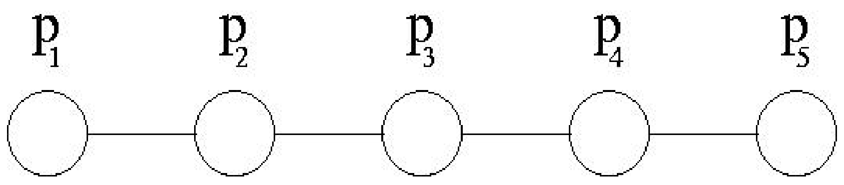


Figure 4.1: The framework solution encoding. A solution is encoded as a list of prototypes. The circles represent the data structures used to encode the prototype.

to any kind of classifier and to any classification problem, once a way for encoding the prototypes has been specified. Given this encoding, each individual consists of a variable number of prototypes representing the whole set of prototypes to be used for the problem at hand (see figure 4.1). The knowledge about the specific problem to be solved is limited to the choice of a training set $\mathcal{D}_{\text{tr}}$ of labeled patterns that must be provided to the framework.

The assumption that an individual is made up by the whole set of prototypes offers several advantages. The most relevant one is that it is possible to find the actual number of prototypes by using a simple and well defined fitness function. In fact, in our approach the fitness of an individual coincides with the recognition rate obtained by using the whole set of prototypes making up that individual. Moreover, in order to find the minimum number of prototypes, a term that favors individuals having a lower number of prototypes, can be added to the fitness function. Discovering the actual number of prototypes needed to represent the patterns for a given class is a crucial issue in the design of any classifier. In most of the real applications, in fact, the variability exhibited even by the pattern belonging to the same class cannot be managed by defining a single prototype. In figure 4.2 a simple example in which this situation may occurs is given. In this example two different ways of writing the character “A” are taken into account. So, here it is clear the difficulty of representing these different patterns by the same prototype.

In figure 4.3 the outline of the algorithm of the framework proposed is shown. In the initial population each of the P individuals is generated first randomly choosing a number n in the range $[N_{\text{min}}, N_{\text{max}}]$ with a uniform probability distribution. While N_{min} is usually chosen equal to the number of classes of the problem at hand, the value of N_{max} has to be chosen by the user and represent a parameter of the algorithm. The number n represent the number of prototypes that makes up the i–th individual. Then n prototypes are generated by using the adopted encoding scheme. This generation can be done randomly or not. Usually, the majority of prototype are randomly

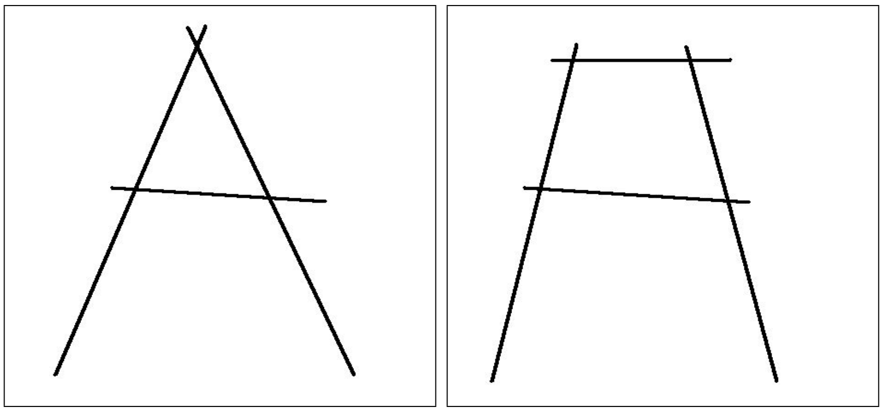


Figure 4.2: Two different ways of writing the letter "A".

generated, while a certain number of them are generated in such a way to match same of the patterns in the training set.

After that the initial population has been created, the prototypes of each individual are labeled by using the dynamic labeling mechanism defined. Then the fitness of each individual is evaluated. Note that also the fitness evaluation is defined within the framework. Afterwards a new population is generated. The first, in terms of their fitness values, $\mathcal{E}$ individuals, according to an elitist strategy[1], are just copied in the new population. Then any selection mechanism can be used in order to choose $(P - \mathcal{E}/2)$ couples of individuals. To every selected couple, the recombination operator defined within the framework is applied according to a chosen probability factor p_c. After the mutation operator specific of the encoding used to represent the prototypes is applied according to a probability factor p_m. The new population so built replaces the old one and the process is repeated until a termination criterion is fulfilled. In the following the specific parts of the framework proposed are detailed.

4.2.1 Solution Encoding

As mentioned above, an individual contains the whole set of the prototypes. This set is encoded in the individual as a list of the data structure that encodes the prototypes. Obviously, in order to automatically find the right

[1]An elitist strategy ensures that the fittest members of the population are passed on to the new one without being altered by genetic operators. Using elitism ensures that the best individuals so far evolved can never been lost from one generation to the next. From a search space perspective, such a mechanism preserves the information contained in the best area so far located.

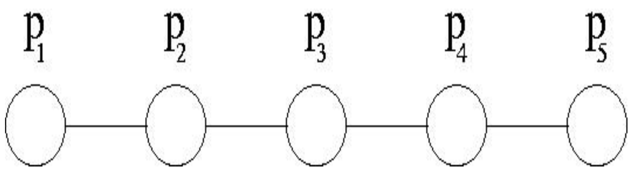
```
begin
  create a population of $\mathcal{P}$ individuals;
  evaluate the fitness of each individual;
  for $i = 0$ to $N_G$ do
    copy the best $\mathcal{E}$ individuals in the new population;
    for $j = 0$ to $(\mathcal{P} - \mathcal{E})/2$ do
      apply the selection mechanism;
      replicate the selected individuals;
      if flip($p_c$) then
        apply the crossover operator on the selected individuals;
      end if
      if flip($p_m$) then
        perform mutation on the offspring;
      end if
      evaluate the fitness of each individual;
    end for
    replace the old population with the new one;
    update variables for termination;
  end for
end
```

Figure 4.3: The outline of the algorithm of the framework defined.
Note that the function flip(p) returns the value 1 with a probability p and the value 0 with a probability $(1 - p)$. Its role is that of implementing the probability of application of the operators.

number of prototypes, the *length* of the individuals, i.e. the cardinality of the set encoded by an individual, is variable. In figure 4.1 an example of an individual that encodes a set of five prototypes is given.

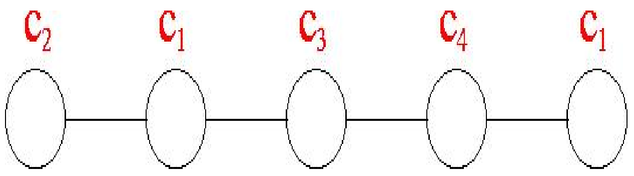
4.2.2 Prototype Labeling

The prototypes making up an individual are not a priori labeled, but their labeling occurs after a matching process: in such a process, each pattern in the training set is assigned to one prototype according to the matching procedure defined for the considered classification problem. In the following the procedure employed to label the prototypes, called *dynamic labeling*, is given:



	P_1	P_2	P_3	P_4	P_5
c_1	3	36	0	0	44
c_2	30	6	0	1	0
c_3	0	0	47	3	0
c_4	0	0	8	68	4

(a)



	P_1	P_2	P_3	P_4	P_5
c_1	3	36	0	0	44
c_2	30	6	0	1	0
c_3	0	0	47	3	0
c_4	0	0	8	68	4

(b)

Figure 4.4: An example of the dynamic labeling of the 5 prototypes of an individual. In (a) the patterns of the training set (belonging to 4 different classes) have been assigned to the prototypes. In (b) each prototype is labeled according to the patterns assigned to it.

1. The assignment of the training set patterns to the prototypes making up the individual is performed. After this step, p_i ($p_i \geq 0$) patterns have been assigned to the i-th prototype. In the following the prototypes for which $p_i > 0$ will be referred to as *valid*. The remaining prototypes ($p_i = 0$) will be ignored.

2. Each valid prototype is labeled with the label most widely represented among the patterns that have been assigned to it.

In figure 4.4 an example of application of this procedure is given. In the example, four classes have been defined and the individual is made up of four prototypes. The first prototype p_1 is labeled with the label of the second class, as it has demonstrated to be a good prototype for that class, while, for the same reason, p_2 has been labeled with the label of the first class. The prototypes p_3, p_4 and p_5 have respectively been labeled with the label of the third, fourth and first class. Note that in this example the individual provides two prototypes for the first class.

Note that the dynamic labeling together with the recombination operator, described in section 4.2.4, allows the framework to automatically find the right number of prototypes for the classification problem faced. Yet, the dynamic labeling of prototypes allows one to relax a strong constraint caused by the a priori labeling of the prototypes (see figure 4.5). In fact, suppose that the analyzed data contains c classes and each individual also contains c a priori labeled prototypes[2], the constraint imposed on the labels of the prototype reduces of a factor (c!) the number of solutions to be considered as good solutions of the problem at hand. This so strong limitation depends on the fact that, given a set of c prototypes in which each prototype is a good one for a different class, of its (c!) possible permutations just one is considered as good solution, while the other ones are considered as bad solutions. This situation occurs because, in these other cases, the position in the list of a given prototype may not coincide coincide with the label preliminarily assigned to it (see figure 4.5(b)). Considering the example given in figure 4.5, if the dynamic labeling is used and the individual is evaluated according to the fitness function described in the next section (without taking into account the term that evaluate the length of an individual) then the value 0.9 is assigned to the individual as fitness value. While, if the prototypes of the individual are a priori labeled, then the value 0.32 is assigned as fitness value if the same fitness function is used.
This constraint become stronger as c increases. For example, in the case in

[2]An obvious way of a priori labeling the prototypes is that of label the first one with the label of the first class, the second one with the label of the second class and so on.

which 20 classes are defined in the data, given a good set of prototypes of cardinality 20, just one solution containing this set has a good fitness, while the other $\approx 2 \cdot 10^{18}$ individuals that contain the same set of prototypes have a fitness value significantly smaller.

4.2.3 Fitness Evaluation

Given a training set $\mathcal{D}_{tr}$ providing the information on the prototypes to be generated, in the framework, the fitness of an individual is defined in a straightforward manner. This straightforwardness is due to the fact that a single individual contains all the prototypes needed to build a classifier for the problem at hand. As a consequence of this fact it is obvious to evaluate the performance of an individual as the recognition rate[3] obtained on the training set by the classifier built up using the prototypes of that individual. Hence such rate is assigned as fitness value to the individual.

Although the defined fitness function allows one to well identify in a straightforward manner those individuals which perform in an optimal way[4] the required task , it does not take into account the number of prototypes provided by the individual evaluated. In fact, such fitness function makes no difference between two individuals that obtain the same recognition rate on the training set but which use a unequal number of prototypes. This lack of information about the number of prototypes in the fitness function may causes the generation of too long individuals, i.e. consisting of too many prototypes. Furthermore, the overestimation of the number of prototypes reduces the effectiveness of the classifier and may also causes overfitting phenomena [5][DHS01]. Thus, in order to build up a more effective classifier, those individuals able to obtain good performances with a smaller number of prototypes are favored increasing their fitness values by c_p/N_p, where N_p is the number of prototypes in an individual[6].

Summarizing, given an individual I that is made up of N_p prototypes and that has obtained the value r_{tr} as recognition rate on the training set $\mathcal{D}_{tr}$, its fitness f_I is equal to:

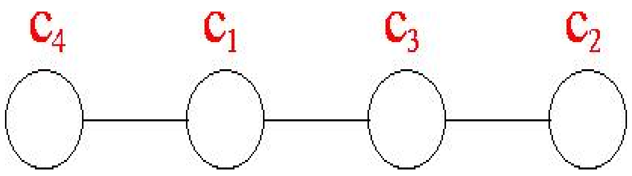
$$f_I = r_{\mathrm{tr}} + \frac{c_p}{N_p}$$

[3]For an exact definition of recognition rate see the Appendix.

[4]Note that the fitness value of an optimal individual, namely one correctly classifying all the training patterns, equals to 1.0 using the fitness function so far defined.

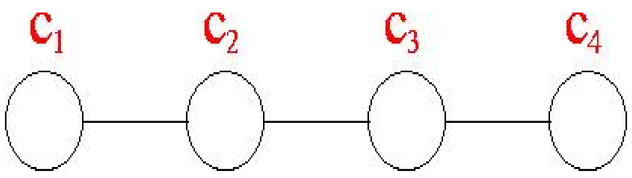
[5]In a learning process, overfitting phenomenas occur when the generalization power of the system, i.e. its ability of obtaining good performances on unknown data decreases significantly.

[6]Given a data set containing N_p clusters, the addition of this term to the fitness function implies that the fitness value of an optimal individual equals to $1.0 + c_p/N_p$.



	P_1	P_2	P_3	P_4
c_1	3	38	0	0
c_2	0	6	0	35
c_3	0	0	47	3
c_4	60	0	0	8

(a)



	P_1	P_2	P_3	P_4
c_1	3	38	0	0
c_2	0	6	0	35
c_3	0	0	47	3
c_4	60	0	0	8

(b)

Figure 4.5: A comparison between the dynamic labeling mechanism used by the framework (a) and that a priori (b). In the example the number of prototypes equals that of the classes defined, i.e. 4. In (b) the prototypes are labeled according to their position within the list making up the individual.

where c_p is a constant and represents a parameter of the algortihm.

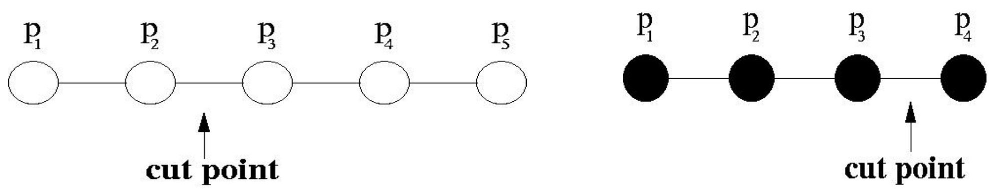
4.2.4 Operators

The recombination operator is applied to two individuals i_1 and i_2 and yields two new individuals by swapping parts of the lists of i_1 and i_2. Assuming that the length of i_1 and i_2 are respectively l_1 and l_2, the recombination is applied in the following way: the first individual i_1 is split in two parts by randomly choosing, with an uniform probability distribution, an integer t_1 in the interval $[1, l_1]$. The obtained lists of prototypes $i_1^{'}$ and $i_1^{''}$ will have length t_1 and $l_1 - t_1$ respectively. Analogously, by randomly choosing an integer t_2 in the interval $[1, l_2]$, two lists $i_2^{'}$ and $i_2^{''}$, respectively of length t_2 and $l_2 - t_2$, are obtained from i_2. At this stage, in order to obtain a new individual, the lists $i_1^{'}$ and $i_2^{''}$ are merged. This operation yields a new individual of length $t_1 + l_2 - t_2$. The same operation is applied to the remaining lists $i_2^{'}$ and $i_1^{''}$ and a new individual of length $t_2 + l_1 - t_1$ is obtained. It is worth noting that this recombination operator allows one to obtain individuals of variable length. Hence, during the evolution process, individuals made of a variable number of prototypes can be evolved.

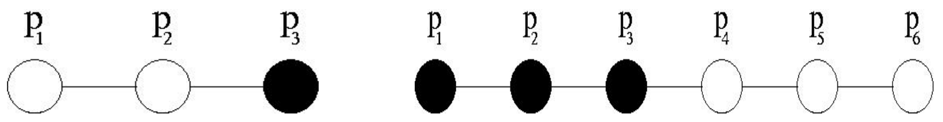
As regards the mutation operator, it has not been defined within the framework, but have to be provided together with the encoding for the prototypes used by the classifier. This choice is a consequence of the fact that in the framework no encoding for the prototypes has been defined.

4.3 A CFG-GP Version of the EC-framework

In this section the first of the applications of the framework described in the previous section is presented. This application is based on the CFG-GP algorithm described in section 3.5. In the developed application the prototypes of the classifier used consist of a set of logical expressions, while the patterns are represented as feature vector. A prototype, i.e. logical expression, may contain a variable number of predicates holding for the patterns belonging to one of the class defined in data analyzed. A predicate establishes a condition on the value of a particular feature. If all the predicates of an expression are satisfied by the values in the feature vector describing a pattern, it is said that the prototypes *matches* the pattern (see figure 4.7). Given the training set $\mathcal{D}_{\mathrm{tr}}$, the set of prototypes for this kind of classifier is generated by using the algorithm shown in figure 4.3. In this case each prototype is encoded as a derivation tree of the expression which it contains. Hence, an individual is a list of derivation trees. The labeling of the prototypes in an individual



(a) Parents



(b) Offspring

Figure 4.6: An example of application of the recombination operator. In (a) the two parents, respectively of length 5 and 4 are shown. While the two offspring obtained, of length 3 and 6, are shown in (b). As regards the cut points t_1 and t_2 they have randomly been chosen respectively equal to 2 and 3.

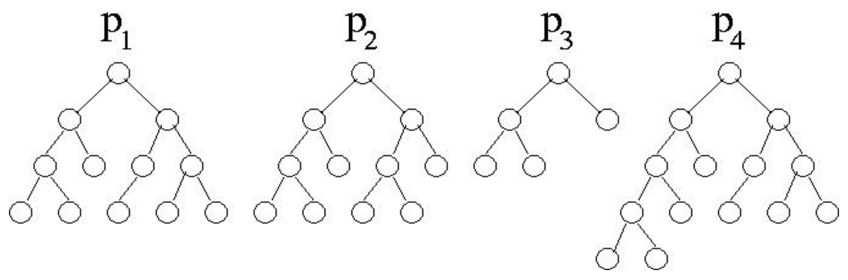
is accomplished by using the dynamic labeling procedure described in the previous section. As regards the operators, the recombination operator defined in the framework is applied as crossover operator in order to modify the length of the individuals. The mutation operator, instead, is independently applied to each of the prototypes. This operator is that defined within the CFG-GP approach (see section 3.5).

Given a pattern to be recognized and a set of labeled expressions, i.e. prototypes, the classifier assigns it to one of the classes defined in the problem in the following way: the pattern is matched against the set of expressions that form its prototypes and is assigned to one of them or rejected (see figure 4.7). If assigned the pattern is recognized as belonging to the same class of the prototype to which it has been assigned.
Different cases may occur:

1. The pattern is matched by just one expression: it is assigned to that expression.

2. The pattern is matched by more than one expression with different number of predicates: it is assigned to the expression with the smallest number of predicates.

3. The pattern is matched by more than one expression with the same



$P_1 = ((x_1 < 0.4) \wedge (x_4 > 0.8)) \vee (x_6 > 1.0)$
$P_2 = (x_3 > 0.1) \vee (x_2 < 0.0)$
$P_3 = (x_1 < 0.0)$
$P_4 = ((x_5 > 0.4) \wedge (x_2 > 0.0)) \vee ((x_3 > 0.1) \wedge (x_7 < 1.0))$

0.3	**1.0**	**0.0**	**0.6**	**1.3**	**0.5**	**1.1**	**5.6**	**2.0**

Figure 4.7: An example of solution encoding that contains 4 prototypes (above) and the prototypes encoded (middle) in the CFG-GP application of the framework. On the bottom an example of feature vector representing a pattern is given. This pattern is assigned to the 4–th prototype as it is the only one that matches it.

number of predicates and different labels: the pattern is rejected.

4. The pattern is matched by no expression: the pattern is rejected.

5. The pattern is matched by more than one expression with equal label: the pattern is assigned to the class the expressions belong to.

The matching between patterns and expressions is computed by means of an automaton which computes Boolean functions. Such an automaton accepts as input an expression and a pattern and returns as output the value true or false depending on the fact that the expression matches or not the pattern.

In table 4.1 the grammar used to generate the expressions that form the prototypes is reported. The root of every tree is the symbol S that, according to the related production rule, can be replaced only by the symbol A. This symbol can be replaced by any recursive combination of logical predicates whose arguments are the occurrences of the elements in the feature vector. Note that the 2–th production rule ensures that a prototype contains at least two predicates. The logical expression encoded in a derivation tree is obtained by visiting the tree in depth first order and copying into a string

Number	**Rule**	**Probability**
1	$S \longrightarrow A$	1.0
2	$A \longrightarrow ABA\|D$	0.2, 0.8
3	$B \longrightarrow \vee\|\wedge$	equiprobable
4	$D \longrightarrow (P > V)\|(P < V)$	equiprobable
5	$P \longrightarrow x_0\|a_1\|\dots\|x_N$	equiprobable
6	$V \longrightarrow +0.XX\| - 0.XX$	equiprobable
7	$X \longrightarrow 0\|1\|2\|3\|4\|5\|6\|7\|8\|9$	equiprobable

Table 4.1: The grammar for randomly generating logical expressions. N is the dimension of the feature space. Nonterminal symbols are denoted by capital letters. On the right the probability for each of the right clauses is reported.

