# Prototype Learning with Attributed Relational Graphs

Pasquale Foggia, Roberto Genna, and Mario Vento

Dipartimento di Informatica e Sistemistica, Università di Napoli "Federico II",
Via Claudio, 21 I-80125 Napoli (Italy)
{foggiapa,genna,vento}@unina.it

**Abstract.** An algorithm for learning structural patterns given in terms of Attributed Relational Graphs (ARG's) is presented. The algorithm, based on inductive learning methodologies, produces general and coherent prototypes in terms of Generalized Attributed Relational Graphs (GARG's), which can be easily interpreted and manipulated. The learning process is defined in terms of inference operations especially devised for ARG's, as graph generalization and graph specialization, making so possible the reduction of both the computational cost and the memory requirement of the learning process. Experimental results are presented and discussed with reference to a structural method for recognizing characters extracted from ETL database.

## 1 Introduction

Structured patterns are patterns represented in terms of simple parts, often called primitives, and relations among them [1]. They are generally represented by means of Attributed Relational Graphs (ARG's), e.g. associating the nodes and the edges respectively to the primitives and to the relations among them. If necessary, their properties are represented by attributes both of the nodes and of the edges.

Despite their attractiveness in terms of representational power, structural methods (i.e. methods dealing with structured information) imply complex procedures both in the recognition and in the learning process. In fact, in real applications the information is affected by distortions, and consequently the corresponding graphs result to be very different from the ideal ones. So, in the recognition stage the comparison among the input sample and a set of prototype graphs cannot be performed by exact graph matching procedures [2]. Moreover the learning problem, i.e. the task of building a set of prototypes adequately describing the objects of each class, is complicated by the fact that the prototypes, implicitly or explicitly, should include a model of the possible distortions. For these reasons nowadays the problem is still under investigation: many of the approaches proposed during the last years consider this task as a symbolic machine learning problem, introducing description languages often more general and complex than needed [3,4,5,6,7,8,9].

The advantage making this approach really effective relies in the obtained descriptions: since the learning method is oriented toward the construction of maximally general prototypes, they are expressed as compact predicates, easily understandable by human beings. The user can acquire knowledge about the domain by looking at them and consequently he can validate or even improve the prototypes

F.J. Ferri et al. (Eds.): SSPR&SPR 2000, LNCS 1876, pp. 447-456, 2000.

or at least understand what has gone wrong in case of classification errors. On the other hand these methodologies are so computationally heavy, both in terms of time and memory requirements, that only simple applications can be actually dealt with.

Our approach is similar to these methods, but has the peculiarity that descriptions are given in terms of Attributed Relational Graphs. The use of a somewhat less powerful description language is compensated by the ability to express the operations of our learning method directly in terms of graph operations, with a significant improvement in the computational requirements of the system. To this aim, we will introduce a new kind of ARG, called Generalized Attributed Relational Graph, devoted to represent in a compact way the features common to a set of ARG's. Then, in section 3 we will formulate a learning algorithm operating directly in the graphs' space: it finds out prototypes that are both general and consistent, like classical machine learning ones. Section 4 reports an experimental analysis of the method with reference to a problem of character recognition, using a standard character database.

## 2 Preliminary Definitions

An ARG can be defined as a 6-tuple $(N, E, A_N, A_E, \alpha_N, \alpha_E)$ where $N$ and $E \subset N \times N$ are respectively the sets of the nodes and of the edges of the ARG, $A_N$ and $A_E$ the sets of node and edge attributes and finally $\alpha_N$ and $\alpha_E$ the functions which associate to each node or edge of the graph the corresponding attribute.

We will assume that the attributes of a node or an edge are expressed in the form $t(p_1,\ldots,p_{k_t})$, where $t$ is a type chosen over a finite alphabet $T$ of possible types and $(p_1,\ldots,p_{k_t})$ are a tuple of parameters, also from finite sets $P_1^t,\ldots,P_{k_t}^t$. Both the number of parameters ($k_t$, the *arity* associated to type $t$) and the sets they belong to depend on the type of the attribute, so that we are able to differentiate the descriptions of different kinds of nodes (or edges), as explained in fig. 1.

Let us introduce the concept of Generalized Attributed Relational Graph (from now on GARG). Basically a GARG is an ARG with an extended attribute definition: the set of types of node and edge attribute is extended with the special type $\phi$, carrying no parameter and matching any attribute type, with no regard to the attribute parameters. For the other attribute types, if the sample has a parameter whose value is within the set $P_i^t$, the corresponding parameter of the prototype belongs to the set $P_i^{*t} = \wp(P_i^t)$, where $\wp(X)$ is the power set of $X$, i.e. the set of all the subsets of $X$.

We say that a GARG $G^* = (N^*, E^*, A_N^*, A_E^*, \alpha_N^*, \alpha_E^*)$ covers a sample $G$ ($G^* \models G$, where the symbol $\models$ denotes the relation from now on called *covering*) iff there is a mapping $\mu : N^* \to N$ such that:

1. $\mu$ is a *monomorphism*; that is:

$$n_1^* \neq n_2^* \Rightarrow \mu(n_1^*) \neq \mu(n_2^*) \;;\; \forall (n_1^*, n_2^*) \in E^*, (\mu(n_1^*), \mu(n_2^*)) \in E \tag{1}$$

2. the attributes of the nodes and of the edges of $G^*$ are compatible with the corresponding ones of $G$; that is:

$$\forall n^* \in N^*, \alpha_N^*(n^*) \succ \alpha_N(\mu(n^*)) \;;\; \forall (n_1^*, n_2^*) \in E^*, \alpha_E^*(n_1^*, n_2^*) \succ \alpha_E(\mu(n_1^*), \mu(n_2^*)) \quad (2)$$

where the symbol $\succ$ denotes a compatibility relation, defined as follows:

$$\forall t, \phi \succ t(p_1, \ldots, p_{k_t}) \;;\; \forall t, t(p_1^*, \ldots, p_{k_t}^*) \succ t(p_1, \ldots, p_{k_t}) \Leftrightarrow p_1 \in p_1^* \wedge \ldots \wedge p_{k_t} \in p_{k_t}^* \quad (3)$$

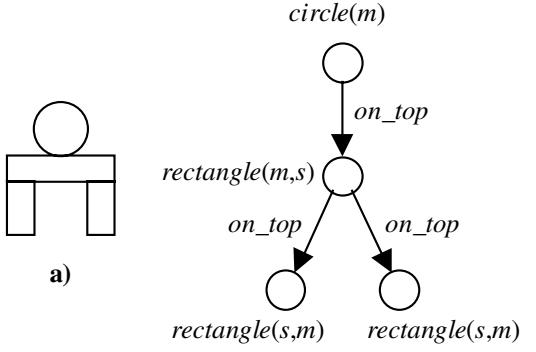



**Fig. 1. a)** An object made of two different kinds of primitives (circles and rectangles) and **b)** the corresponding graph. **c)** The type alphabets. The description scheme defines two types of nodes, each associated to a different primitive. Each type contains a set of parameters to suitably describe a component of that type (*s*, *m*, *l* stand for *small*, *medium*, *large*, respectively). Similarly edges of the graph describe topological relations among the primitives.

Condition (1) requires that each primitive and each relation in the prototype must be present also in the sample, while the converse condition does not hold; this allows the prototype to specify only the features which are strictly required for discriminating among the various classes, neglecting the irrelevant ones. Condition (2) constrains the monomorphism required by condition (1) to be consistent with the attributes of the prototype and of the sample: the compatibility relation defined in (3) simply states that the type of the attribute of the prototype must be either equal to $\phi$ or to the type of the corresponding attribute of the sample, in which case all the parameters of the attribute (that are actually sets of values) must contain the value of the corresponding parameter of the sample.

Another important relation that will be introduced is *specialization* (denoted by the symbol $\trianglelefteq$): a prototype $G_1^*$ is said to be a *specialization* of $G_2^*$ iff:

$$\forall G, \quad G_1^* \models G \Rightarrow G_2^* \models G \quad (4)$$

In other words, a prototype $G_1^*$ is a specialization of $G_2^*$ if every sample covered by $G_1^*$ is necessarily covered by $G_2^*$ too. Hence, a more specialized prototype imposes stricter requirements on the samples to be covered.