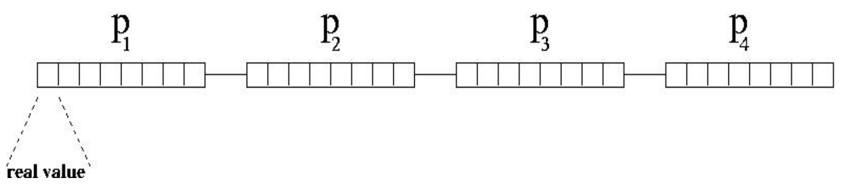


Figure 4.8: An example of individual containing 4 prototypes. Each prototype is real-valued vector of 8 elements.

the symbols contained in the leaves. Since the grammar is non-deterministic, in order to reduce the probability of generating too long expressions (i.e. too deep trees) the action carried out by a production rule is chosen on the basis of fixed probability values (shown in the last column of Table 4.1)

4.4 The LVQ Version of the EC-framework

In this section the second of the applications of the EC-based framework devised is presented. This application is based on a particular class of EC–based algorithms, namely the Breeder Genetic Algorithms (BGA) [MSV93] in which the individuals are encoded as real valued vectors. In our case a single individual consists of variable–length list of real-valued vectors (see figure 4.8).

In the classifier used here both patterns and prototypes are represented by feature vectors. This classifier is based on the widely used approach of Learning Vector Quantization (LVQ) proposed by Kohonen [Koh01]. LVQ classifiers have been applied in a variety of practical problems, including medical image analysis, classification of satellite spectral data, fault detection in technical processes, and language recognition, to name only a few. The LVQ approach offers a method to form a quantized approximation of an input data set $\mathcal{D} \subset R^p$ using a finite number k of *reference vectors* $\omega_i \in R, i = 1, \ldots, k$. These vectors represent the prototypes to be used to represent the patterns belonging to the classes present in the data. Given an incoming pattern $\mathbf{x}$ to be recognized and a set of reference vectors $\{\omega_1, \ldots, \omega_k\}$, the classification is performed by calculating the Euclidean distance between $\mathbf{x}$ and each of the reference vector ω_i. Then the pattern is assigned to the closest reference vector and it is recognized as belonging to the same class of the reference to which it has been assigned. In the LVQ method the reference vectors are algorithmically determined by using a training set of labeled patterns. Note that usually the number of reference vectors for each of the classes have to be provided by the user.

Given a training set $\mathcal{D}_{\text{tr}}$, the set of prototypes for this kind of classifier, i.e. reference vectors, is generated by using the algorithm shown in figure 4.3. In this case each prototype is a real–valued vector which number of elements equals the dimensionality of the data analyzed. As regards the individuals in the initial population, each of the prototypes contained in these individuals is initialized by randomly choosing a pattern from $\mathcal{D}_{\text{tr}}$. The dynamic labeling is used to label the prototypes, while the crossover operator defined in the framework is applied in order to modify the length of the individuals. As regards the mutation operator, it is independently applied to each of the prototypes in the input individual and is used to modify the single prototypes (see figure 4.9).

```
begin
  for $j = 0$ to $N_F$ do
    range = rndreal(0.1 * $\delta_j$)
    if flip($p_m$) then
      $\mathbf{x}[j] = \mathbf{x}[j] \pm$ range (+ or - with equal probability);
    end if
  end for
end
```

Figure 4.9: The mutation operator applied to each of the prototypes, i.e. reference vectors, in an individual. N_F is the number of features of the patterns in the data analyzed. δ_j is the range of the j-th feature computed on the training set, while p_m represents the probability of mutation of each single feature value in the prototype.

Chapter 5

A New EC–based Method for Evolving Graphs

In this chapter a new EC–based approach for generating variable size graphs is presented. The method is based on a problem independent data structure, called *multilist*, which encodes undirected attributed relational graphs. For this data structure two operators have also been defined: a recombination one, called *crossover*, that swaps subgraphs between two graphs. This operator allows the devised approach to generate graphs of variable size. The second operator, called *mutation*, changes the input graph into a new one by modifying both node and link attributes.
In order to preliminarily ascertain the effectiveness of the devised approach a hard non–linear optimization problem, regarding the planning of wireless LANs, has been faced.

The rest of the chapter is organized as follows: Section 5.1 illustrates previous and related work in the field of graph generation; in Section 5.2 a first description of the defined data structure is given; Section 5.3 contains a formal definition of this data structure; Section 5.4 contains descriptions of various type of graphs encoded by multilists, while in Section 5.5 the operators defined within the approach are detailed. Finally, Section 5.6 is devoted to the description of the problem taken into account to test the approach.

5.1 Previous and Related Work

In the last decades, there has been an increasing interest in studying and using graphs in various fields of science and engineering as they are a powerful and versatile data structure, well suited to represent in an effective way

complex and structured information. In fact, they may effectively represent physical networks, such as transportation systems, power systems, and mobile communication infrastructures [GY01, Cas01, Cro03], but have also been used to model less tangible interactions, as might occur in ecosystems, databases or in the control flow of computer programs [GY01].
In fields like pattern recognition and machine vision, the high representational power of graphs makes them very attractive and well-suited to model complex patterns in terms of parts and their relations. Attributes of graph nodes and arcs are often added to incorporate further information, leading to a graph representation form generally denoted as Attributed Relational Graph (ARG) [EF84]. Examples of successful applications include shape analysis and 3-D object recognition [PSZ98, ACS01], character recognition [FGK95], classification of ideograms and symbols in document analysis and technical drawing interpretation [CV00].

In many cases, a prominent problem is that of generating graphs exhibiting some particular properties. The generation of class prototypes in a pattern recognition problem, so as the generation of the optimal set of connections between a given set of nodes, with given constraints, are examples of such problem. Thus, the use of graph representations often requires the definition of effective techniques for generating the graphs representing the desired solutions. To this purpose, two main different approaches can be identified, depending on the nature of the problem: in case of applications in which examples of the population to model are available, the graphs may be generated by exploiting the information included in a training set of examples. In all the other cases the solution is found by defining a function $\mathcal{F}$ able to measure the goodness of tentative solutions in a given space: the graphs representing the solutions are generated by finding all the absolute maxima of the function $\mathcal{F}$. Combinatorial, heuristic and inductive learning approaches have been used, among others, to generate graphs [CFSV02].
Several attempts to generate graphs using evolutionary approaches have also been done. Methods have been proposed in the fields of molecular design [GLW98] and electrical circuit design [CAH$^+$02], using a direct encoding of the evolving graph. It is worth noting that these methods define evolutionary operators tailored for the considered problem. Indirect encoding of graphs in terms of bit strings [LC98] or trees [HG04] has also been used. In the latter approach, for instance, a tree encodes the operations to be applied to a very simple starting graph, in order to transform it into another one arbitrarily complex.

With evolutionary computation, methods for encoding graphs have also been studied within the area of artificial neural networks (ANN) [Yao99]. In fact, because of the difficulties of training an ANN when the error function employed is multimodal or nondifferentiable at all, an evolutionary approach may be adopted so that near optimal solutions can be obtained. With this kind of approach the first choice to made concerns the encoding scheme in order to provide an initial population and variation operators for the evolutionary algorithm. Most of the approaches adopted used a direct encoding and one of the most successful examples of evolving ANN using direct encoding scheme is presented in [CF01], in which an ANN is optimized and used to play checkers. The success of this kind of encoding, although it could in some cases result inefficient, is due, in large measure, to its simplicity and to its convergence properties [Man94, YL97]. However, direct encoding allows one to implement simpler operators, like, for example, the operators employed for the bitstring representation used in GAs. Moreover, I think that this simplicity is a key for the effectiveness of any EC–based algorithm.

5.2 A New Data Structure for Graph Encoding

In this section a data structure specifically devised for graph encoding will be illustrated. Let us consider a graph[1] G of N nodes. Both nodes and arcs in G may have attributes respectively belonging to the sets A_n and A_a. The data structure we have adopted to represent attributed relational graphs has been called *multilist* (ML in the following) since it is based on the list concept and consists of two basic lists. The first one, called *main list*, contains the information on graph node attributes, thus its number of elements is equal to the number N of nodes in G. Each element of the second list is on its turn a list, called *sublist* (see Figure 5.1). One sublist is associated to each node and includes the attributes of the arcs connected to that node. In order to preserve information about the nodes interconnected by each arc, arc attributes are sorted in each sublist in a suitable order. Namely, the i-th sublist contains information on the arcs connecting the i-th node of the graph to the nodes following it in the main list, in the order they appear in such list. If two nodes are not connected, this information is anyway suitably stored in the proper place of a sublist. In practice, a "null" relation has been defined so that even the absent arcs are encoded in the ML representation of a graph. It is worth noticing that here only simple (i.e., without loops) undirected graphs are considered, so the relation linking the i-th node to the j-th node coincides with the relation linking the j-th node to the i-th node. For this reason, the length of the sublist associated to a node decreases as

[1]Graphs are formally defined in Appendix

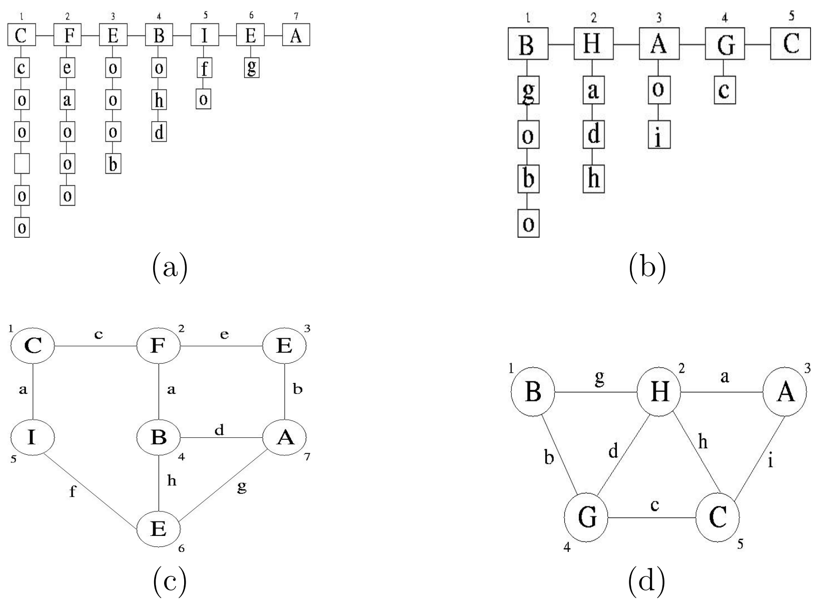


Figure 5.1: Two generic ML's (top) and the graphs encoded (bottom). The horizontal list is the main one and the vertical lists are the sublists. The elements of the set A_n are denoted by capital letters, while those of A_a are denoted by small letters. The null element is denoted by the letter 'o'.

the position of the node in the main list increases: the first sublist is made of (N-1) elements, the second sublist has (N-2) elements and so on. In fact, the information on the link between each node and the previous ones in the main list is already expressed in the previous sublists. As a consequence, the sublist of the last node of the graph is void. Thus a ML has a triangular shape: the base of the triangle is the main list and is long N, while the height is represented by the first sublist and is long (N-1). In the following, the operations defined on the ML's will be introduced.

In the defined encoding scheme, positional notation is used. Given an element of a sublist, the other node linked by the encoded arc is determined by the position of the element within the sublist. Moreover, the connected node referred by that element depends on the position of sublist within the main list. In practice, the i-th element of the j-th sublist contains the information on the arc connecting the i-th node and the $(j+i)$-th node. In the following will be seen that this encoding allows a sort of "invariance" property for some subgraphs encoded within a ML. This property will be useful in order to simplify operations as, for example, the merge of two graphs by means of the multilists encoded. Within the devised encoding scheme, a crossover operator that can be applied to two multilists of different length has also been defined. Moreover, this operator, differently from those defined in other approaches [GLW98, CAH+02] is implemented in a quite simple way, (in analogy with the operators used in classic GAs) without perform any search on the multilists given in input.

5.3 A Formal Definition of Multilist

In this section a formal definition of multilist is given. Moreover, some *operations* to be applied to multilists are also formally defined. This formalism has been conceived in order to characterize this data structure independently from its capability in encode graph. The purpose of this section is that of formally define the operations and the structures involved in the definition of the operators employed within the proposed approach. Then the formalism illustrated above allows one to formally characterize these operators.

Definition 1 *A multilist L is defined as a couple of lists:*

$$L = (l_M = \{n_1, n_2, \ldots, n_N\}, l_S = \{l_1, l_2, \ldots, l_N\})$$

The list l_M is called main list *and contains N elements, called* nodes *of the multilist. The list l_S is a list of lists, called* list of sublists. *The* length *of a multilist is defined as the length of its main list. In a multilist of length N the list of the sublists has the same length N. The couple of lists L is a multilist if*[2]

1. $\forall l_i \in l_S \qquad length(l_{(i+1)}) = length(l_i) - 1$*;*
2. *It exists at most one $l_k \in l_S$ such that:*
   $length(l_{(k+1)}) = length(l_k) + J \qquad 1 \leq k < N,\ J \in N - \{-1\}$*;*
3. $length(l_i) > 0 \qquad 1 \leq i < N$*;*
4. *if $length(l_N) > 0$ then*
   does not exist any l_k such that defined at point 2;

The value of J is called *jump* (point 2), while the value of k is called *jump point.* Third condition states that only the N-th sublist may be void. Fourth condition, instead, states that if there is a jump within the multilist, then the N-th sublist must be void. The elements of the sublist l_i, called *links* of the i-th node, are denoted by the set of symbols $\{e_{ij} | 1 \leq i, j \leq N\}$. The symbol e_{ij} denotes the j-th element of the i-th sublist. Moreover, the j-th element of the i-th sublist (l_i) represent the link between the node i and node j. The link between the node i and node j of the main list l_M is in:

[2]The function $length(l)$ compute the number of elements in the list l; see Appendix.

– the $(j-i)$-th element of the sublist l_i if $i < j$.

– the $(j-i)$-th element of the sublist l_j if $i > j$.

The set of links of the i-th element of the main list is called the *connection set* of the i-th node of the main list l_M. Hence, the connection set of the node i of l_M consists of the elements of the sublist l_i more the set [3] $S_C = \{e_{1(i-1)}, e_{2((i-2))}, \ldots, e_{(i-1)1}\}$.

5.3.1 Some Definitions on the Multilists

In the following some definitions concerning sublists will be given. These definitions characterize different types of MLs, which may be obtained from the application of some operations described afterwards.

Definition 2 *A multilist L of length N is called* complete *if does not contain jump point and the sublist l_N is the void list (length$(l_N) = 0$).*

Examples of complete MLs are given in Figure 5.1.

Definition 3 *A multilist L of length N is called* incomplete *if contains a jump $J \geq 0$.*

The jump point k is called *incompleteness degree* of the sublist, the value of J is called *incompleteness level.* The sublists l_i such that $1 \leq i \leq k$ are called *incompletes* (see Figure 5.2)).

Definition 4 *A multilist L of length N is called* redundant *if one of the following conditions hold:*

1. *It contains a jump[4] $J < -1$.*
2. *Does not contain jumps and the length of the sublist L_N is greater than zero (length$(l_N) > 0$).*

In the case 1 the jump point k is called *redundancy degree*, while the value of the jump is called *redundancy level.* The sublists preceding the jump point are called *redundants* so as the elements $e_{ij}: \quad 1 \leq i \leq k, (N-i) < j \leq (N-i+J)$ (see Figure 5.2). In the case 2, the sublist L is *totally* redundant: its redundancy degree equals the length of l_M and the length of the sublist l_N (*length*(l_N)) represents the redundancy level of the multilist; in this case all the sublists are redundants so as the elements e_{ij} such that $(N-i) < i < (N-i+J)$ (see Figure 5.3).

[3] In the following will be seen that, sometimes, some of the elements of the list S_C can be absent.

[4] Notice that the value of J is, by definition, different from -1.

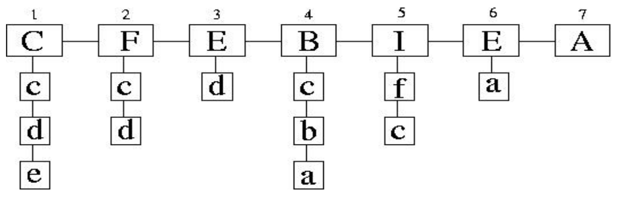


Figure 5.2: An incomplete multilist of length 7. Its incompleteness degree is 3 just as its incompleteness level.

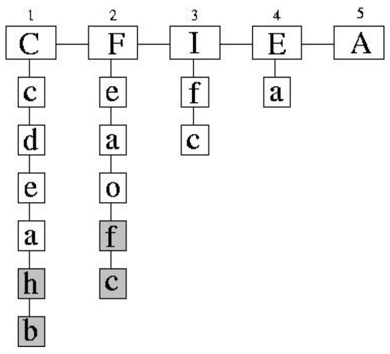


Figure 5.3: A redundant multilist of length 5. Its redundancy degree is 2, just as its redundancy level. The redundant elements are those shaded.

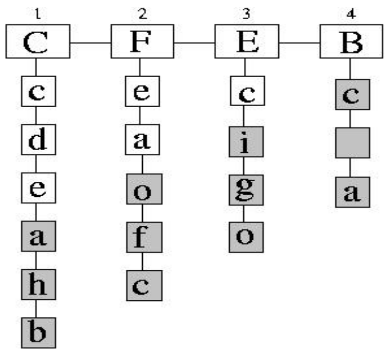
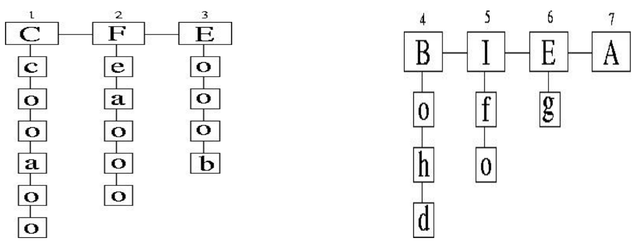


Figure 5.4: A multilist of length 4 totally redundant. Its redundancy level is 3. The redundant elements are those shaded.



(a) left multilist (b) right multilist

Figure 5.5: The two multilists generated by the 3-cut operation applied to the multilist of Figure 5.1(a).

5.3.2 The Cut Operation

Given a complete multilist $L = (l_M = \{n_1, n_2, \ldots, n_N\}, l_S = \{l_1, l_2, \ldots, l_N\})$ of length N, the operation *t-cut* ($1 \leq t < N$) applied to L consists of the following operations:

1. t-cut[5] of the main list l_M.

2. t-cut of the list of the sublists l_S.

The cut of l_M (point 1) yields two new lists l'_M and l''_M, the former consisting of the first t nodes of l_M, the latter of the remaining $(N - t)$ nodes of l_M. The t-cut of l_S (point 2), instead, produces two new lists of sublists. The first one consists of the first t lists of L while the second one contains the remaining $(N - t)$ sublists of L. The value of t is called *cut point.*

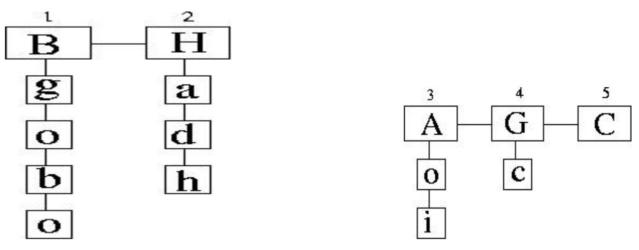
In a more formal way, the cut operation split the multilist L in two multilists (see Figure 5.5):

– $L' = (l'_M = \{n_1, n_2, \ldots, n_t\}, l_S = \{l_1, l_2, \ldots, l_t\})$;

– $L'' = (l''_M = \{n_{t+1}, n_{t+2}, \ldots, n_N\}, l''_S = \{l_{t+1}, l_{t+2}, \ldots, l_N\})$;

The first multilist, called *left multilist* contains the main list l'_M, which consists of the first t elements of the main list l_M of L; this multilist is totally redundant and its redundancy level equals the difference between the length of L and the cut point t. The second multilist called *right multilist*, contains the main list l''_M which consists of the remaining $(N - t)$ nodes of the main list l_M and of the remaining $(N - t)$ sublists of L. This multilist is complete.

[5]The t-cut operation for a list is defined in Appendix.



(a) Right multilist (b) Left multilist

Figure 5.6: The multilist of Figure 5.1(b) may be obtained applying the merge operation to these multilists. The left multilist has length 2 and is completely redundant, the right one has length 3 and is complete.