Notice that the specialization relation introduces a non total ordering in the prototype space, whose minimum (if we consider only non-empty graphs) is the GARG having only one node with attribute $\phi$: any non-empty GARG is a specialization of it.

## 3 The Learning Algorithm

The goal of the learning algorithm can be stated as follows: the algorithm is given a *training set* $S$ of labeled patterns, partitioned into $C$ different classes ( $S = S_1 \bigcup \ldots \bigcup S_C$ with $S_i \bigcap S_j = \varnothing$ for $i \neq j$), from which it tries to find a sequence of prototype graphs $G_1^*, G_2^*, \ldots, G_p^*$, each labeled with a class identifier, such that:

1. $\forall G \in S \quad \exists i : G_i^* \models G$ (*completeness* of prototype set) (5)

2. $\forall G \in S \quad G_i^* \models G \Rightarrow \text{class}(G) = \text{class}(G_i^*)$ (*consistency* of the prototype set) (6)

where class($G$) and class($G^*$) refer to the class associated with samples $G$ and $G^*$ respectively.

Equations (5) and (6) would be simply satisfied by defining a prototype for each sample in $S$. However, such a trivial solution requires a number of prototypes which could be too large for many applications; besides in a complex domain it is difficult to obtain a training set which covers exhaustively all the possible instances of a class. Hence, for eq. (5) the prototypes generated should be able to model also samples not found in $S$, that is they must be more *general* than the enumeration of the samples in the training set. However, they should not be too general otherwise eq. (6) will not be satisfied. The achievement of the optimal trade-off between completeness and consistency makes the prototypation a really hard problem.

To this concern, our definition of the *covering* relation, which allows the sample to have nodes and edges not present in the prototypes, is aimed at increasing the generality of the prototypes; in fact, each prototype must specify only the distinctive features of a class, i.e. the ones which allow the class' samples to be distinguished from those of other classes; optional features are left out from the prototype, and their presence or absence has no effect on the classification.

It's worth pointing out that in our definition of GARG's there is no possibility of expressing *negation*; this fact allows our method to employ a fast graph matching algorithm [2,10,11] suitable to verify whether a prototype covers a sample, instead of the usual unification algorithm. On the other hand, such a lack limits the expressiveness of the prototypes. In order to deal with situations in which the patterns of some class can be viewed as subpatterns of another class, a sample is compared sequentially against the prototypes in the same order in which they have been generated, and it is attributed to the class of the first prototype that covers it. One of the strength points of the proposed learning method is the automatic handling of these situations, by adopting a learning strategy which considers simultaneously all the classes that must be learned in order to determine (without hints from the user) the proper ordering for the prototypes.

A sketch of the algorithm is shown by the following code, where $S(G^*)$ denotes the sets of all the samples of the training set covered by a prototype $G^*$, and $S_i(G^*)$ the samples of the class *i* covered by $G^*$.

*A sketch of the learning procedure*

```
FUNCTION Learn(S) // Returns ordered list of prototypes
   L := [ ] // L is list of prototypes, initially empty
   WHILE S≠∅
      G* := FindPrototype(S)
      IF NOT Consistent(G*) THEN
         FAIL // Algorithm terminates unsuccessfully
      END IF
      // Assign prototype to the class most represented
      class(G*) := argmax_i |S_i(G*)|
      L := Append(L, G*) // Add G* to the end of L
      S := S-S(G*) // Remove the covered samples from S
   END WHILE
   RETURN L
END FUNCTION
```

It is worth pointing out that the test of consistency in the algorithm actually checks whether the prototype is almost consistent:

$$Consistent(G^*) \Leftrightarrow \max_i \frac{|S_i(G^*)|}{|S(G^*)|} \geq \theta \tag{7}$$

In eq. (7) $\theta$ is a threshold close to 1, used to adapt the tolerance of the algorithm to slight inconsistencies in order to have a reasonable behavior also on noisy training data. For example, with $\theta = 0.95$ the algorithm would consider consistent a prototype if at least the 95% of the covered training samples belong to a same class, avoiding a further specialization of this prototype that could be detrimental for its generality.