5.3.3 The Merge Operation

The *merge* operation yields a new multilist by joining two given multilists. This operation can be applied only if two requirements are satisfied. In fact, given two multilists:

– $L^{'} = (l_M^{'} = \{n_1^{'}, n_2^{'}, \ldots, n_{N'}^{'}\}, l_S^{'} = \{l_1^{'}, l_2^{'}, \ldots, l_{N'}^{'}\})$

– $L^{''} = (l_M^{''} = \{n_1^{''}, n_2^{''}, \ldots, n_{N''}^{''}\}, l_S^{''} = \{l_1^{''}, l_2^{''}, \ldots, l_{N''}^{''}\})$

of length respectively $N^{'}$ and $N^{''}$; the merge operation yields a multilist L if the following conditions hold:

1. The multilist $L^{'}$ is totally redundant. It is called *left* multilist.

2. The multilist $L^{''}$ is complete. It is called *right* multilist.

The merge of the multilist $L^{'}$ and $L^{''}$ consists of the following operations:

1. The main list l_M of the new multilist L is obtained concatenating the main list $l_M^{''}$ to the main list $l_M^{'}$. The length of l_M is the sum of the length of the main lists $l_M^{'}$ and $l_M^{''}$ ($length(l_M) = length(l_M^{'}) + length(l_M^{''})$). Hence, the length of the new multilist is $N^{'} + N^{''}$.

2. Concatenating the list of the sublists $l_S^{''}$ of $L^{''}$ to the list $l_S^{'}$ of $L^{'}$, a new list of sublists $l_S = \{l_1, l_2, \ldots, l_{(N'+N'')}\}$ consisting of $(N^{'} + N^{''})$ sublists is generated. In l_S, the first $N^{'}$ sublists are those of $L^{'}$ while the remaining ones $(N^{''})$ are those of $L^{''}$.

To summarize, the merge operation applied to the multilists $L^{'}$ and $L^{''}$ yields the multilist:

$L = (l_M = \{n_1^{'}, n_2^{'}, \ldots, n_{N'}^{'}, n_1^{''}, n_2^{''}, \ldots n_{N''}\}, l_S = \{l_1^{'}, l_2^{'}, \ldots, l_{N'}^{'}, l_1^{''}, l_2^{''}, \ldots l_{N''}\})$

This multilist will be:

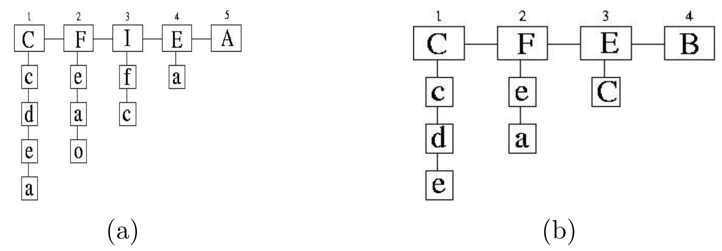


Figure 5.7: Multilist obtained from the reduction operation applied to the multilists of the Figure 5.3) and Figure 5.4 (figure b).

Complete if the redundancy level R of the left multilist equals the length of the right one ($R = length(L'')$) (see Figure 5.6).

Incomplete if the redundancy level R of the right multilist is less than the length of the right multilist ($R < length(L'')$).

Redundant if the redundancy level of R is greater than the length of the right multilist ($R > length(L'')$).

5.3.4 Further Operations on the Multilists

A redundant multilist L of length N, with redundancy degree d_r and redundancy level l_r can be transformed in a complete multilist by a *reduction* operation. Such a operation is carried out applying the l_r-shortening[6] operation to the first d_r sublists of L. In the case of a multilist completely redundant with a l_r redundancy level, the reduction operation can be obtained applying the l_r-shortening operation to all the sublists of L (see Figure 5.7).

An incomplete multilist L of length N, with d_i incompleteness degree and l_i incompleteness level can be transformed in a complete multilist by the *completion* operation. Such a operation is carried out concatenating a list of length l_i to the first d_i sublists of L (see Figure 5.8). Note that if N_A is the cardinality of the definition set of the sublists, then the number of the completions available for L is $N_A^{(d_i \cdot l_i)}$.

5.4 Graph Encoding by Multilists

As previously seen in Section 5.4, an undirected relational graph with attributes of N nodes can be encoded by a complete multilist. The incidence set of the node i is represented by the connection set of i-th element of the

[6]The l_r-shortening operation for a list is defined in Appendix.

main list. Thus, repeating the reasoning previously done to find the connection set of a node in a multilist, the relation associated to the arc l_{ij} can be found in this way:

– in the element $(j-i)$ of the sublist l_i if $i<j$.

– in the element $(j-i)$ of the sublist l_j if $i>j$.

Therefore, the incidence set of the node i is represented as the elements of the sublist l_i plus the set $S_I=\{e_{1(i-1)}, e_{2((i-2))}, \ldots, e_{(i-1)1}\}$. If $i=1$ then the set S_I is void and the incidence set of the node is represented by the sublist l_1. For $i=N$, instead, the set of the arcs of the node is represented by $S_I=\{e_{1(N-1)}, e_{2((N-2))}, \ldots, e_{(N-1)1}\}$, whereas the list l_N is void[7].

In the following, the graphs related to the different types of multilist (complete, redundant, etc.) previously described will be illustrated. The simplest case is that of a complete multilist of length N because it encodes a canonical graph[8] of N nodes. Instead, a redundant multilist L of length N and redundancy degree and redundancy level respectively equal to d_r and l_r encodes a cut graph of N nodes if at least one of the redundant elements of L contains a relation different from the null one (formally: G is cut if $\exists e_{ij} \neq \nu : \ 1 \leq i \leq d_r, \ (N-i) < j \leq (N-i+l_r))$, otherwise the graph encoded is canonical. The number of redundant elements not null is called *redundancy index*[9] of L. Finally, an incomplete multilist L_i of length N and redundancy degree and level respectively equal to d_i and l_i encodes a "part" of a graph G of N nodes, because it lacks the information related to the incomplete sublists. The information that lacks corresponds to the arcs which link the first d_i nodes to the last l_i nodes of the graph (formally: $l_{ij} : \quad 1 \leq i \leq d_i \quad (N-l_i) < j \leq N$). This multilist can be made complete by an operation of completion; if the elements added to the incomplete sublists are all null, then the number of arcs of the graph stays unchanged, because added information does not encode any arc. This particular type of completion is called *null completion* and, among all the available ones[10], is the least arbitrary; in fact, no information is added by arbitrarily adding new arcs to the graph, as the null completion transform the previously defined "part of a graph" in a canonical graph, without modifying the initial graph.

[7]By definition, the N-th sublist of a complete multilist is always void.

[8]The canonical, and the other types of graphs mentioned in the following are defined in Appendix.

[9]Practically it equals the number of suspended arcs in the graph encoded.

[10]if N_I is the cardinality of the definition set of the sublists, the number of completions for L_i is $N_I^{(d_i \cdot l_i)}$

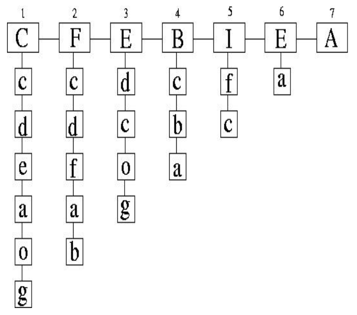


Figure 5.8: Multilist obtained from the completion operation applied to the multilist of Figure 5.4.

5.4.1 The Cut Operation

See now the effect of t-cut operation applied to a graph G of N nodes encoded by a multilist $L = (l_M, l_S)$ of length N (see Figure 5.9). As previously seen, this operation applied to L produces a left multilist L_l totally redundant (its redundancy level is $(N - t)$) and a right multilist complete L_r. The left multilist encodes a graph $G^{'}$ (see Figure 5.9(a)) consisting of t nodes, which is cut[11] if at least one of the first t nodes of G, corresponding to the first t elements in l_M, is connected to one of the remaining $(N-t)$ nodes of G. The right multilist, instead, (Figure 5.9(b)) encodes a graph $G^{''}$, which is always canonical and consists of the remaining $(N - t)$ nodes of l_M. Therefore, the number of suspended arcs in $G^{'}$ equals the number of arcs which connect the first t nodes of G to the remaining $(N - t)$ nodes.

Note that, thanks to the encoding scheme adopted, given the i-th node of l_M, the subgraphs in G represented by the last $(N-i)$ nodes and the related sublists are invariant to all the t-cut operation with $t < i$. The same happens for the subgraphs encoded in the multilist represented by the first i nodes and the related sublists, respect to all t-cut operation with $(t > i)$. Thus the subgraphs disrupted by a t-cut operation are those (and only those) encoded in the multilists cut by the t-cut operation applied to L. Hence, the t-cut operation disrupt a well determined set of the subgraphs of G, remaining unchanged the other ones, and this set is determined by the value of t. This fact offers the advantage that the subgraphs to be cut can be determined without any search on the encoded graph.

[11]If the graph G encoded by L is connected, $G^{'}$ is always cut.

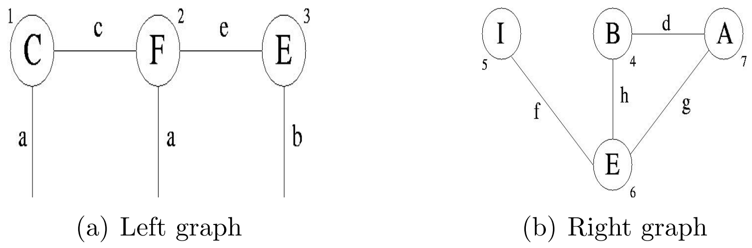


Figure 5.9: The two encoded graphs by the multilists produced by the application 3-cut operation to the multilist of Figure 5.1(a) (see Figure 5.5).

5.4.2 The Merge Operation

See now what happens if, given two graphs $G^{'}$ and $G^{''}$ consisting respectively of $N^{'}$ and $N^{''}$ nodes, merge operation is applied to the multilists $L^{'}$ and $L^{''}$ encoding them. As already said, merge operation can be applied only if the following conditions hold:

1. The multilist $L^{'}$ is totally redundant (the left one).
2. The multilist $L^{''}$ is complete (the right one).

the graph $G^{'}$ can be either cut or canonical, while $G^{''}$ must be a canonical graph. Depending on circumstances, the merge operation yields different types of graphs:

1. If $G^{'}$ is canonical (the redundant nodes in $L^{'}$ are null ones), then merge generates an unconnected graph G, in which the subgraphs made by $G^{'}$ and $G^{''}$ are unconnected.
2. If $G^{'}$ is cut and $N^{'} < N^{''}$, then L, the ML resulting from the merge operation, is complete and the encoded graph G is canonical.
3. If $G^{'}$ is cut and $N^{'} > N^{''}$, then L, the ML resulting from the merge operation, is redundant. In this case, G is canonical if the sublists of L do not contain redundant elements. On the contrary, if exist at least one of these non null elements then G is cut, and the number of the suspended arcs in G equals that of these non null elements.

Note that, thanks to the encoding scheme adopted, the graph $G^{''}$ encoded by the right multilist $L^{''}$ stay unchanged, regardless of the left multilist to which it is merged. The same happens for the uncut subgraphs contained in left multilist. As regards the latter subgraphs, it may be observed that, if the redundancy level of the left multilist is less than the length $N^{''}$ of the right multilist then these subgraphs are joined with some subgraphs contained in the right multilist. Instead, if this degree is greater than $N^{''}$, then G is a cut graph, since the resulting multilist L is redundant. Notice that the choice of the subgraph encoded in $L^{'}$ and $L^{''}$ to be merged does not depend on any parameters, but is determined by the multilists to be merged. This fact allows to avoid any search of subgraphs within the graphs to be merged.

5.5 Operators

In this section the operators defined for multilists are illustrated. Two operators have been defined, the first one, called *crossover* belongs to the class of recombination operators, while the second one, called *mutation*, is a mutation operator. The defined operators are problem independent. Moreover, crossover is able to swap subgraphs between two input graphs in a quite simple way, without performing any search on the input graphs. In the following both operators will be illustrated.

5.5.1 Crossover

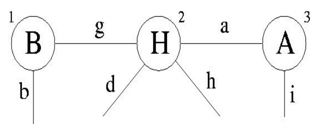
As mentioned in Section 3.2.2 the effects of the crossover operator on the search process performed by any EC–based algorithm have been extensively studied since the seminal work of John Holland. In fact, the ability to produce fitter and fitter partial solutions by combining building blocks provided by recombination operators is believed to be a primary source of the search power of any EC–based algorithm in which a recombination operator is defined.

While the crossover is easy to implement for strings and trees because these data structure can be divided into two pieces at any point, the same does not occur for graphs, since they can be split only breaking a variable number of arcs. The main difficulties of this operator in the graph domain can be summarized as follows:

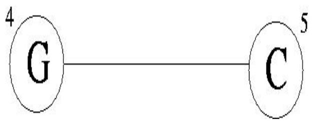
– A graph can't be divided into two parts choosing a single point. This operation implies the choice of a variable number of arcs to be broken[12].

– Subgraphs produced by splitting may have more than one crossover point (suspended arcs) requiring reattachment during the recombination of them.

[12]This arc will be referred in the following as "suspended arcs", see Appendix.



(a) Left graph



(b) Right graph

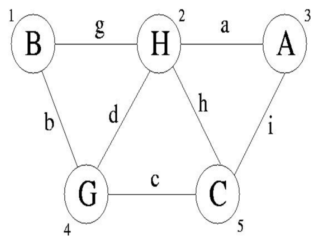


Figure 5.10: The graphs corresponding to the multilists of Figure 5.6 (top). The graph produced by the merge operation applied to the multilists of Figure 5.6 (bottom).

– When two fragments are merged they may have different numbers of suspended arcs to be merged.

– An effective crossover in the graph domain must be able to create and destroy subgraphs within the graphs to be crossed.

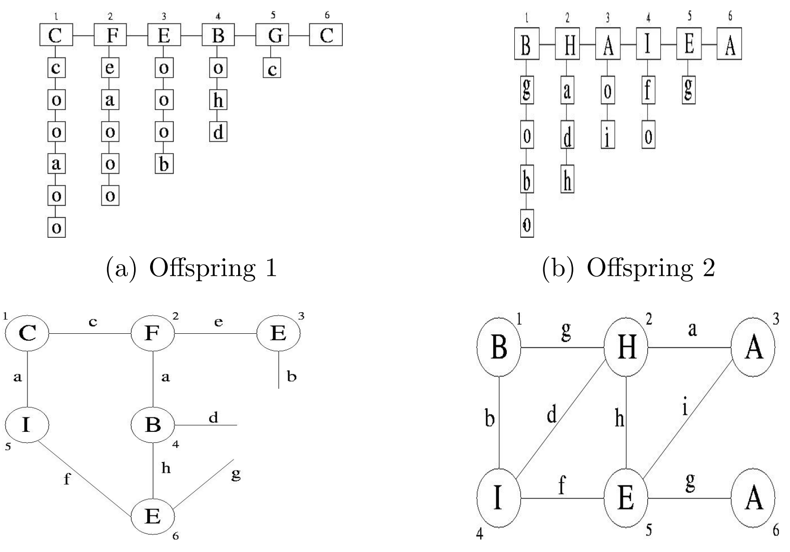
See now how the crossover operator has been implemented. The crossover is applied to two multilists $L^{'}$ and $L^{''}$ called in the following *parents* and generates two new multilists $M^{'}$ and $M^{''}$, called *offspring*. If the parents are length respectively N_1 and N_2, the length of the offspring may vary in the interval $[2,(N_1+N_2)-2]$. Crossover is based on two operations previously defined, t-cut and merge and is applied by accomplishing the following steps:

1. Choose randomly, with a uniform distribution of probability, a number t_1 in the interval $[1,N_1-1]$. Apply t_1-cut operation to $L^{'}$;

2. Choose randomly, with a uniform distribution of probability, a number t_2 in the interval $[1,N_2-1]$. Apply t_2-cut operation to $L^{''}$.

3. Merge operation is applied to: the left multilist produced by the t_1-cut operation applied to $L^{'}$ (point 1) and the right multilist produced by the t_2-cut operation applied to $L^{''}$ (point 2);

4. Apply the merge operation to: the left multilist produced by the t_2-cut operation applied to $L^{''}$ (point 2) and the right one obtained from the application of the t_1-cut operation to $L^{'}$ (point 1);



(a) Offspring 1 (b) Offspring 2

(c) Offspring 1 (d) Offspring 2

Figure 5.11: The offspring obtained by the crossover operator, with $t_1 = 4$ and $t_2 = 3$, applied to the multilists of Figures 5.1(a) and 5.1(b) (top). The encoded graph (bottom). The first ML obtained (left) is redundant and the related graph is cut. The second one ML (right) is incomplete, while the encoded graph is canonical.

Crossover can produce multilists which, depending on circumstances can be: complete, redundant, or incomplete. The conditions which determine the occurrence of the different cases are listed below (we assume $N_1 \geq N_2$).

$\boldsymbol{(N_1 - t_1) > (N_2 - t_2)}$ (Figure 5.11)

- Multilist M' is redundant.
- Multilist M'' is incomplete.

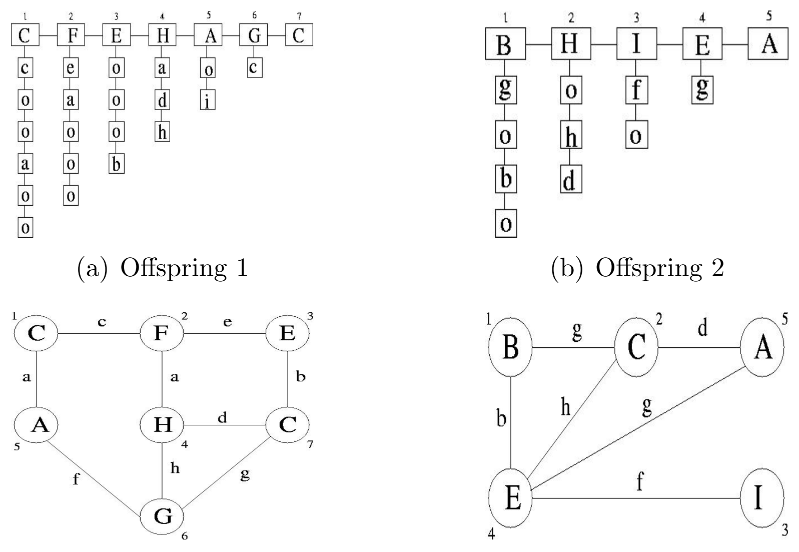
$\boldsymbol{(N_1 - t_1) = (N_2 - t_2)}$ (Figure 5.12)

- Both multilists M' and M'' are complete.

$\boldsymbol{(N_1 - t_1) < (N_2 - t_2)}$ (Figure 5.13)

- Multilist M' is incomplete.
- Multilist M'' is redundant.

Depending on the circumstances, to the offspring generated by the application of the crossover operator, the following operations are applied:



(a) Offspring 1 (b) Offspring 2

(c) Offspring 1 (d) Offspring 2

Figure 5.12: The offspring produced by the crossover, with $t_1 = 3$ and $t_2 = 1$, applied to the multilists of Figures 5.1(a) and 5.1(b) (top) and the encoded graph (bottom). Both the multilists are complete, it follows that the encoded graphs are canonical.

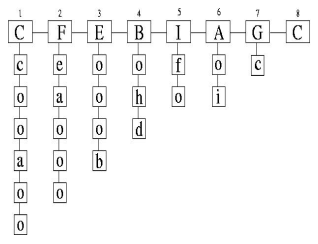
complete
The multilist stays unchanged.

incomplete
The multilist is made complete by the null completion operation.

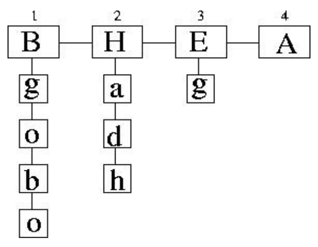
redundant
The multilist is made complete by the reduction operation.

It is worth noting that the crossover operator just defined, thanks to the t-cut operation, is such that the choice of the points (arcs) involved in the split are uniquely determined by the choice of a unique point in the main list (on its turn fixed by the value of t). Hence, by using this encoding scheme, the split of a graph can be obtained at any point, thanks to the fact that a ML can be split choosing just one point. Also the recombination of two graphs, is obtained in a quite simple way. In fact, the choice of the nodes to link to the suspended arcs is intrinsically determined by the MLs involved in the merge operation. Moreover, this operator may either create or destroy subgraphs.

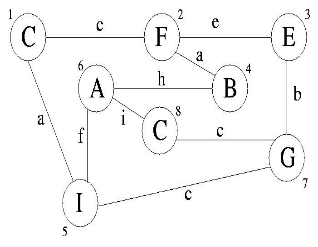
Note that the defined operator is quite simple and, differently from other approaches evolving graphs adopting crossover, does not perform any search



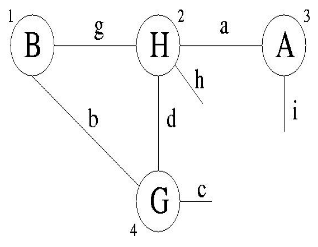
(a) Offspring 1



(b) Offspring 2



(c) Offspring 1



(d) Offspring 2

Figure 5.13: Offspring produced by the crossover, with $t_1 = 5$ and $t_2 = 2$ applied to the multilists of Figures 5.1(a) and 5.1(b). First multilist (left) is incomplete and the encoded graph is canonical. Second multilist (right) is redundant and encodes a cut graph.

on the graphs to be crossed. In [CAH$^+$02], for instance, the crossover is implemented by randomly choosing a fixed number of subgraphs for each individual to be crossed. Hence, this operator depends on this choice. Moreover, in growing graphs this number raise exponentially and this operator becomes unfeasible. In [GLW98], instead, the splitting of the graphs to be crossed is performed randomly choosing an initial arc of the graph and then finding and breaking all the path (randomly choosing one of the arcs in the path) between the nodes connected by the initial arc chosen. Then also this operator needs some searching on the graphs to be crossed. In the defined approach, instead, the split of a graph is obtained in a quite simple way with the t-cut operation, in which the subgraph obtained from the split is determined by the value of t. Hence, the randomness of the choice of the subgraphs to be split is related to the random choice of the value of t.

5.5.2 Mutation operator

The mutation operator defined here, is such that it can be defined as "micro" mutation, because it does not modify the structure of the multilist to which it is applied, but just the values contained in the elements, both of the main list and of the sublists. Such operation is based on a real number in the

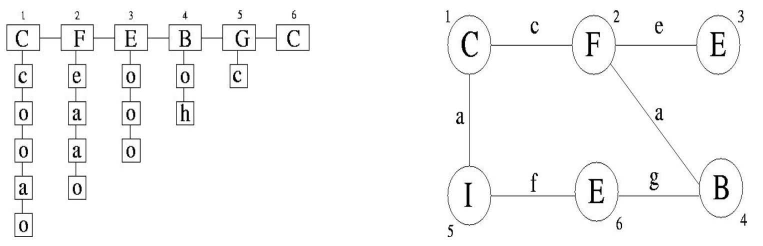


Figure 5.14: The multilist (left) and the encoded graph (right) after the application of the reduction operation to the multilist of Figure 5.11(a).

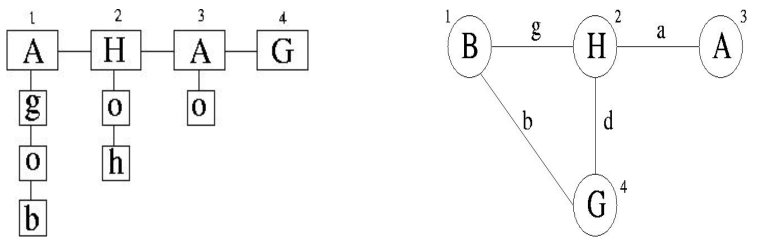


Figure 5.15: The multilist (left) and the encoded graph (right) obtained after the application of the reduction operation to the multilist of Figure 5.13(b).

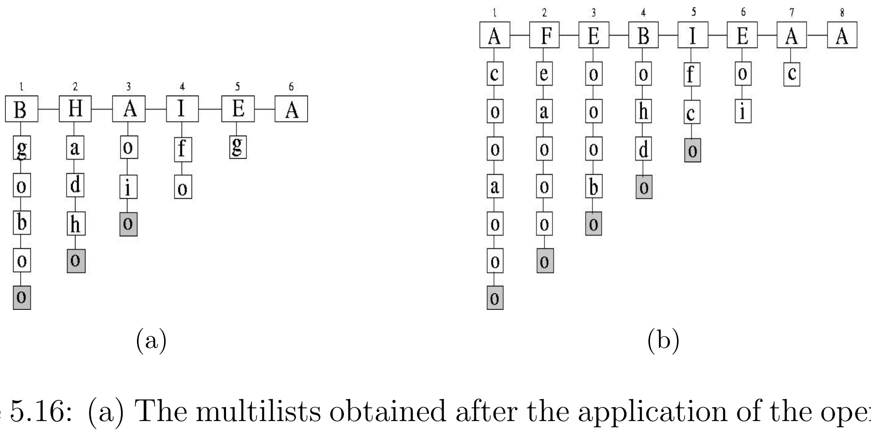


Figure 5.16: (a) The multilists obtained after the application of the operation of null completion to the multilists of Figure 5.11(b) (left) and figure 5.13(b) (right). The elements added are those shaded. The encoded graphs stay unchanged respect to that of Figures 5.11(d) and 5.13(c) respectively.

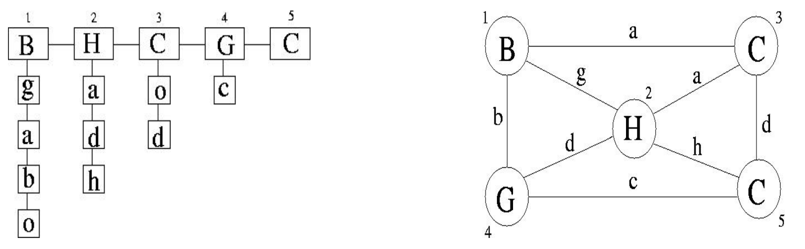


Figure 5.17: The ML (left) and the graph encoded (right), derived from the application of the mutation operator, with a probability equal to 0.1, to the ML of Figure 5.1(b). The mutation modifies the attribute of node 3 and that associated to the arc which links nodes 3 and 5. Moreover, the mutation added a new arc, which links nodes 1 and 3, absent in the graph before of the application of the mutation operator.

interval $[0, 1]$, called *mutation probability*.
Let L a multilist

$$L = (l_M = \{n_1, n_2, \ldots, n_N\}, l_S = \{l_1, l_2, \ldots, l_N\})$$

of length N, the mutation operator with a probability p_m is carried out by accomplishing the following steps:

1. For each of the n_i nodes of the main list:
   (a) Randomly, with uniform distribution probability, a real number r in the interval $[0, 1]$ is generated.
   (b) **If** $(r \leq p_m)$ **then**
      i. Choose randomly, with uniform distribution probability, an element in the set $I(l_M)$.
      ii. Substitute n_i for the value found in the previous step.
2. For each of the elements e_{ij} of the N sublists in l_S:
   (a) Randomly, with uniform distribution probability, a real number r in the interval $[0, 1]$ is generated.
   (b) **If** $(r \leq p_m)$ **then**
      i. Choose randomly, with uniform distribution probability, an element in the set[13] $(I(l_S) \bigcup \{\nu\})\}$.
      ii. Substitute e_{ij} for the value found in the previous step.

[13]The symbol ν denotes the null relation.

If L is a generic multilist and $L^{'}$ is the one produced by the application of the mutation operator to L, let's see which can be the differences between the graphs G and $G^{'}$ encoded respectively by L and $L^{'}$. G and $G^{'}$ contain the same number of nodes. $(|G| = \left|G^{'}\right|)$. Only the attributes associated to the nodes of the two graphs may vary. A quite different reasoning must be done for the arcs, because the number of arcs in the two graphs can be different. In fact, the substitute for a non null element e_{ij} of L can be either a different element of the definition set of the sublist or the null element (point 2(b)i); in this case, the graph $G^{'}$ does not contain the arc l_{ij} encoded by the element e_{ij}, because after the mutation the latter has became null, thus it encodes an absent arc. The opposite case may occurs as well: an absent arc in G can be present in $G^{'}$. It happens if a null element e_{ij} of L is substituted by a not null element. If, finally, the not null element e_{ij} of L is substituted for a different not null element, then both graphs G and $G^{'}$ contain the arc l_{ij}, but the relation associated to the arc in G is different from that in $G^{'}$. Notice that diversity between the graphs G and $G^{'}$ is directly proportional to the mutation probability p_m.

It is worth noting that the encoding scheme employed allows to implement the mutation in a quite simple way. Besides, with this unique operator it is possible a lot of actions on the graph encoded with a unique operation. Particularly, as concerns the arcs, they can be changed, added or deleted. Nodes, instead, can only be changed by this operator. However, as previously seen, the variation of the number of nodes in the graphs to be evolved is provided by the crossover operator.

5.6 Testing the approach

The proposed approach has been tested on a planning and optimization problem. To cope with this kind of problems, several approaches have been proposed in the literature, including genetic programming [HG04], simplex method [Wri98], simulated annealing [Hur02], Tabu search [LK00] and genetic algorithms [KLL98]. In [HG04], the wireless access point configuration problem, a hard non-linear optimization problem, has been considered. The same problem has been taken into account here in order to compare the obtained results with those presented in [HG04].

The scenario of the problem is the following: a community is planning to provide wireless Internet service to its citizens (clients) who are scattered around a given area. A certain number of access points need to be placed to cover all clients, because each access point has a limited service radius. All access points are wired and one of them is connected to an Internet gateway.

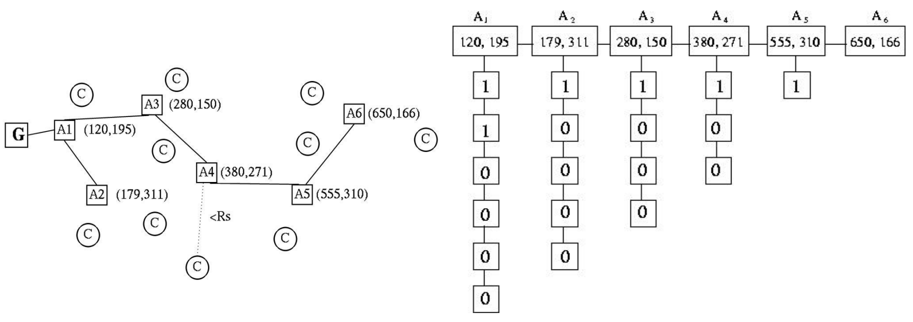


Figure 5.18: An instance of the wireless access point configuration problem (left) and the multilist encoding it (right). The citizens (clients) are labeled as circled C, while the AP's are reperesented by squares.

The design problem consists in determining the optimal configuration of the AP's in the area to cover. To reduce the cost, a configuration with minimal number of AP's and minimum length of the wires connecting them is considered optimal. According to the constraints imposed, the wireless access point configuration problem can be formulated in different ways. E.g., [KY03] assumes that the AP's are located at a specified set of possible points. Here, it is assumed that the AP's can be located at any place. More precisely, the problem to solve is defined as follows:
GIVEN a set of N clients located at (x_i^c, y_i^c) $\;i = 1 \ldots N$ in an area of size $W \times H$ where $x_i^c \in [0, W]$ and $y_i^c \in [0, H]$, and the gateway G located at (x^g, y^g), let us assume that all AP's are equal and that the service radius of an AP is r_s;
FIND a configuration of wired access points located at $(x_i^{AP},\ y_i^{AP})$ with $i = 1 \ldots N_{AP}$, connected to the gateway port G in such a way that each client is covered by at least one AP and the total cost of the AP's and the wires is minimal. Thus, let C_{AP} be the cost of each AP and C_w the cost of a unit length wire, the aim is:
minimize $f = C_{AP} * N_{AP} + C_w * \sum |L_i|$
where the L_i are the lengths of the connections among AP's.

The Fitness Function

In order to solve the problem taken into account, a configuration of AP's is represented with a graph whose nodes are the AP's and whose arcs are the wire segments connecting the AP's. The set of node attributes is made up of the AP coordinates in the area to cover (see Figure 5.18). In the problem at hand, it is necessary to know only which nodes are linked to a given node. Hence, in the ML representation encoding the graph, the value 1 is used to indicate the presence of an arc, while the 0 indicates the absence of an arc.

As mentioned in the previous section the fitness function has to consider three aspects of the problem: the percentage of covered clients, the number of AP's employed and the total length of the wires connecting them. For this reason, the fitness function is the weighted sum of three terms. The first term F_{cover} measures how well clients are covered by the configuration of AP's: the more clients are covered, the better. The second term F_{wires} should measure how good the connection topology is: the shorter the wires used, the better. Finally, the term F_{AP} should estimate the goodness of a configuration as regards the number of wireless AP's employed: now, the fewer AP's are used, the better. It may be convenient that the fitness terms are normalized and suitably weighted, so as to reflect their different importance when the goodness of a configuration is evaluated. Since the aim of the work presented here is that of presenting a general purpose method for graph generation, for the specific problem considered we have adopted the same fitness function as proposed in [HG04], in order to ascribe any difference in the performance of the methods to the way the solutions are generated, not to the way they are evaluated. Namely, the fitness terms are:

$$F_{\text{cover}} = \frac{4.0 * C_c}{C_c + C_T}; \quad F_{\text{wires}} = \frac{10000}{10000 + L_w}; \quad F_{\text{AP}} = \frac{C_T}{C_T + 1.5 * N_{AP}} \tag{5.1}$$

where C_c is the number of clients covered, C_T the total number of clients, N_{AP} the number of AP's and L_w is the total length of wire segments connecting the AP's. Then, the fitness function F_{tot} is the weighted sum of the above three terms:

$$F_{\text{tot}} = 0.7 * F_{\text{cover}} + 0.1 * F_{\text{wires}} + 0.2 * F_{\text{AP}} \tag{5.2}$$

Moreover, solutions (i.e., configurations) containing isolated AP's are penalized by multiplying their fitness by 0.5.

Chapter 6

Experimental Results

In order to asses the effectiveness of the approaches proposed in the previous chapters, several experiments have been performed. In this chapter the obtained results are illustrated. Experiments have been performed both on the EC–based framework described in Chapter 4 and on the evolutionary method for graph generation described in Chapter 5. The behavior of two applications based on the framework has been investigated on several data sets, including medical data, remote sensed images and synthetically generated data. Instead, the ability of the proposed approach for graph generation has been investigated by considering an hard non–linear optimization problem concerning the layout of access points in the planning of a wireless LAN. Moreover, results obtained by the implemented systems have been compared with those obtained by previously proposed approaches. The comparisons performed have confirmed the effectiveness of both approaches.

In Section 6.1 results obtained by the CFG-GP version of the framework described in Chapter 4 are presented; Section 6.2 reports, the results of the LVQ version of the framework are illustrated; finally, in Section 6.3 experiments performed to test the method devised for graph generation are described.

6.1 The CFG-GP Version of the Framework

In this section, the experiments carried out on the system that implements the CFG-GP application of the framework defined in Chapter 4 are described. As already said in Section 4.3, this application generates prototypes consisting of sets of logical expressions, which represent patterns described as feature vectors. A Prototype is made of predicates describing feature values of the patterns belonging to the class which it represents (see Figure 4.7). Given

Name	**Classes**	**Features**	**Size**
IRIS	3	4	150(50+50+50)
BUPA	2	6	345(145+200)
Vehicle	4	18	846(212+217+218+199)

Table 6.1: The data sets used in the experiments. The class distribution is shown between brackets in the last column.

a set of prototypes, an unknown patterns is recognized as belonging to the class represented by the prototype whose predicates satisfies the values in the feature vector describing the pattern.

The effectiveness of the implemented system has been tested on three publicly available data sets. These data sets have been chosen in order to compare the performance of the developed system with those obtained by the method described in [MPD04]. All the considered data sets refer to real data, named IRIS [And35], BUPA [BM98] and Vehicle [BM98]. Last two sets have been extensively studied in [LLS00].

6.1.1 Data sets

Considered data sets (see Table 6.1) have been divided in two parts, a training set and a test set. These sets have been randomly extracted from the data sets and are disjoint and statistically independent, in order to ensure that the recognition rate obtained on the test set by the evaluated classifier represent a reliable assessment of the performance on unknown patterns. The first one has been used during the training phase to evaluate, at each generation, the fitness of the individuals in the population. The second one has been used at the end of the evolution process in order to evaluate the performance obtained by the best set of prototypes evolved. In particular, the recognition rate on the test set has been computed using a classifier implemented by choosing the best individual generated during the training phase. In the following the data sets analyzed will be detailed.

IRIS

This is the well known Anderson's Iris data set [And35]. It consists of 150 patterns characterized by four features representing measures taken on iris flowers of three different classes, equally distributed in the set. The four features are sepal length, sepal width, petal length and petal width.

BUPA

The BUPA liver disorders data set consists of 345 patterns distributed in two classes of liver disorders. Six features characterize each pattern.

Vehicle

The patterns of this data set are images of 3D objects (vehicles). The data set is made of 846 patterns distributed in four classes. Each pattern is described by a vector of 18 features.

6.1.2 Data Normalization

In order to make the method independent from the ranges of values assumed by the features of the patterns contained in the data sets considered, these features have been normalized in the range $[-1.0, 1.0]$. The normalization of pattern features makes the grammar employed to generate the derivation trees independent from the range of values of the pattern feature, allowing to strongly reduce the differences among grammars (see Table 4.1) to be used for two different data sets. Actually, thanks to data normalization, it exists only one difference between two grammars to be used for generating prototypes, i.e. expressions, for different data sets: the number of features describing the patterns. Then, only the fifth rule of Table 4.1 have to be modified in order to generate the right number of variables needed to represent the patterns.

Given a not normalized pattern $\mathbf{X} = (x_1, \ldots, x_N)$, every feature x_i is normalized using the formula:

$$x_i = \frac{x_i - \overline{x}_i}{2\sigma_i}$$

where $\overline{x}_i$ and σ_i, respectively represent the mean and the standard deviation of the i-th feature computed over the whole data set.

6.1.3 Parameter Settings

The values of the evolutionary parameters, used in all the performed experiments, are summarized in Table 6.2. As regards the maximum total number of nodes (N_{max}) and the maximum length (L_{max}), i.e. maximum number of prototypes, allowed for an individual, they have been a priori set[1].

[1]Individuals that do not satisfy both requirements are *killed*, i.e. marked in such a way that they are ignored in the selection phase.

Parameter	symbol	value
Population size	P	200
Tournament size	$\mathcal{T}$	6
Elitism size	e	5
Crossover probability	p_c	0.4
Mutation probability	p_m	0.8
Number of Generations	N_G	300
Maximum number of nodes	N_{max}	1000
Maximum length of an individual	L_{max}	20

Table 6.2: Values of the evolutionary parameters used in the experiments on the CFG-GP system implemented.

Each of the other parameters reported in Table 6.2 have been chosen among a set of values and a set of experiments have been executed. The set of values considered for each of the parameters tuned is listed below:

$$\begin{aligned}
P &= \{50, 100, 150, 200, 250, \mathbf{300}, 350, 400, 450, 500\} \\
\mathcal{T} &= \{4, 5, \mathbf{6}, 7, 8, 9, 10\} \\
E &= \{1, 2, 3, \mathbf{4}, 5, 6, 7, 8, 9, 10\} \\
p_c &= \{0.1, 0.2, 0.3, 0.4, \mathbf{0.5}, 0.6, 0.7, 0.8.0.9\} \\
p_m &= \{0.1, 0.2, 0.3, 0.4, \mathbf{0.5}, 0.6, 0.7, 0.8, 0.9\} \\
N_G &= \{100, 300, \mathbf{500}, 800, 1000\}
\end{aligned}$$

These sets have been determined by performing a set of preliminary trials. During the tuning of a specific parameter, the values of all the other ones have been kept constant. These default values have been reported above in bold type. For each of the values to be tested 10 runs have been executed. For each run, the fitness of the best individual generated has been stored, and then the mean and the corresponding standard deviation of these fitness values have been computed. The value that has achieved the best mean have been chosen. Moreover, parameters have been tuned in the order reported in Table 6.2.

6.1.4 Results

In order to compare the obtained results with those reported in the literature, the 10-fold cross validation procedure has been used. In this procedure, the performances of a classifier on a data set $\mathcal{D}$ are evaluated by randomly dividing $\mathcal{D}$ into 10 disjoint sets of equal size $N/10$, where N is the total

Data sets	R_2	$\mathbf{R_1}$	σ_{R_1}	N_{P_1}	$\sigma_{N_{P_1}}$
IRIS	98.67	**99.4**	0.5	3.03	0.2
BUPA	69.87	**74.3**	3.0	2.36	0.5
Vehicle	61.75	**66.5**	2.0	4.8	0.6

Table 6.3: The average recognition rates (%) R_1 and R_2 for the classifier C_1 using the generated prototypes and the comparison classifier C_2. The standard deviation in case of C_1 is given. The average number N_P of prototypes found by C_1 and the related standard deviation are also shown.

number of patterns in $\mathcal{D}$. Then the classifier is trained 10 times, each time with a different set held out as a test set. In order to evaluate the performance of the classifier on unknown data, the performance is computed as the mean of the results obtained on the ten different test sets [DHS01]. For each of the 10 test sets, 10 runs have been executed, hence the total number of runs executed equals to 100.

After the tuning of the parameters, the behavior of the system implemented has been investigated. The main aspect of the analysis has regarded the generalization power, i.e. the ability of obtaining a similar rate on a different data set (the test set). In fact, in a learning process, in most cases, when the maximum recognition rate is achieved, the generalization power may not be the best [DHS01]. In order to investigate such ability for the developed system, the recognition rates on training and test set have been taken into account for the different considered data sets. In Figures 6.1 and 6.2 such recognition rates, evaluated every 50 generations, in a typical run for the BUPA and Vehicle data sets, are displayed. It can be observed from the figure that, in the experiments carried out, the recognition rate increases with the number of generations both for the training set and for the test set. The best recognition rates occur in both cases nearby generation 250. Moreover, the fact that the difference between the two recognition rates tends to increase when that on the training set reaches its maximum, suggests that the use of a *validation* set could further improve the classifier performances. The recognition rate obtained on the validation set would be used to evaluate the fitness of the individuals instead of that obtained on the training set. The latter set would be used only to accomplish the dynamic labeling of prototypes.