Note that the assignment of a prototype to a class is done *after* the prototype has been found, meaning that the prototype is not constructed in relation to an *a priori* determined class: the algorithm finds at each step the class which can be better covered by a prototype and generates a prototype for it. In this way, if the patterns of a class *i* can be viewed as subpattern of samples of another class *j* (e.g. the graphs describing the character 'F' are often subgraphs of those representing character 'E'), the algorithm will cover first the class *i* and then the class *j*; in this case, we say that the prototypes of the class *i* have precedence over those of class *j*.

The most important part of the algorithm is the *FindPrototype* procedure, that performs the construction of a prototype, starting from the trivial GARG (which covers any non-empty graph) and refining it by successive specializations until either it becomes consistent or it covers no samples at all. The *FindPrototype* algorithm is *greedy*, in the sense that at each step it chooses the specialization that seems to be the

best one, looking only at the current state without any form of look-ahead. This search is guided by the *heuristic function H*, which will be examined later.

*The function FindPrototype*

```
FUNCTION FindPrototype(S) // Finds the best prototype
                          // covering one class in S
   G* := TrivialPrototype // Only one node with attr. φ
   WHILE |S(G*)| > 0 AND NOT Consistent(G*)
      Q := Specialize(G*)
      G* := argmax_{X∈Q} H(S, X) // H is heuristic function
   END WHILE
   RETURN G*
END FUNCTION
```

### 3.1. The Heuristic Function

The heuristic function *H* is introduced for evaluating how promising a provisional prototype is. It is based on the estimation of the consistency and completeness of the prototype (see eq. 5 and 6):

$$H(S,G^*) = H_{\text{compl}}(S,G^*) \cdot H_{\text{cons}}(S,G^*) = \left|S(G^*)\right| \cdot (I(S) - I(S(G^*))) \tag{8}$$

where

$$I(S) = -\sum_i \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|} \quad \text{and} \quad I(S(G^*)) = -\sum_i \frac{\left|S_i(G^*)\right|}{\left|S(G^*)\right|} \log_2 \frac{\left|S_i(G^*)\right|}{\left|S(G^*)\right|} \tag{9}$$

In other words, to evaluate the consistency degree of a provisional prototype $G^*$, we have used the quantity of information (in bits) necessary to express the class a given element of $S(G^*)$ belongs to, i.e. $I(S(G^*))$; the completeness of $G^*$, instead, is taken into account by simply counting the number of samples covered by $G^*$, so preferring general prototypes versus more specialized ones.

### 3.2. The Specialization Operators

An important step of the *FindPrototype* procedure is the construction of a set $Q$ of specializations of the tentative prototype $G^*$. At each step, the algorithm tries to refine the current prototype definition, in order to make it more consistent, by replacing the tentative prototype with one of its specializations. To accomplish this task we have defined the following set of specialization operators which, given a prototype graph $G^*$, produce a new prototype $\overline{G^*}$ such that $\overline{G^*} \trianglelefteq G^*$:

1. NODE ADDITION: $G^*$ is augmented with a new node *n* whose attribute is $\phi$. This operator is always applicable.

2. EDGE ADDITION: a new edge $(n_1^*, n_2^*)$ is added to the edges of $G^*$, where $n_1^*$ and $n_2^*$ are nodes of $G^*$ and $G^*$ does not contain already an edge between them. The edge attribute is $\phi$. This operator is applicable if $G^*$ is not a *complete graph*.
3. ATTRIBUTE SPECIALIZATION: the attribute of a node or an edge is specialized according to the following rule:
   - If the attribute is $\phi$, then a type t is chosen and the attribute is replaced with $t(P_1^t, \ldots, P_{k_t}^t)$. This means that only the type is fixed, while the type parameters can match any value of the corresponding type.
   - Else, the attribute takes the form $t(p_1^*, \ldots, p_{k_t}^*)$, where each $p_i^*$ is a (non necessarily proper) subset of $P_i^t$. One of the $p_i^*$ such that $|p_i^*| > 1$ is replaced with $p_i^* - \{p_i\}$, where $p_i \in p_i^*$. In other words, one of the possible values of a parameters is excluded from the prototype.