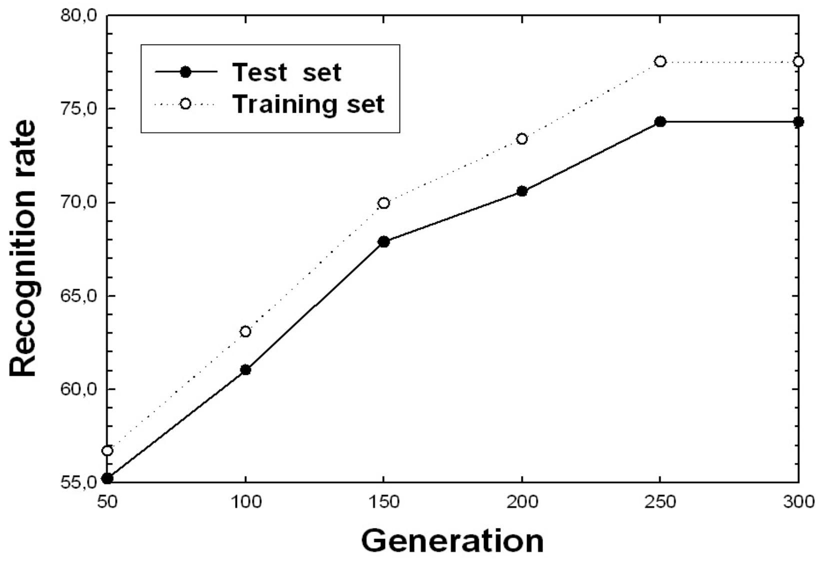


Figure 6.1: Recognition rate obtained on training and test sets during a typical run for BUPA data.

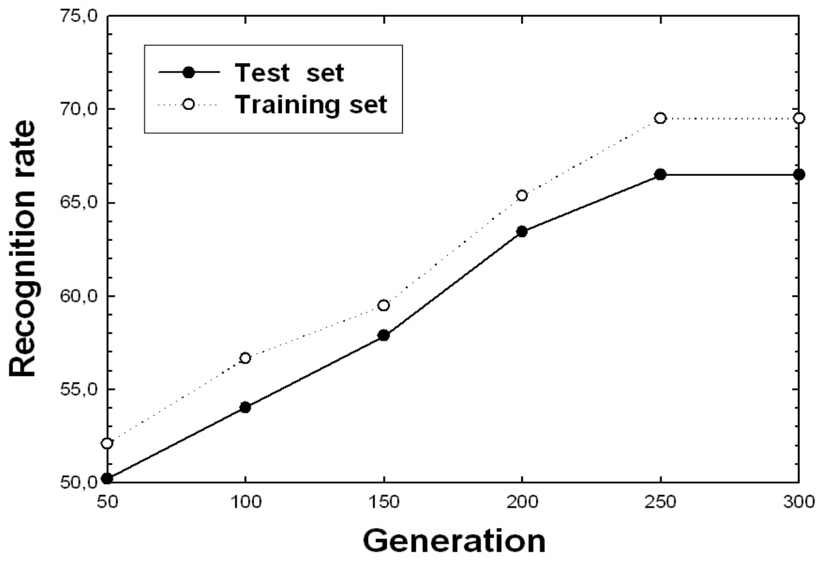


Figure 6.2: Recognition rate obtained on training and test sets during a typical run for Vehicle data.

6.1.5 Comparison Findings

The proposed approach has been compared with another GP based approach [MPD04] in which an individual consists of a set of expressions (i.e. prototypes) involving simple arithmetic operators, constants and feature variables. Each expression establishes conditions on the values of a variable number of features characterizing the data to be analyzed. The patterns are matched against the expressions and assigned to the one satisfying the constraints on feature values.
Similarly to the devised approach, each individual is encoded as a multitree, but the number of trees (i.e. expressions) for each individual is constant and a priori fixed equal to the number of classes of the problem at hand. Moreover, each tree is a priori labeled: the first tree with the label of the first class, the second tree with that of the second class and so on. As a consequence, a pattern belonging to the i–class is correctly classified only if it is assigned to the i–tree of the individual.

In Table 6.3 the recognition rates obtained, by using the 10–fold validation method mentioned above, by the two methods are shown. Since the GP approach is a stochastic algorithm, the standard deviations are also shown. Moreover, the average number of prototypes found by the proposed method (represented by valid expressions in the considered individual) and the related standard deviation are reported.
For the IRIS data set, 99.4% of the test set has been correctly recognized by using the prototypes evolved by the developed system, against 98.67% obtained by the method considered for comparison. Instead, for the BUPA data set the generated prototypes has been able of correctly recognizing 74.3% of the test set, a rate significantly better than that obtained in [MPD04] (69.87%). Also for the Vehicle data set, the generated prototypes by the system performs significantly better than the comparison method, achieving on the test set the recognition rate of 66.5%, against 61.75% obtained by the other method.

Summarizing, the comparison performed demonstrates that the proposed method outperforms the other one compared on all the data sets taken into account. Thus, the comparison carried out confirms the validity and the effectiveness of the proposed approach. In my opinion, the outperforming obtained by using the proposed approach, as already said in Section 4.2, depends on two main features of the proposed approach: (i) the ability to automatically find the needed number of prototypes; (ii) the relaxation of the constraint caused by the a priori labeling of the prototypes.
For the IRIS data set, the average number of prototypes found equals to 3.03; 3 prototypes have been found in 97 runs of the 100 ones performed, while 4

prototypes have been found in the remaining 3 runs. The results obtained on this data set have demonstrated that the developed system is able to find the minimum number of prototypes, even when this number coincide with the number of classes defined in the problem.
For the BUPA and Vehicle data sets the number of prototypes found by the implemented system has been respectively 2.36 and 4.8 For the former data set the system have found 2 prototypes for most of the runs executed (65), for the latter data set, instead, the system has been able to find 5 prototypes in most of the runs performed (80). Better performances have been obtained in the runs in which 5 prototypes have been found. In this case, the system has been able to discover that, most likely 5 prototypes better characterize the data.

6.2 The LVQ Version of the Framework

In this section, the experimental activities carried out on the implementation of the LVQ version of the framework defined in Chapter 4 are illustrated. As mentioned in Section 4.4 this application is able to generate set of real–valued vectors to be used as prototypes of patterns represented as points in a feature space. Given a set of generated prototypes, unknown patterns are classified by first computing the Euclidean distance between the pattern and each of the prototypes and then assigning the pattern to the closest prototype.

The effectiveness of this version of the framework has been tested on several data sets. In the following subsections the experiments performed for each of the different kind of data will be illustrated. For brevity sake, parameter setting phases will not be detailed, although the parameters used in each of the set of experiments will be shown.

6.2.1 Synthetically Generated Data

The first set of experiments for the LVQ version of the framework has been performed on four data sets synthetically generated. Each data set contains 5000 patterns, equally distributed in five classes. Each pattern is a feature vector consisting of 2 real elements.

For each data set 20 runs have been performed. The values of the evolutionary parameters employed in the runs are shown in Table 6.4. As regards the minimum number (N_{min}) and the maximum number (N_{max}) of prototypes allowed in a individual, they have been set respectively to 5 (the number of classes actually present or, in other words, of labels to be assigned) and 20. In Figures 6.3–6.6, for each data set both patterns synthetically generated

Parameter	symbol	value
Population size	$\mathcal{P}$	100
Tournament size	$\mathcal{T}$	7
elithism size	$\mathcal{E}$	5
Crossover probability	p_c	0.4
Mutation probability	p_m	0.08
Number of Generations	N_g	1000

Table 6.4: Values of the basic evolutionary parameters used in the experiments performed on the four synthetic data sets and on the NIST database.

and prototypes generated by the best run among the twenty performed are displayed. The figures show how the devised approach is able to finding out the minimum number of prototypes (five, as the actual number of classes) able to cover all the present clusters. Only for the data set shown in Figure 6.5 seven prototypes have been found. But in this case, the crossing of two clusters does not allow to further reducing the number of prototypes.

6.2.2 NIST Database

A second set of experiments has been performed on handwritten digits extracted from the National Institute of Standards and Technology (NIST) database. Two statistically independent sets of data, a training set and a test set, have been randomly extracted from this database. Each set is made of 5000 patterns (500 per class). A total number of 20 runs have been performed on this data sets.

Bitmaps provided by the NIST database describing handwritten digits have been converted into real–valued vectors, in order make feasible the application of the LVQ version of the framework described in Section 4.4. To this purpose, a well known statistical description method, the geometrical moments, directly computed from the character bit map has been adopted.

Table 6.5: Recognition rates obtained on the NIST data sets.

Data set	**Rec rate**
Train set	95.0%
Test set	94.9%

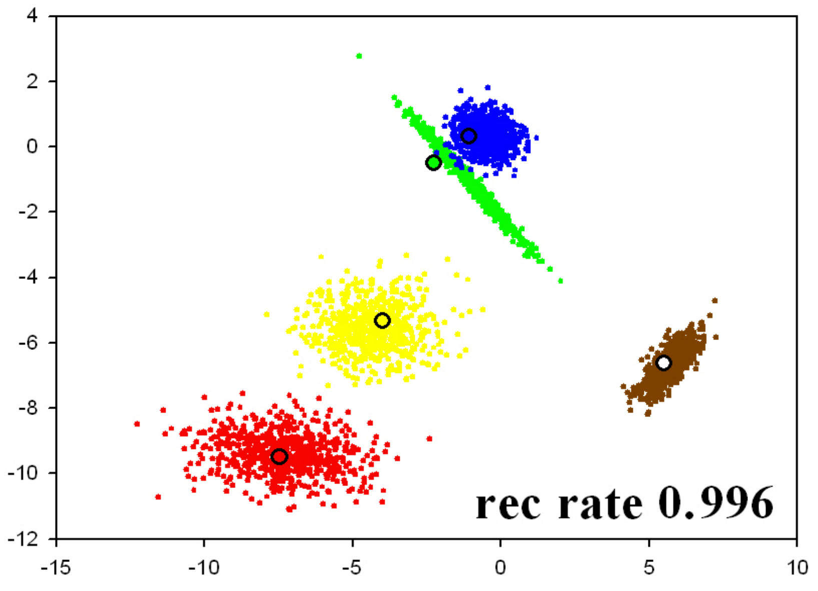


Figure 6.3: On this data set a recognition rate of 99.6% has been obtained by using 5 prototypes.

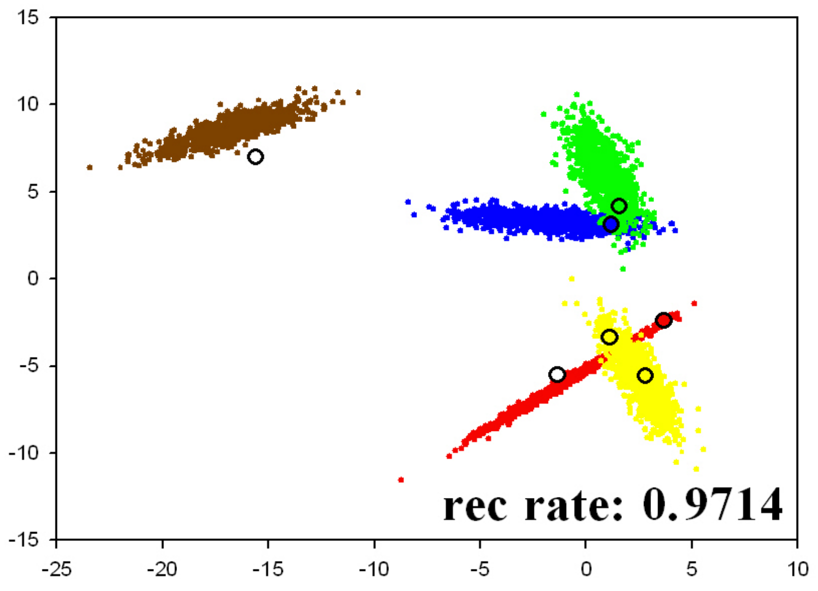


Figure 6.4: On this data set a recognition rate of 97.14% has been obtained by using 7 prototypes.

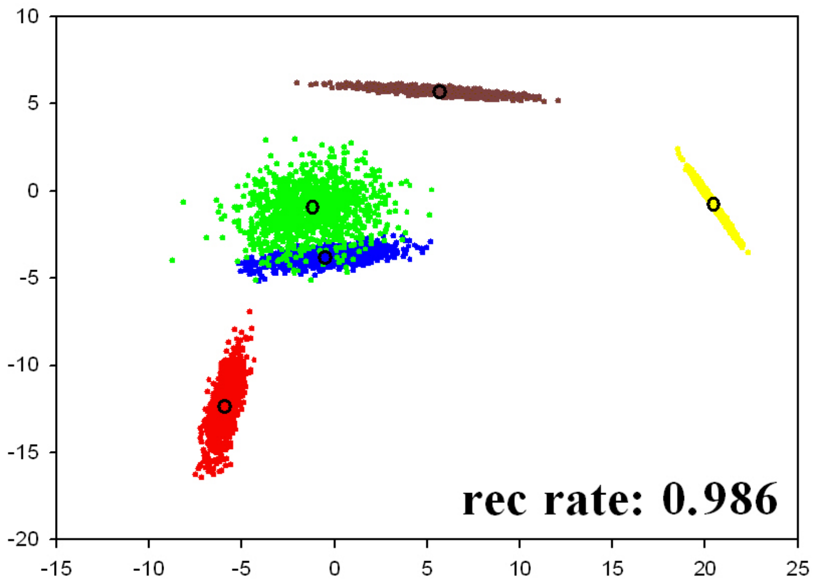


Figure 6.5: On this data set a recognition rate of 98.6% has been obtained by using 5 prototypes.

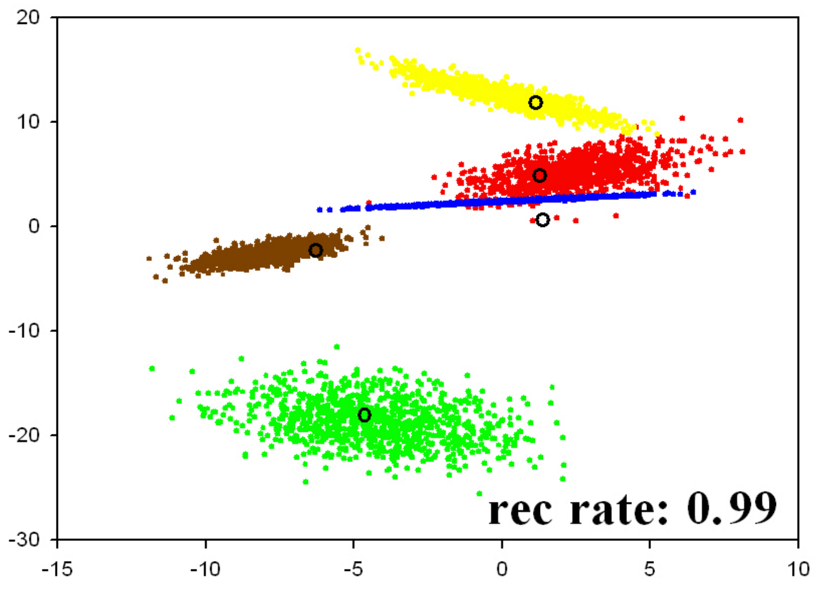


Figure 6.6: On this data set a recognition rate of 99.0% has been obtained by using 5 prototypes.

Table 6.6: Values of the basic evolutionary parameters used in the satellite image experiments.

Parameter	symbol	value
Population size	$\mathcal{P}$	300
Tournament size	$\mathcal{T}$	6
elithism size	$\mathcal{E}$	5
Crossover probability	p_c	0.4
Mutation probability	p_m	0.05
Number of Generations	N_g	500

The description of each input pattern is obtained by computing the central geometric moments until the 5th order: this implies a total number of moments equal to 33. Under this assumption, the elements making up a single prototype are real numbers belonging to the interval $[-1.0, 1.0]$. Therefore, each pattern is represented by a feature vector of 33 real elements. In this case, $N_{\min}$ and $N_{\max}$ have been set respectively to 10 (also in this case, the number of classes actually present) and 60. In this set of experiments, the best recognition rate obtained on the training set is 95%, whereas that obtained on the test set is 94.9%. The negligible difference between the above rates highlight the good generalization power of the system.

6.2.3 Satellite Images

A third set of experiments have been performed on data extracted from two 6 band multispectral images taken by the landsat satellite. The first one[2] is 2030x1167 pixels large and has been taken in order to distinguish between forest–non forest areas, while the second one[3] is a 1000x1000 pixels large, related to the land cover mapping for desertification studies. To this images a segmentation method has been applied in order to obtain regions formed by the same type of pixel [DPS03]. From each region provided by the segmentation algorithm implemented, a set of features has been extracted, related to its geometrical characteristics and to its spectral data. For each region the extracted features have been used to build up a data record. Figures 6.7 and 6.8 show the classification maps obtained from the images after the application of the segmentation process and the feature extraction.

[2]The image has been provided by courtesy from JRC.

[3]This image has been provided by courtesy from ACS spa as part of the Desert Watch project.

	NN	**9-NN**	**LVQ**	**EC-LVQ**
Mean	72.69	83.05	72.05	82.50
Std	4.34	0.25	3.36	1.7
$\mathbf{N_P}$	2250	2250	80	10.6 (1.7)

Table 6.7: Means and standard deviations of recognition rate on the test set for the forest cover dataset.

	NN	**6-NN**	**LVQ** (700)	**EC-LVQ**
Best	69.2	74.08	76.47	75.6
$\mathbf{N_P}$	3800	3800	700	136

Table 6.8: The recognition rates on the test set obtained for the land cover dataset.

From the first image a data set made up of 2500 items, i.e. regions, have been derived; each item may belong to one of two classes, forest or non–forest. From the second image 7600 items have been extracted. The items may belong to 7 classes, representing various land cover types: various vegetation or water. Although the segmentation process used extracts more than 10 features for each of the regions identified, for both analyzed images only six features, concerning both geometrical and spectral characteristics, have been considered,.

The evolutionary parameters used in the experiments reported below are shown in Table 6.6. For each of the two data sets taken into account 20 runs have been performed. The reported results are those obtained using the individual having the highest fitness among those obtained during the 20 performed runs.

As already said in Section 6.1.4, in any learning process, when the maximum performance is achieved on the training set, generalization power, i.e. the ability of correctly classify unknown data, may significantly decrease. In order to investigate such aspect for the developed system, the recognition rates on training and test set have been taken into account for the land cover data set. In Figure 6.9 such recognition rates, evaluated every 50 generations, in the best run for the land cover data set, are displayed. From the figures can be observed that, in the experiments carried out, the recognition rate increases with the number of generations both for the training set and for the test set. The best recognition rates occur in both cases nearby generation

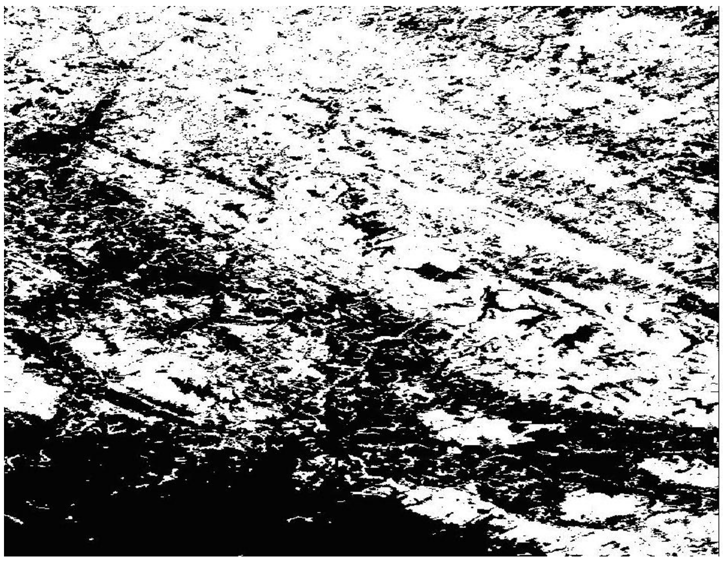


Figure 6.7: Classification map of the forest cover image obtained after the segmentation process. Black areas represent forest, whereas the white ones represent non-forest areas.

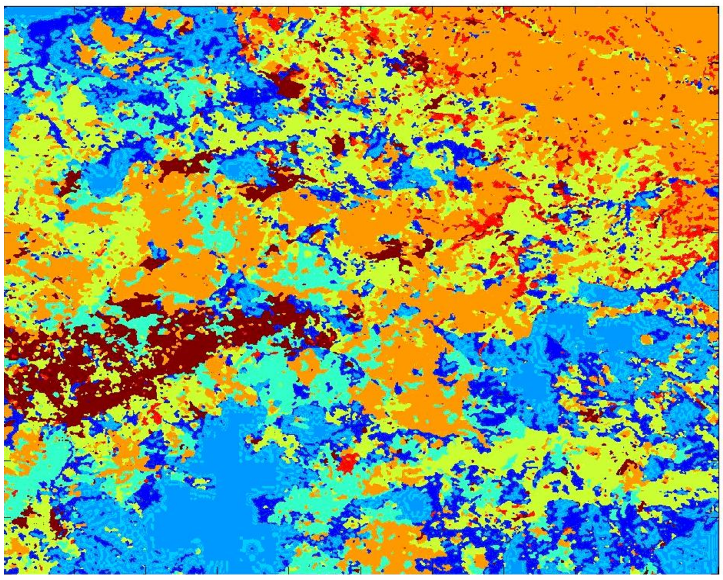


Figure 6.8: Classification map of the land cover image obtained after the segmentation process. Different colors represent represent different type of cover.

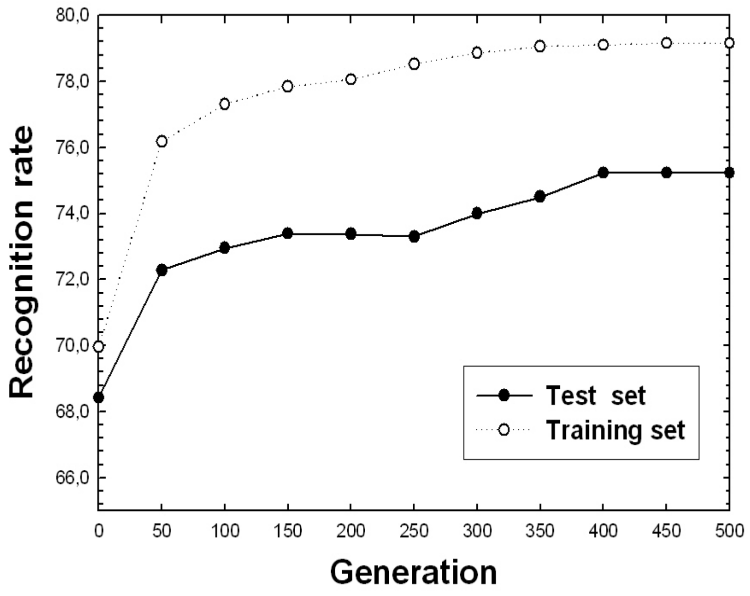


Figure 6.9: Recognition rate on training and test sets for the land cover data set during the best run.

400. Moreover, the fact that the difference between the two recognition rates does not tend to increase when that on the training set reaches its maximum, demonstrates the good generalization power of the system.

Comparison Findings

The results obtained by the proposed method on the data described above have been compared with those obtained by other three classification algorithms: nearest neighbor (NN), k–nearest–neighbor (k–NN) and the LVQ. First two algorithms have been described in Section 2.1 while the LVQ algorithm has been briefly described in Section 4.4. Particularly, the LVQ version used for comparing the obtained results is an improved one of that standard, called *Frequency Sensitive Competitive Learning* (FSCL) [AKCM90] and is often used to compare the performances of other classification algorithms.

In Table 6.7 the results obtained on the forest cover data set are shown. In all the runs performed on this data set the minimum length $N_{\min}$ allowed for an individual has been set to 2, while the maximum one has been set to 20. In order to avoid any bias in the comparison, due to the low number of patterns in the data set, the 10 fold cross validation procedure, already described in Section 6.1.4, has been used. As a consequence of the choice of

this procedure, 200 runs have been performed.
In Table 6.7, the mean and the standard deviation obtained on the 10 test sets together with the number of prototypes employed are reported. As regards the NN and the k–NN classifiers, the number of prototypes equals the number of patterns in the training set, while for the LVQ this number has been set to 80. For the EC–based LVQ method, this number is not provided by the user, but it has been automatically found by the system. Particularly, the average number of prototypes, found for the ten considered test set, equals to 10.6, while the standard deviation is 1.7. These figures demonstrate that the automatism devised allow the system to strongly reduce the number of prototypes needed to perform the classification task demanded. The outcome of this reduction is a strong enhancement of the classifier efficiency, as the number of Euclidean distances to be computed to recognize unknown patterns equals the number of prototypes used. As regards the recognition rate obtained on test set by the different algorithms, the performance of the developed system is significantly better than that of the NN and LVQ classifiers, although that of the k-NN (obtained setting k equal to 9) are slightly better than that of the implemented system. However, in my opinion, the huge difference in the number of necessary prototypes used compensates this little difference in the recognition rate.

In Table 6.8 results obtained on the land cover data set are shown. In this case the original data set has been randomly split in two sets, respectively as training set and test set. For this data set the total number of executed runs has been 20. In Table 6.8 the best results obtained and the number of prototypes employed are reported. The results of the framework–based system are significantly better than those obtained from the NN classifier and slightly better than that obtained from the k–NN (in this case k has been set equal to 6). Only the LVQ classifier has obtained a performance slightly better than ours, but such performance has been obtained with a total number of 700 prototypes, while the performance of the developed system has been obtained using only 136 prototypes. Also in this case, the difference in the number of prototypes compensates the little difference in the performance achieved.

6.3 Evolutionary Graph Generation

In this section, the experiments carried out on the system, described in Chapter 5, devised for evolving variable size graphs are illustrated. As already said in Section 5.6, in order to ascertain the effectiveness of the proposed method,

Parameter	symbol	value
Population size	P	1000
Tournament size	$\mathcal{T}$	60
elitism size	E	40
Crossover probability	p_c	0.3
Mutation probability	p_m	0.04
Number of Generations	N_g	500
Maximum number of nodes	N_{max}	50

Table 6.9: Values of the basic evolutionary parameters used in the experiments.

a hard non linear optimization problem has been chosen. The problem concerns the design of a wireless LAN providing wireless Internet services to users scattered in a given area. The wireless LAN is made of a set of access points (APs). APs are wired and connected to an Internet gateway. The optimization problem consists in determining the optimal layout of the AP's in the area to cover. Moreover, in order to reduce the total cost of the LAN, a configuration with minimal number of AP's and minimum length of the wires connecting them is considered optimal. In order to assess the quality of the solutions obtained by the proposed approach, another EC–based approach that generates graphs, called *EvoGraph* [HG04], has been taken into account. The approach is based on the GP paradigm and generates graphs by evolving individuals that encode a series of some predefined operations to be applied to an initial graph, the *embryo*, consisting of a single node. The results achieved by EvoGraph in solving the problem described in Section 5.6 have been illustrated in [HG04]. These results has been compared with those obtained by the developed system in solving the same problem. In the particular instance solved in [HG04] the area to covered equals to 1000x1000 in unit length, while both clients and APs are represented by points having integer coordinates. Then in order to obtain a fair comparison the same instance has been solved.

6.3.1 Parameter Settings

The values of the evolutionary parameters, used in all the performed experiments, are summarized in Table 6.9. The maximum number of nodes (N_{max}) allowed for an individual, has been a priori set[4].

[4]Also in this case individuals do not satisfying the constraint are ignored in the selection phase, i.e. killed

Other parameters reported in Table 6.2 have been set accomplishing the steps described in Section 6.1.3. The sets of values considered for each of the parameters tuned is listed below:

$$\begin{aligned}
P &= \{400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000\} \\
\mathcal{T} &= \{4, 5, \mathbf{6}, 7, 8, 9, 10\} \\
E &= \{1, 2, 3, \mathbf{4}, 5, 6, 7, 8, 9, 10\} \\
p_c &= \{0.1, 0.2, 0.3, 0.4, \mathbf{0.5}, 0.6, 0.7, 0.8.0.9\} \\
p_m &= \{0.01, 0.02, 0.03, 0.04, \mathbf{0.05}, 0.06, 0.07, 0.08, 0.09, 0.1\} \\
N_G &= \{100, 300, \mathbf{500}, 800, 1000\}
\end{aligned}$$

These sets have been determined by performing a set of preliminary trials. Default values are reported in bold-type. Also in this case, for each of the values to be tested 10 runs have been executed. Parameters have been tuned in the order reported in Table 6.9.

6.3.2 Results

Once the parameters have been set, a set of experiments have been performed in order to investigate the behavior of the implemented system when the number of clients to be covered varies. To this aim, the number of clients has been varied starting from 25 up to 50 with increments equal to 5. For each considered value, a distribution of clients has been randomly generated, and 50 runs have been performed with different initialization of the population. At the end of each run, the best solution found by the algorithm has been stored. The corresponding length of the wires connecting the AP's has been computed and stored as well. For each distribution of clients, we have computed the mean $\overline{N}_{AP}$ and the standard deviation $\sigma_{N_{AP}}$ of the number of AP's found by the proposed method while performing 50 runs (see Figure 6.10(a)). The mean $\overline{L}$ and the standard deviation σ_L of the lengths of the wires have been computed as well (see Figure 6.11(b)).

In order to investigate the ability of the system to find near- optimal connection topologies, for each solution provided by the system, the Minimum Spanning Tree[5] (MST) [GY01], representing the connection topology with minimal wire cost, of the corresponding graph have separately been computed. Then the lengths of the wire topology evolved by the system have been compared with the optimal ones represented by the MST lengths. This comparison, performed for each of the distributions of clients considered, represents a good assessment of the quality of the connection topology found

[5]A formal definition of minimum spanning tree is given in Appendix.

by the system. The comparison has been performed by computing the mean $\overline{L}$ of the length of the connection topologies of the solutions found by the system and the mean $\overline{L'}$, the length of the MST connection topologies separately computed for the same solutions. Both means have been computed on the best solutions found by the 50 runs executed. The plot of $\overline{L'}$ and $\overline{L'}$ as a function of the number of clients, is shown in Figure 6.11.

The results achieved can be considered satisfactory as in every run a complete coverage of the clients has been provided and the wire costs are very close to those computed on the MST. Moreover, the number of AP's needed to solve the problem, as well as the lengths of their connections, slightly increase with the number of clients. Finally, the standard deviations of both the number of AP's and the wire lengths assume small values, thus indicating that the solutions are widely independent of the initial conditions. Note that, for each distribution of clients, the evolutionary algorithm converges to solutions having almost the same number of AP's and the same wire cost.

Figures 6.12–6.15 illustrate some results of one of the experiments performed by using a randomly generated distribution of 40 clients. In particular, the best solutions obtained at generation 10, 100, 300 and 500 are shown. During the initial generations, the evolutionary process tends to improve the client covering by adding more and more AP's, without optimizing the connection topology (Figure 6.12). Only after an almost complete coverage has been obtained, the system tries to reduce the number of AP's and focuses the search on optimizing the connection topology. This behavior can be explained considering that the term F_{cover} in the fitness function has the highest weight, while the term F_{wires} the lowest. For instance, at generation 10 (Figure 6.12), all clients but one are covered using 23 AP's, but the connection topology is messy. At generation 100 (Figure 6.13), all the clients are covered with 21 AP's and the connection topology is significantly improved. At generation 300 (Figure 6.14) the total coverage is obtained with 20 clients and the connection topology is nearly optimal. At generation 500 (Figure 6.15), finally, 19 AP's are used and the topology connection is optimal (i.e., it coincides with the MST of the related graph).

In comparing the obtained results with those reported in [HG04] can be observed that the implemented system performs a global optimization in that both the number of AP's and their connections are simultaneously exploited for computing the fitness function. On the contrary, the genetic programming based method, is able to find a solution only when a sequential approach is adopted: first, solving the coverage problem by GP and then using a MST algorithm for determining the connection topology. Thus, it succeeds only when problem specific knowledge can be exploited to reformulate the original global optimization problem as a sequence of partial optimization problems.

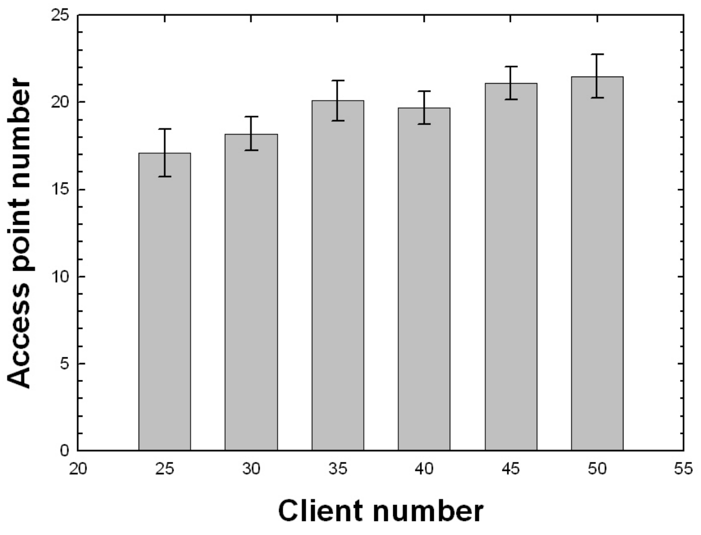


Figure 6.10: The mean number of access points and its standard deviation as a function of the number of clients are respectively represented by bars and segments on top of the bars.

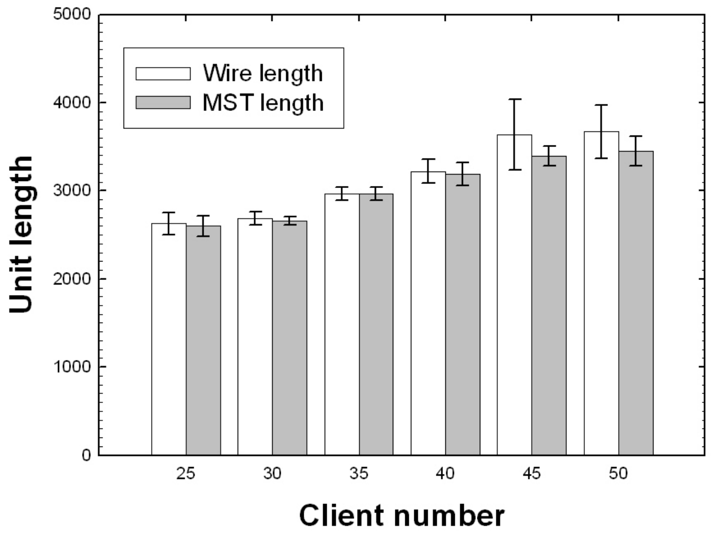


Figure 6.11: The mean of the wire lengths and its standard deviation as a function of the number of clients, found by the sytem and by using the MST for finding the connection topology.

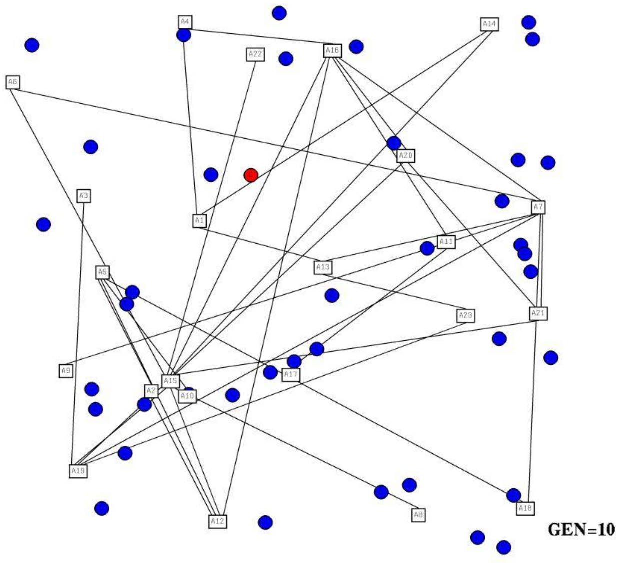


Figure 6.12: The best solution obtained after 10 generations. All clients but one (in red) are covered. This solution employs 23 Access points. The connection topology is messy.

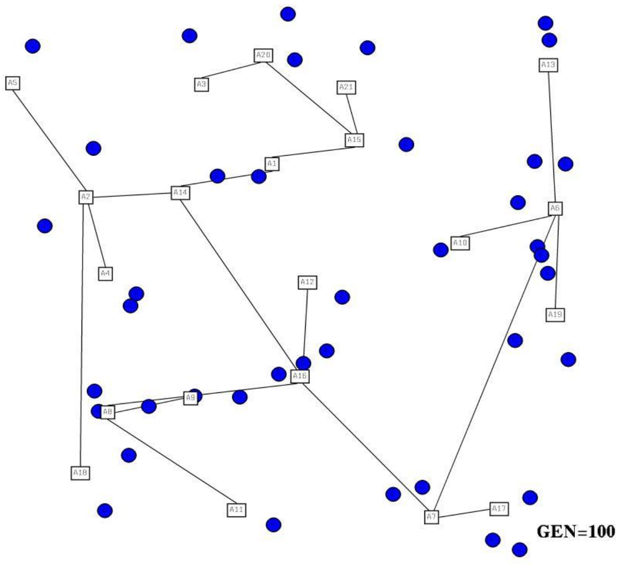


Figure 6.13: The best solution obtained after 100 generations. All clients are covered. This solution employs 21 Access points. The connection topology is suboptimal but no more messy.

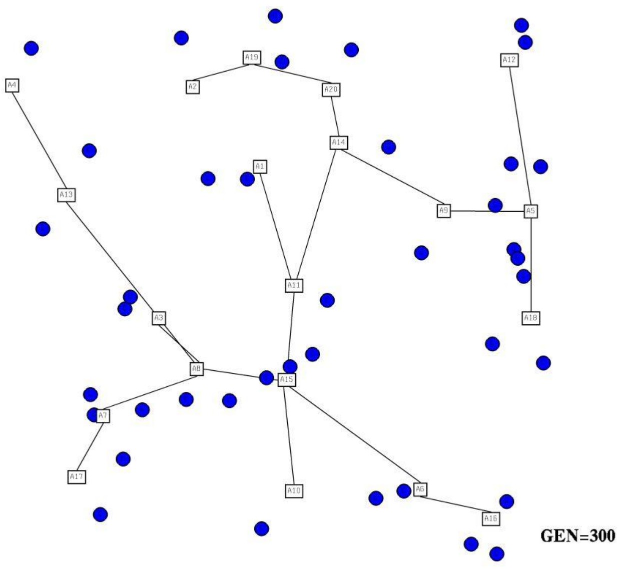


Figure 6.14: The best solution obtained after 500 generations. All clients are covered. This solution employs only 20 Access points. The connection topology is near to the optimal one.

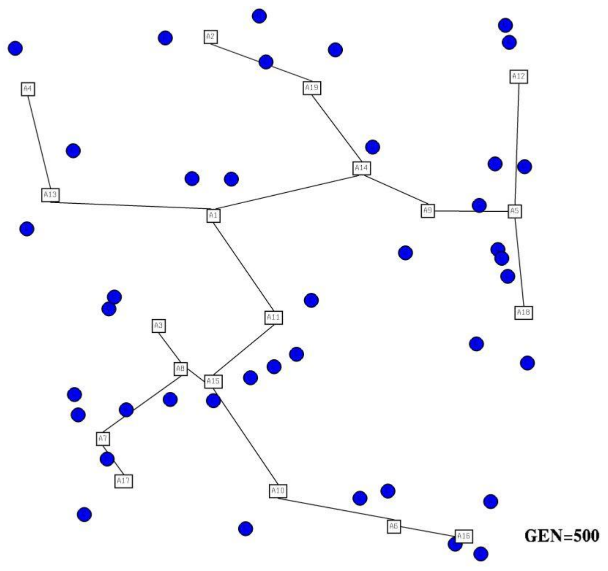


Figure 6.15: The best solution obtained after 500 generations. All clients are covered. This solution employs only 18 Access points while the connection topology is the optimal one.

Chapter 7

Conclusions and Future Work

Evolutionary Computation (EC) has been inspired by the natural phenomena of evolution. It provides a quite general heuristic, widely used in the last years to effectively solve hard, non linear and very complex problems. In the EC field four main branches can be distinguished: genetic algorithms, genetic programming, evolutionary strategies and evolutionary programming; each using a particular encoding for solutions. Nonetheless, many variants have been proposed since the beginnings of the field and others, most likely, will be proposed in the future. These variants, usually developed to solve specific problems, use the basic concepts provided by EC: reproduction of individuals, variation phenomena that affect the likelihood of survival of individuals and inheritance of parents features by offspring.

EC–based algorithms have also been used to tackle classification problems. The main aspect of classification usually concerns the generation of prototypes to be used to recognize unknown patterns. The role of prototypes is that of representing patterns belonging to the different classes defined within a given problem. For most of the problems of practical interest, the generation of such prototypes is a very hard problem, since a prototype must be able to represent patterns belonging to the same class, which may be significantly dissimilar each other. On the other hand, they must be able to discriminate between patterns belonging to classes different from the one that they represent, namely they must not contain any information that could lead to assign to them patterns belonging to other classes. Moreover, a prototype should contain the minimum amount of information required to satisfy the requirements mentioned above.

The research presented in this thesis, has been mainly devoted to the definition of an EC–based framework, to be used for prototype generation. The defined framework does not provide for the use of any particular kind of prototypes. In fact, it can generate any kind of prototype once an encoding

scheme for the used prototypes and the corresponding mutation operator have been defined. The generality of the framework can be exploited to develop many applications.
Moreover, in this thesis also a new method for graph generation has been presented. The devised method is able to generate variable size graphs. The purpose of the research leading to the definition of the method was that of developing a system that generates graphs to be used as prototypes. But in the thesis only preliminary tests on a hard non linear optimization problem are reported.

In the following, the contributions of the research presented in the thesis are reported in Section 7.1.2, while the conclusions are illustrated in Section 7.2. Finally, is Section 7.3.1 a number of ideas that could lead to future researches in the field are discussed.

7.1 Contributions

The work presented in this thesis has made the following contributions.

7.1.1 A Framework for Prototype Generation

In Chapter 4 a new EC–based framework able to generate prototypes for classification problems is illustrated. The proposed framework is quite general and can be applied to any classifier once a way for encoding the used prototypes is given. Moreover, any application based on the framework is able to automatically find the minimum number of prototypes needed to represent patterns belonging to the different classes defined. This remarkable ability does not require any specific knowledge of the classification problem to be solved, but only a training set containing samples of the patterns to be recognized. Such ability derives from two key choices made: (i) the encoding of *all* the prototypes searched in a *single* individual; (ii) prototypes within an individual are not a priori labeled.

Two specific applications based on the framework have been developed. The first application generates prototypes made of logical expressions, which represent patterns described as feature vectors. A prototype describes patterns by means of predicates expressing properties of the feature values of the patterns. Unknown patterns are recognized by assigning them to the prototype whose predicates satisfies the values in the feature vector describing the pattern.
The second application, instead, generates set of real-valued vectors to be used as prototypes of patterns represented as points in a feature space. In

this case unknown patterns are classified by first computing the Euclidean distance between the pattern and each of the prototypes and then assigning the pattern to the closest prototype.

7.1.2 Evolving Graphs

Chapter 5 illustrates a new EC-based approach, devised in order to generate variable size graphs. The approach uses a new defined and problem independent data structure, called *multilist*, employed to encode undirected attributed relational graphs. The defined encoding scheme, uses a positional notation that allows a sort of invariance property for some of the subgraphs encoded within a multilist. This property has made easier the definition of the recombination operator, allowing to overcome the main problems that affect the definition of this operator in the graph domain.

For this data structure two operators have also been defined: a recombination one, called *crossover*, that swaps subgraphs between two graphs and gives the approach the ability to evolve graphs of variable size. This operator has been defined in a quite simple way and does not perform any search on the input graphs, differently from several other approaches. In fact, thanks to the data structure defined, the split of an encoded graph is simply obtained by cutting a list, while the choice of how to divide the graph is made by choosing how to cut the list. In the same way, multilists allows the merging of two graphs by simply concatenating two lists.
The second operator, called *mutation*, changes the input graph into a new one without changing the number of nodes. By performing the same kind of operation, arcs can be easily added or deleted. In the same way both node and link attributes can be modified.

7.2 Conclusions

The evolutionary computation paradigm was chosen as the base of the research work presented in this thesis. This choice has been motivated by the fact that many of the approaches based on this paradigm have provided satisfactory solutions for hard, high dimensional and complex problems, often hardly addressed by other approaches. The research presented here has exploited the main feature of the EC, namely its generality. In fact, EC does not provide for a specific problem or application, but supplies a powerful search tool that can be used to find solutions for problems in a wide range of domains.

EC generality has been exploited to solve two specific problems: prototype generation for classification problems and graph generation. For the former problem a framework for evolving set of prototypes has been defined. The framework, exploiting the generality provided by EC, can be used to develop many applications, each using different kind of prototypes. The problem of graph generation has faced by defining a new data structure. In the following the conclusions related to the two approaches will be discussed.

7.2.1 Framework for Prototype Generation

Classification problems have been already faced in the past by using EC–based algorithms (see Section 4.1). In the learning classifier systems (LCS), for example, sets of rules are evolved. The rules distinguish patterns belonging to different classes by specifying the values of the pattern features. LCSs solve multi–class problem by evolving multiple populations, each searching for the rules characterizing patterns of a specific class. This approach present some difficulties, as it is possible that an instance is matched by several rules, each predicting a different class, or it is also possible that an instance is matched by none of any rule predicting any class.
Also GP–based methods have been devised to cope with classification problems. These methods, usually, evolve mathematical expressions, involving arithmetic functions and simple operators, that discriminate patterns belonging to different classes. In all the approaches just mentioned, the number c of classes to be dealt with is used to divide the data set at hand in exactly c parts. Thus, these approaches do not take into account the existence of subclasses within one or more of the classes in the analyzed data sets.
In the EC field, classification problems have also been faced in the EC field considering them as multimodal problems, in which prototypes are seen as solutions to be searched by using a single population. Within the unique population used, these algorithms try to evolve different groups of individuals, each searching for a different prototype. The effective use of the approach is limited by the amount of knowledge required about the fitness landscape, while for most of the real world problems such knowledge is hardly available.

The framework for prototype generation described in Chapter 4 offers several advantages that allows it to overcome the drawbacks that affect most of the EC–based approaches described above. These advantages are listed below:

– The framework has been naturally defined to cope with multi–class problems. Thus it is non necessary to separately evolve prototypes for different classes, as they are evolved together within each single individual. I this way, any framework–based system can model the interaction that occurs among the prototypes belonging to different classes when a pattern have to be classified. Modeling this interaction may, in many cases, significantly improve the performance of the system, as the behavior of a prototype, when a pattern has to be classified, may be sensitive to the presence of the prototypes representing the other classes.

- The framework does not need any a priori knowledge about the number of prototypes, but it is able to automatically find out the needed number of prototypes. This feature is a remarkable one as for many problems it is hard to expect this number.
- The only knowledge required by the framework is that represented by a training set, providing samples of the patterns to be recognized. Thus, the framework defined does not require any addition of knowledge about the problem to discover the number of prototypes involved in the classification task to be performed.

The effectiveness of the proposed framework to generate prototypes has been tested by performing a set of experiments on the two applications based on it. As mentioned above the first application generates prototypes consisting of logical expressions. In this case prototypes are encoded as derivation trees, as in the CFG-GP approach illustrated in Section 3.5. The second application, instead, generates real–valued vectors, to be used as prototypes in the feature space.
Both applications have been tested on several data sets, and the obtained results have been compared with those of other methods previously proposed in the literature. The experiments performed have demonstrated the ability of the applications developed to obtain good performances. Moreover, some of these experiments have shown the ability of the applications to find the minimum number of needed prototypes. Finally, for both applications, the comparisons performed have confirmed the quality of the performances achieved.

7.2.2 Evolving Graphs

Graphs are a data structure whose representative power have aroused an increasing interest in various fields of science and engineering (see Section 5.1). In fact, they may be used both to represent physical networks or to model the interactions occurring in complex systems, e.g. computer programs. Moreover, graphs are also used in pattern recognition and machine vision fields to

represent complex patterns in terms of parts and their relations.
The problem of generating graphs exhibiting the desired properties has also been faced by using EC–based techniques. Most of the proposed approaches use graph encoding methods specifically tailored for the faced problem. Moreover, the definition of a recombination operator in the graph domain present several problems, among the others:

– The splitting of a graph into subgraphs usually requires the choice of a variable number of crossover points, i.e. arcs to be broken;

– The merge of two subgraphs produced by the splitting of a graph usually require the reattachment of more than one arc;

The consequence of these problems is that in most of the approaches previously defined recombination operators are implemented by performing a search of the arcs to be broken. The search may make the recombination operator inefficient.

In Chapter 5 a new EC–based approach for graph generation is proposed. The approach is based on a new data structure, called *multilist*, specifically devised to encode graphs. Multilists have been defined without taking into account any specific problem. Moreover, multilists have some properties that solve most of the problem related to the definition of a recombination operator in the graph domain. The properties are listed below:

– A single graph can be split into two ones by choosing a single point in a list. Thus, no search has to be performed on the graph to be split;

– The reconnection of the suspended arcs in the subgraphs to be merged is automatically defined by the merge operation. This automatism exclude any search on the subgraphs to be merged;

These properties have allowed to easily define a recombination operator able to generate variable size multilists. Consequently, given a suitable fitness function, the devised approach can automatically determine the number of nodes for the desired graph.

The purpose of the research leading to the definition of the method was that of generating graphs to be used as prototypes. But, in order to ascertain the effectiveness of the method in solving complex problems, like may be that of generating graphs generating prototypes, it has been preliminarily tested on a hard non linear problem concerning the design of a wireless LAN providing wireless Internet services to users scattered in a given area. The wireless LAN is made of a set of access points (APs) anf the optimization problem consists in determining the optimal layout of the APs.

The obtained results in the performed experiments (see Section 6.3) have also been compared with those reported in the literature by another EC–based method. The obtained results and the comparison performed have demonstrated the effectiveness of the proposed approach.

7.3 Future Work

In this Section, a series of directions that should be investigated in order to improve the proposed systems are illustrated.

7.3.1 Framework for Prototype Generation

The recombination operator defined within the framework for prototype generation acts in a completely stochastic way, without exploiting any information about the behavior of the involved prototypes in the assignment of the training patterns. Moreover, the probability of applying the mutation operator is a priori fixed. In the following, proposals for a heuristic recombination operator and for a criterion to calculate the probability of applying the mutation operator will be illustrated.

Recombination

A heuristic recombination operator that builds up new individuals by considering the similarities among the involved prototypes could be defined. To this purpose, a similarity distance among prototypes that takes into account the patterns assigned to each of the prototypes should be used. This distance should be defined in such a way that, given two prototypes, the higher the number of different patterns assigned to the prototypes, the higher the value of the distance between the prototypes. The role of this distance measure would be that of maximizing the diversity, meant as diversity among the represented prototypes, among the prototypes used to build up the output individuals. This operator, would build up new individuals in a more effective way, because the new individuals built up will better represent the patterns contained in the training set.

Note that the heuristic operator employing the distance mentioned above could be defined within the framework, without taking into account the kind of prototypes used.

Mutation

The performances of any framework–based system could be easily improved by providing a way to calculate, for any single prototype, the probability of applying mutation operator that takes into account the performances obtained by the single prototype. In practice, this probability should be calculated as a function of the performance of the single prototype. Specifically, the lower the recognition rate obtained by the prototype, the higher the probability of applying the mutation to it. In this way, the research of prototypes becomes more effective, since the probability of modifying good prototypes is much lower than that of modifying bad prototypes, i.e. those performing worse in recognizing patterns belonging to the same class.

The calculation of the probability of applying the mutation operator can be defined without considering the kind of prototype used. Thus, this improvement can be defined within the framework and applied to any framework–based system developed.

7.3.2 Evolving Graphs

As mentioned in Chapter 5 the aim of the work leading to the definition of the EC–based for graph generation illustrated there, was that of building up a system able to generate graphs to be used as prototypes. In the experiments reported in this thesis the proposed method have preliminarily tested on a hard non linear optimization problem, implying the design of a wireless LAN. In the near future, the approach will be employed to develop a system for prototype generation, in which prototypes are represented as graphs. To this purpose different options will be investigated:

– The framework for prototype generation could be used. In this case, prototypes would be encoded by using multilists. Moreover, in order to evolve variable size prototypes, i.e. prototypes consisting of a different number of nodes, a macro–mutation operator that modifies multilists should be defined. This operator should be able to modify the number of nodes in the input graph.

– A multi population approach could be used, in which in each population prototypes representing different classes are evolved. In this case, the approach defined in Chapter 5 could be used, including the defined operators.

A longer term research in this field will imply the study of graph grammars. The purpose of this research will be that of employing these grammars to generate graphs. Using grammars will strongly reduce the search space represented by the solutions graphs. This reduction should lead to a significantly improvement of the performances.

Appendix A

Formal Definition of Data Classification

In the data classification context a set of objects to be analyzed is called *data set*, and each object is called *sample* and represented by $\mathbf{X} = (x_1, \ldots, x_\ell)$ with $\mathbf{X} \in \mathbf{S}$, where $\mathbf{S}$ is the universe of all possible elements characterized by ℓ features and x_i denotes the i–th feature of the sample. A data set with cardinality N_D is denoted by $\mathcal{D} = \{\mathbf{X}_1, \ldots, \mathbf{X}_{N_D}\}$ with $\mathcal{D} \subseteq \mathbf{S}$. The set $\mathcal{D}$ is said *labeled* if it exists a set of integers:

$$\Lambda = \{\lambda_1, \ldots, \lambda_{N_D}\} : \lambda_i \in [1, c]$$

The i–th element λ_i of Λ is said the *label* of the i–th sample $\mathbf{X}_i$ of $\mathcal{D}$. We will say that the samples of $\mathcal{D}$ can be grouped into c different classes. Moreover, given the sample $\mathbf{X}_i$ and the label $\lambda_i = j$, we will say that $\mathbf{X}_i$ belongs to the j–th class.

Given a data set $\mathcal{D} = \{\mathbf{X}_1, \ldots, \mathbf{X}_{N_D}\}$ containing c classes, a classifier Γ is defined as a function

$$\Gamma : \mathcal{D} \longrightarrow [0, c]$$

In other words, a classifier assigns a label $\gamma_i \in [0, c]$ to each input sample $\mathbf{X}_i$. If $\gamma_i = 0$, the corresponding sample $\mathbf{X}_i$ is said *rejected.* This fact means that the classifier is unable to trace the sample back to any class.

The sample $\mathbf{X}_i$ is *recognized* by Γ if and only if:

$$\gamma_i = \lambda_i$$

otherwise the sample is said *misclassified.* If N_{corr} is the number of samples of $\mathcal{D}$ recognized by Γ the ratio N_{corr}/N_D is defined as the *recognition rate* of the classifier Γ obtained on the data set $\mathcal{D}$.

Appendix B

In the following formal definitions of the main data structure used in this thesis are given.

Graph

An *undirected* or simply *graph* G is defined as $G = (A_n, E)$, where $A_n = \{n_1, n_2, \ldots, n_N\}$ is a set of *nodes* and $E = \{\langle n_i, n_j\rangle \,|n_i, n_j \in A_n\}$ is a symmetric binary relation defined on the set A_n. The couples in E are said *arcs* and denotated by symbols set $\{l_{ij}|1 \leq i, j \leq N\}$. A graph G is defined with *attributes* if exists a set, finite or infinite, $S_a = \{a_1, a_2, \ldots, a_{N_a}\}$, called *attributes set*, and a function A:

$$A : \{n_1, n_2, \ldots, n_N\} \rightarrow S_a$$

which binds to each node of the graph an attribute (an element of the set A_a). Moreover, a graph with attributes is said *relational* if exists a set, finite or infinite, $S_r = \{r_1, r_2, \ldots, r_{N_R}\}$, called *relations set* and a function R:

$$R : \{l_{ij}|1 \leq i, j \leq N\} \rightarrow S_r$$

The function R binds to each arc of the graph a *relation*, that is an element of the set S_r.

Definitions on Graphs

The number of nodes of G defines its *order* and is denotated by the symbol $|G|$; the number of arcs, instead, by the symbol $\|G\|$. Two nodes which belong to the same couple in E are said *adiacents* and *extremes* of the related arc. The arc $\langle n_i, n_j\rangle$ is said *incident* both to the node n_i and n_j. Yet, we will say that the arc $\langle n_i, n_j\rangle$ *links* the node n_i to the node n_j. An arc which has equal extremes is said *loop*.
The set of arcs incident to a node is said its *incidence set* and denotated by

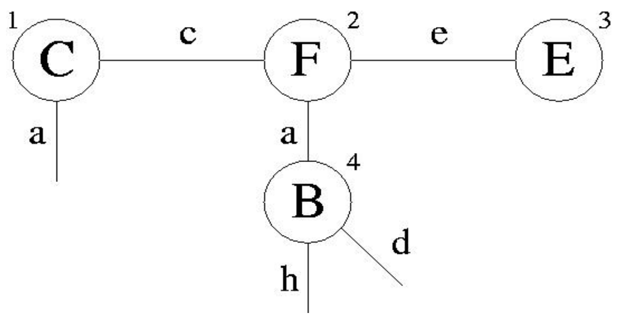


Figure 7.1: An example of cut graph.

the symbol $E(n)$ (formally: $E(n_i) = \{l_{jk} \mid (j = i) \vee (k = i)\}$). A *subgraph* of G is a graph $G' = (I'_{N'}, E')$ such that $I'_{N'} \subseteq I_N$ ed $E' \subseteq \{\langle n_i, n_j\rangle | n_i, n_j \in I'_{N'} \wedge \langle n_i, n_j\rangle \in E\}$. A *path from* n_1 *to* n_k is defined as a sequence $P = n_1, n_2, \ldots, n_k$. The number of nodes in P is said *length* of the path. If $n_1 = n_k$ the path is said *cycle*.
A graph G is *complete* if for each couple of nodes exists an arc which links them (formally: $\forall n_i, n_j \in A_n, \quad \langle n_i, j\rangle \in E$). Moreover, a graph is *connected* if, for each couple of nodes in G exists a path between them. A graph without loops and arcs which connect the same couple of arcs is said *simple*. In the following, a simple graph as that just described will be said *canonical*, so as to distinguish it from the next defined graph.

Cut Graph

A *cut* graph G is defined as $G = (A_n, E)$, where $I_N = \{n_1, n_2, \ldots, n_N\}$ is a set of *nodes* and E is a simmetric binary relation defined on the set $I_N \bigcup \{\nu\}$, where ν is said *null* node. The couples in E are said arc and denotated by the set of symbols $\{l_{ij} | 0 \leq i, j \leq N\}$. The couples which belong to the set $I_S = \{\langle n_i, \nu\rangle | 1 \leq i \leq N\} \subseteq E$ are said *suspended* arcs and denotated by the set of symbols $\{l_{oi} | 1 \leq i \leq N\}$.
If we remember the subgraph definition previously done, we observe that a subgraph $G' = (A'_{N'}, E')$ of a graph $G = (I_N, E)$, whether canonical or cut, is cut if $(\langle n_i, \nu\rangle \in E') \wedge (n_i \in I'_{N'})$.

Minimum Spanning Tree

Given a connected, undirected graph G, a *spanning tree* of G is a subgraph which is a tree and connects all the nodes together. Note that a single graph can have many different spanning trees. If G is a weighted graph, to each spanning tree T can be assigned a weight W_T, that equals the sum of the weights of the arcs in T. The *Minimum Spanning Tree* (MST) T' of a wighted graph G is defined as the spanning tree of G with weight less or equal to the weight of every other spanning tree in G.

Lists

A list is defined as a set of objects placed in a predefined order:

$$e_1, e_2, e_3, e_4, e_5, e_6, e_7, \ldots$$

The object e_i of the list is said *element* i-th of the list. The cardinality of the list is said *length* of the list[1] The list of length equal to 0, by definition, is said *void list*.
Let $I = \{i_1, i_2, \ldots, i_{N_I}\}$ a set of cardinality N_I, the list $l = e_1, e_2, e_3, e_4, \ldots e_n$ is said *in* I if $e_i \in I \quad \forall i \in [1, n]$. The set I is said *definition set* of the list and denotated by the symbol $I(l)$[2]
Two lists $l^{'} = \{e_1^{'}, e_2^{'}, \ldots, e_N^{'}\}$ and $l^{''} = \{e_1^{''}, e_2^{''}, \ldots, e_N^{''}\}$ both of length N, are equal ($l^{'} = l^{''}$) if $e_i^{'} = e_i^{''} \quad \forall i \in [1, n]$. Given two lists $l^{'} = \{e_1^{'}, e_2^{'}, \ldots, e_N^{'}\}$ and $l^{''} = \{e_1^{''}, e_2^{''}, \ldots, e_N^{''}\}$ of length equal respectively to $N^{'}$ and $N^{''}$ ($N^{'} < N^{''}$), is said that the list $l^{'}$ is *included* in the list $l^{''}$ ($l^{'} \subset l^{''}$) if $\exists\, i \quad : \ 1 \leq i \leq (N^{''} - N^{'}) : \ e_k^{'} = e_k^{''} \quad \forall k \in [i, (i + N^{'})]$.

Some Operations on Lists

The operation *n-shortening* applied to the list $l = \{e_1, e_2, \ldots, e_N\}$ of length N, generates the list $l^{'} = \{e_1, e_2, \ldots, e_{(N-n)}\}$ of length [3]$(N - n)$.
The operation *t-cut*, applied to the list $l = \{e_1, e_2, \ldots, e_N\}$ of length N generates the two lists $l^{'} = \{e_1, e_2, \ldots, e_t\}$ and $l^{''} = \{e_{t+1}, e_{t+2}, \ldots, e_{(N)}\}$, of length respectively equal to t and $(N - t)$.
The operation $i - j$*-swap* applied to the list $l = \{e_1, e_2, e_i \ldots, e_j, \ldots, e_N\}$ generates the list [4]$l^{'} = \{e_1, e_2, e_j \ldots, e_i, \ldots, e_N\}$.
Given two lists $l^{'} = \{e_1^{'}, e_2^{'}, \ldots, e_{N'}^{'}\}$ and $l^{''} = \{e_1^{''}, e_2^{''}, \ldots, e_{N''}^{''}\}$ of length

[1]In the following we use the function *length*, which associate to the generic list l of n element the natural number n, it follows that *length*$(l) = n$.

[2]This concept is useful because defines the set of values which the objects in a list can take on.

[3]Such operation consists of the elimination of the last n elements from the list.

[4]Pratically, such operation swaps the places of the elements i and j of the list.

respectively equal to $N^{'}$ and $N^{''}$, the *concatenate* operation of the list $l^{''}$ to the list $l^{'}$ generates the list

$$l = \{e^{'}_1, e^{'}_2, \ldots, e^{'}_{N^{'}}, e^{''}_1, e^{''}_2, \ldots, e^{''}_{N^{''}}\}$$

of length $(N^{'} + N^{''})$.
Given two lists:

- $l^{'} = \{e^{'}_1, e^{'}_2, \ldots, e^{'}_i, \ldots, e^{'}_{N^{'}}\}$
- $l^{''} = \{e^{''}_1, e^{''}_2, \ldots, e^{''}_j, \ldots, e^{''}_{N^{''}}\}$

of length respectively equal to $N^{'}$ and $N^{''}$, the operation *i-j-swap* applied to the two lists $l^{'}$ $l^{''}$ generates the lists[5]:

- $l^{*} = \{e^{'}_1, e^{'}_2, \ldots, e^{''}_j, \ldots, e^{'}_{N^{'}}\}$
- $l^{**} = \{e^{''}_1, e^{''}_2, \ldots, e^{'}_i, \ldots, e^{''}_{N^{''}}\}$

Given two lists:

- $l^{'} = \{e^{'}_1, e^{'}_2, \ldots, e^{'}_i, \ldots, e^{'}_{(i+n)}, \ldots, e^{'}_{N^{'}}\}$
- $l^{''} = \{e^{''}_1, e^{''}_2, \ldots, e^{''}_j, \ldots, e^{''}_{(j+n)}, \ldots, e^{''}_{N^{''}}\}$

of length respectively equal to $N^{'}$ and $N^{''}$, the operation *n-i-j-swaps* applied to the two lists $l^{'}$ and $l^{''}$ generates the lists[6]:

- $l^{*} = \{e^{'}_1, e^{'}_2, \ldots, e^{''}_j, \ldots, e^{''}_{(j+n)}, \ldots, e^{'}_{N^{'}}\}$
- $l^{''} = \{e^{''}_1, e^{''}_2, \ldots, e^{'}_i, \ldots, e^{'}_{(i+n)}, \ldots, e^{''}_{N^{''}}\}$

Given two lists $l^{'} = \{e^{'}_1, e^{'}_2, \ldots, e^{'}_{N^{'}}\}$ and $l^{''} = \{e^{''}_1, e^{''}_2, \ldots, e^{''}_{N^{''}}\}$ of length respectively equal to $N^{'}$ and $N^{''}$, the operation *p-inserimento* of $l^{''}$ in $l^{'}$ generates the list

$$l = \{e^{'}_1, e^{'}_2, \ldots, e^{'}_p, e^{''}_1, e^{''}_2, \ldots, e^{''}_{N^{''}}, e^{'}_{(p+1)}, e^{'}_{(p+2)}, \ldots, e^{'}_{N^{'}}\}$$

of length $(N^{'} + N^{''})$. Th value of p is said *inserting point* of $l^{''}$ in $l^{'}$.

[5]Pratically, such operation swaps the element i of the list $l^{'}$ and the element j of the lists $l^{''}$.

[6]Such operation is simply the application to the lists $l^{'}$ and $l^{''}$ of the following n operations: i-j-swap, $(i+1)$-$(j+1)$-swap, …, $(i+n)$-$(j+n)$-swap.

Bibliography

[ABL02] D. Agnelli, A. Bollini, and L. Lombardi. Image classification: an evolutionary approach. *Pattern Recognition Letters*, 23(1-3):303–309, 2002.

[ACS01] C. Arcelli, L.P. Cordella, and G. Sanniti di Baja, editors. *Visual Form 2001, LNCS 2059*. Springer-Verlag, 2001.

[AKCM90] S.C. Ahalt, A.K. Krishnamurthy, P. Chen, and D.E. Melton. Competitive learning algorithms for vector quantizationn. *Neural Networks*, 3(10):277–290, October 1990.

[And35] E. Anderson. The irises of the gaspe peninsula. *Bull. Amer. IRIS Soc.*, 59:2–5, 1935.

[BB01] M. Brameier and W. Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26, 2001.

[BM98] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998.

[BT96] Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394, 1996.

[Bul80] M.G. Bulmer. *The Mathematical Theory of Quantitative Genetics*. Clarendon Press, Oxford, 1980.

[CAH$^+$02] Dingjun Chen, Takafumi Aoki, Naofumi Homma, Toshiki Terasaki, and Tatsuo Higuchi. Graph-based evolutionary design of arithmetic circuits. *IEEE Trans. Evolutionary Computation*, 6(1):86–100, 2002.

[Cas01] E. Cascetta. *Transportation systems engineering: theory and methods*. Kluwer Academic, 2001.

[CF01] Kumar Chellapilla and David B. Fogel. Evolving an expert checkers playing program without using human expertise. *IEEE Trans. Evolutionary Computation*, 5(4):422–428, 2001.

[CFSV02] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Learning structural shape descriptions from examples. *Pattern Recognition Letters*, 23:1427–1437, 2002.

[Cho56] Noam Chomski. Three models for the description of language. *IEEE Transactions on Information Theory*, 2(3):113–124, 1956.

[CLH02] B.-C. Chien, J. Lin, and T.-P. Hong. Learning discriminant functions with fuzzy attributes for classification using genetic programming. *Expert Systems with Applications*, 23:31–37, 2002.

[Cro03] M. Crow. *Computational Methods for Electric Power Systems.* CRC Press, 2003.

[CV00] L. P. Cordella and M. Vento. Symbol recognition in documents: A collection of techniques. *International Journal on Document Analysis and Recognition (IJDAR)*, 3(2):73–78, 2000.

[Dar59] Charles Darwin. *On the the Origin of Species by Means of Natural Selection.* John Murray, 1859.

[DDAS03] D.Maio, D.Maltoni, A.K.Jain, and S.Prabhakar. *Handbook of A Fingerprint Classification.* Springer Verlag, 2003.

[DDT02] I. De Falco, A. Della Cioppa, and E. Tarantino. Discovering interesting classification rules with genetic programming. *Applied Soft Computing*, 1(4):257–269, 2002.

[DG89] Kalyanmoy Deb and David E. Goldberg. An investigation of niche and species formation in genetic function optimization. In *Proceedings of the third international conference on Genetic algorithms*, pages 42–50, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[DHS01] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification.* John Wiley & sons, Inc., 2001.

[DJ95] C. Dorai and A.K. Jain. Shape spectra based view grouping for free-form objects. In *Proceedings of the International Conference on Image Processing*, pages 240–243, 1995.

[DPS03] C. D'Elia, G. Poggi, and G. Scarpa. A tree-structured random markov field model for bayesian image segmentation. *IEEE Transactions on Image Processing*, 12(10):1259–1273, October 2003.

[DR95] D.A.Reynolds and R.C.Rose. Robust text-independent speaker identification using gaussian mixture speaker models. *Communications of the ACM*, 3(1):72–83, 1995.

[DSGJ93] K. A. De Jong, W. M. Spears, D. F. C. Gordon, and Z. Janikow. Using genetic algorithms for concept learning. *Machine Learning*, (13):161–188, 1993.

[EF84] M. A. Eshera and K. S. Fu. A graph distance measure between attributed relational graphs for image analysis. In *Proceedings of 7th Int. Conf. on Pattern Recognition*, pages 75–77. IEEE Press, May 1984.

[Fal98] E. Falkenauer. *Genetic Algorithms and Grouping Problems.* John Wiley and Sons, New York, 1998.

[FDN59] R. M. Friedberg, B. Dunham, and J. H. North. A learning machine: Part II. *IBM Journal of Research and Development*, 3(3):282–287, 1959.

[FGK95] A. Filatov, A. Gitis, and I. Kil. Graph-based handwritten digit string recognition. In *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 2)*, page 845. IEEE Computer Society, 1995.

[Fog99] D.B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence.* IEEE Press, Piscataway, NJ, 2nd edition, 1999.

[FOW66] I. J. Fogel, A. J. Owen, and M. J. Walsh. *Artificial Intelligence Through Simulated Evolution.* John Wiley and Sons, New York, 1966.

[Fra57] A. Fraser. Simulation of genetic systems by automatic digital computers. *Australian Journal of Biological Science*, 10:484–491, 1957.

[Fri59] R. M. Friedberg. A learning machine: Part I. *IBM Journal of Research and Development*, 3(3):282–287, 1959.

[GB89] John J. Grefenstette and James E. Baker. How genetic algorithms work: A critical look at implicit parallelism. In *ICGA*, pages 20–27, 1989.

[GLW98] Al Globus, John Lawtonb, and Todd Wipkeb. Automatic molecular design using evolutionary techniques. In Al Globus and Deepak Srivastava, editors, *The Sixth Foresight Conference on Molecular Nanotechnology*, Westin Hotel in Santa Clara, CA, USA, 1998.

[GN95] A. Giordana and F. Neri. Search-intensive concept induction. *Evolutionary Computation*, 3(4):375–416, winter 1995.

[Gol66] David Goldberg. Genetic and evolutionary algorithms come of age. *Communications of the ACM*, 37(3):113–119, March 1966.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., 1989.

[GR87] David E. Goldberg and Jon Richardson. Genetic algorithms with sharing for multimodal function optimization. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 41–49, Mahwah, NJ, USA, 1987. Lawrence Erlbaum Associates, Inc.

[GS93] D. P. Greene and S. F. Smith. Competition-based induction of decision models from examples. *Machine Learning*, (13):229–257, 1993.

[GY01] J.L. Gross and J. Yellen. *Graph Theory and Its Application.* McGrawHill, 2001.

[Hay94] Simon Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1994.

[Hek97] J. Hekanaho. Ga-based rule enhancement in concept learning. In *Proceedings of the third International Conference on Knowledge Discovery and Data*, page 183186, Newport Beach, CA, 1997.

[HG96] Jeffrey Horn and David E. Goldberg. Natural niching for evolving cooperative classifiers. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming: Proc. of the First Anuual Conf. 1996*, pages 553–564, Cambridge, MA, 1996. The MIT Press.

[HG04] Jianjun Hu and Erik Goodman. Wireless access point configuration by genetic programming. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 1178–1184, Portland, Oregon, jun 2004. IEEE Press.

[HGD94] Jeffrey Horn, David E. Goldberg, and Kalyanmoy Deb. Implicit niching in a learning classifier system: Nature's way. *Evolutionary Computation*, 2(1):37–66, 1994.

[Hol92] J.H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, second edition, 1992.

[Hol00] John H. Holland. Building blocks, cohort genetic algorithms, and hyperplane-defined functions. *Evolutionary Computation*, 8(4):373–391, 2000.

[Hor97] Jeffrey Horn. *GAs (with sharing) in search, optimization and machine learning.* Morgan Kaufmann, San Mateo, CA, 1997.

[Hur02] S. Hurley. Planning effective cellular mobile radio networks. *IEEE Transactions on Vehicular Technology*, 51:243–253, March 2002.

[Jan93] C. Z. Janikow. A knowledge-intensive genetic algorithm for supervised learning. *Machine Learning*, (13):189–228, 1993.

[KLL98] K.Lieska, E. Laitinen, and J. Lahteenmaki. Radio coverage optimization with genetic algorithms. In *Proceedings of PIMRC*, volume 1, pages 318–322, Sept 1998.

[Koh82] T. Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(5):59–69, 1982.

[Koh89] T. Kohonen. *Self-Organization and Associative Memory.* Springer-Verlag, Berlin, Heidelberg, 1989.

[Koh01] Teuvo Kohonen. *Self-Organizing Maps.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 3-rd edition, 2001.

[Koz92] J.R. Koza. *Genetic Programming: on the programming of computers by means of natural selection.* MIT Press, 1992.

[KPMA00] J. K. Kishore, L. M. Patnaik, V. Mani, and V. K. Agrawal. Application of genetic programming for multicategory pattern classification. *IEEE Transactions on Evolutionary Computation*, 4(3):242–258, sep 2000.

[KY03] E. Koichi and W. Yoishinori. Automatic cell design for wide area wireless lan systems. *Special Issue on Devices and Systems for Mobile Communications*, 44(4), 2003.

[LC98] Jason D. Lohn and Silvano P. Colombano. Automated analog circuit synthesis using a linear representation. *Lecture Notes in Computer Science*, 1478:125+, 1998.

[LK00] C.Y Lee and H.G. Kang. Cell planning with capacity expansion in mobile communications a tabu search approach. In *IEEE VTC2000*, pages 1678–1691, Sept 2000.

[LLS00] Tjen-Sien Lim, Wei-Yin Loh, and Yu-Shan Shih. A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms. *Machine Learning*, 40(3):203–228, 2000.

[LP02] Bill Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer, Berlin, 2002.

[LSW00] P. L. Lanzi, W. Stolzmann, and S. W. Wilson, editors. *Learning Classifier Systems: From Foundations to Applications*, volume 1813 of *Lecture Notes on Artificial Intelligence*. Springer-Verlag, Berlin, Germany, 2000.

[LW66] E.L. Lawler and D.E. Wood. Branch and bound methods: A survey. *Operations Research*, 14:699–719, 1966.

[Man94] V. Maniezzo. Genetic evolution of the topology and weight distribution of neural networks. *IEEE Transactions on Neural Networks*, 5(1):39–53, 1994.

[MHT$^+$01] C. Miyajima, Y. Hattori, K. Tokuda, T. Masuko, T. Kobayashi, and T. Kitamura. Speaker identification using gaussian mixture models based on multi-space probability distribution. In *Proceedings of International Conference on Acoustic Speech and Signal Processing*, pages 433–436, 2001.

[Mit97] T.M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[MJ95] M.N. Murty and A.K. Jain. Knowledge-based clustering scheme for collection management and retrieval of library books. *Pattern Recognition*, 28(5):949–964, 1995.

[MKS94] Sreerama K. Murthy, Simon Kasif, and Steven Salzberg. A system for induction of oblique decision trees. *Journal of Artificial Intelligence Research*, 2:1–32, 1994.

[Mon93a] David J. Montana. Strongly typed genetic programming. Technical Report #7866, 10 Moulton Street, Cambridge, MA 02138, USA, 7 1993.

[Mon93b] David J. Montana. Strongly typed genetic programming. Technical Report #7866, 10 Moulton Street, Cambridge, MA 02138, USA, 7 1993.

[MPD04] Durga Prasad Muni, Nikhil R. Pal, and Jyotirmoy Das. A novel approach to design classifiers using genetic programming. *IEEE Trans. Evolutionary Computation*, 8(2):183–196, 2004.

[MSV93] H. Muhlenbein and D. Schlierkamp-Voosen. The science of breeding and its application to the breeder genetic algorithm (bga). *Evolutionary Computation*, 1(4):335–360, 1993.

[MSV95] Heinz Mühlenbein and Dirk Schlierkamp-Voosen. Analysis of selection, mutation and recombination in genetic algorithms. In *Evolution and Biocomputation*, pages 142–168, 1995.

[MVFN01] RRF Mendes, FB Voznika, AA Freitas, and JC Nievola. Discovering fuzzy classification rules with genetic programming and co-evolution. In *Principles of Data Mining and Knowledge Discovery (Proc. 5th European Conference PKDD 2001) - Lecture Notes in Artificial Intelligence*, 2168, pages 314–325, Berlin, 2001. Springer-Verlag.

[NY91] Ryohei Nakano and Takeshi Yamada. Conventional genetic algorithm for job shop problems. In *ICGA*, pages 474–479, 1991.

[Och05] Gabriela Ochoa. Error thresholds in genetic algorithms. *Evolutionary Computation*, 2005. in press.

[PGP97] M. Pei, E. D. Goodman, and W. F. Punch. Pattern discovery from data using genetic algorithms. In *Proceedings of the first Pacific-Asia Knowledge & Data Mining (PAKDD-97)*, February 1997.

[PL97] Riccardo Poli and W. B. Langdon. Genetic programming with one-point crossover and point mutation. Technical Report CSRP-97-13, Birmingham, B15 2TT, UK, 15 1997.

[PM03a] Riccardo Poli and Nicholas Freitag McPhee. General schema theory for genetic programming with subtree-swapping crossover: part i. *Evolutionary Computation*, 11(1):53–66, 2003.

[PM03b] Riccardo Poli and Nicholas Freitag McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part ii. *Evolutionary Computation*, 11(2):169–206, 2003.

[PSZ98] Marcello Pelillo, Kaleem Siddiqi, and Steven W. Zucker. Matching hierarchical structures using association graphs. *Lecture Notes in Computer Science*, 1407:3–13, 1998.

[Qui93] J. Ross Quinlan. *C4.5: programs for machine learning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[Ras92] E. Rasmussen. Prentice-Hall, Upper Saddle, 1992.

[RCO98] Conor Ryan, J. J. Collins, and Michael O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *EuroGP '98: Proceedings of the First European Workshop on Genetic Programming*, pages 83–96, London, UK, 1998. Springer-Verlag.

[RDC00] Patrick J. Rauss, Jason M. Daida, and Shahbaz A. Chaudhary. Classification of spectral image using genetic programming. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 726–733, 2000.

[Rec73] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Fromman-Holzboog Verlag, Stuttgart, 1973.

[RK82] A. Rosenfeld and A.C. Kak. *Digital Picture Processing.* Academic Press, New York, NY, 2nd edition, 1982.

[Ros94] Gerald P. Roston. *A Genetic Methodology for Configuration Design.* PhD thesis, Pittsburgh, PA 15213-3891, USA, 1994.

[SA98] S.Connell and A.K.Jain. Learning prototypes for online handwritten digits. In *Proceedings of the 14th International Conference on Pattern Recognition*, pages 182–184, Brisbane, Australia,, 1998.

[Sha01] J. L. Shapiro. *Theoretical aspects of evolutionary computing*, chapter Statistical mechanics theory of genetic algorithms, pages 87–108. Natural Computing. Springer-Verlag, London, UK, 2001.

[SM98] R. Silipo and C. Marchesi. Artificial neural networks for automatic ecg analysis. *IEEE Transactions on Signal Processing*, 46(5):1417–1425, 1998.

[Sta02] P.F. Stadler. *Fitness landscape*, volume 585 of *Lecture Notes Physics*, pages 187–207. Springer-Verlag, Heidelberg, 2002.

[Tal83] J.L. Talmon. Pattern recognition of the ECG. Technical report, Akademisch Proefscrift, Berlin, Germany, 1983.

[Whi89] D. Whitely. The genitor algorithm and selection pressure: Why rank–based allocation of reproductive trials is best. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121, San Mateo, California, 1989. Morgan Kaufmann Publishers.

[Whi96] Peter Alexander Whigham. *Grammatical Bias for Evolutionary Learning*. PhD thesis, School of Computer Science, University College, University of New South Wales, Australian Force Academy, Canberra, Australia, October 1996.

[Wri32] S. Wright. The roles of mutation, inbreeding, crossbreeding and selection in evolution. In *Proceedings of the 6th International Congress on Genetics*, volume 1, pages 356–366, 1932.

[Wri98] M. H. Wright. Optimization method for base station placement in wireless applications. In *Proceedings of the 1998 IEEE Conference on Vehicular Technology*, pages 287–291. IEEE Press, May 1998.

[XY99] X.Yao and Y.Liu. Neural networks for breast cancer diagnosis. In *Proceedings of Congress on Evolutionary Computation*, pages 1760–1767, 1999.

[Yao99] Xin Yao. Evolving artificial neural networks. *PIEEE: Proceedings of the IEEE*, 87, 1999.

[YL97] X. Yao and Y. Liu. A new evolutionary system for evolving artificial neural networks. *IEEE Transactions on Neural Networks*, 8(3):694–713, May 1997.

[ZQC96] B. Zheng, W. Qian, and L.P. Clarke. Digital mammography: Mixed feature neural network with spectral entropy decision for detection of micro-calcifications. *IEEE Transactions on Medical Imaging*, 15(5):595–560, 1996.

[ZSQ96] L. Zhang, R. Sankar, and W. Qian. An improved shift-invariant artificial neural network for computerized detection of clustered microcalcifications in digital mammograms. *Medical Physichs*, 33(4):595–560, 1996.

[ZSQ02] L. Zhang, R. Sankar, and W. Qian. Advances in micro-calcification clusters detection in mammography. *Computers in Biology and Medicine*, 32(6), November 2002.