Note that, except for the node addition, the specialization operators can be usually applied in several ways to a prototype graph; for example, the edge addition can be applied to different pairs of nodes. In these cases, it is intended that the function *Specialize* exploits all the possibilities.

*The function Specialize*

```
FUNCTION Specialize(G*) // Returns the set of the
direct
                        // specializations of G*
   Q := ∅
   FOREACH σ IN SpecializationOperators
      IF Applicable(σ, G*) THEN
         Q := Q∪Apply(σ, G*)
      END IF
   END FOREACH
   RETURN Q
END FUNCTION
```

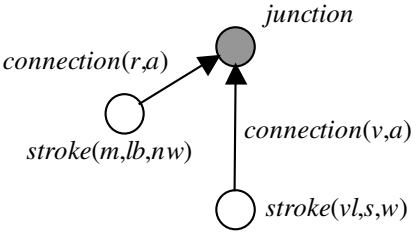
## 4. Application and Discussion

The method has been experimented on a character recognition problem, obtained by selecting from the ETL-1 Character Database [12] about 9000 random digits. We have partitioned the whole data set into a *training set* of 230 samples per class, and a separate *test set* of 680 samples per class. Each character of the database is represented by a 63×64 bitmap that we have described in terms of circular arcs by means of a preprocessing phase depicted in [13]. Fig. 2 illustrates the adopted description scheme; basically, we have defined two node types for representing our primitives (the circular arcs, here called *strokes*) and their junctions; the edges

represent the adjacency of a stroke to a junction. Node attributes encode the size of the strokes (normalized with respect to the size of the whole character), their shape (ranging from straight line segment to full circle) and their orientation; edge attributes represent the relative position of a junction with respect to the strokes it connects.

Our learning algorithm has generated in about 27 hours a list of 136 prototypes, consistent and complete with respect to the training set. While the required time seems to be quite high, it has to be considered that other first-order symbolic learning algorithms are generally unable at all to produce results on training sets of the considered size: for instance, Quinlan's FOIL [9] was not able to run on our training set due to memory limitations. Moreover the samples are highly noisy and no effort has been taken to polish the training set, as is usually done when working with symbolic machine learning methods.

**a)**




**b)**

NODE TYPE ALPHABET

$T = \{ stroke , junction \}$

$k_{stroke} = 3$

$k_{junction} = 0$

$p_1^{stroke}$ = size = $\{vs,s,m,l,vl\}$

$p_2^{stroke}$ = shape = $\{s,lb,b,hb,c\}$

$p_3^{stroke}$ = orientation = $\{n,nw,w,sw,s,se,e,ne\}$

**c)**

EDGE TYPE ALPHABET

$T = \{ connection \}$

$k_{connection} = 2$

$p_1^{connection}$ = $x$-projection = $\{l,v,r\}$

$p_2^{connection}$ = $y$-projection = $\{b,h,a\}$

**d)**

**Fig. 2.** An example of a sample of the database. **a)** Its representation in terms of strokes and junctions: notice that the junction has been highlighted for the sake of clarity. **b)** The corresponding graph (topologically arranged to make clear the matching between the nodes and the strokes/junctions), according to **c)-d)** the formal description of node and edge types. Nodes of the ARG are used for describing both the strokes (type *stroke*) and the connections among them (type *junction*). Nodes associated to strokes have three parameters: the size can be *very short*, *short*, *medium*, *long* or *very long*; the shape *straight*, *slightly bent*, *bent*, *highly bent* or *circular*; the orientation can be one of the 8 directions of the compass card. Junctions have no parameters. The edges of the graphs are used for describing the position of a junction with respect to a stroke, by means of the projections of the junction and of the stroke on $x$ and $y$ axes: the junction can be on the *left* or on the *right* of the stroke or else the stroke is approximately *horizontal*; similarly, the junction can be *below* or *above* the stroke, else the latter is *vertical*.

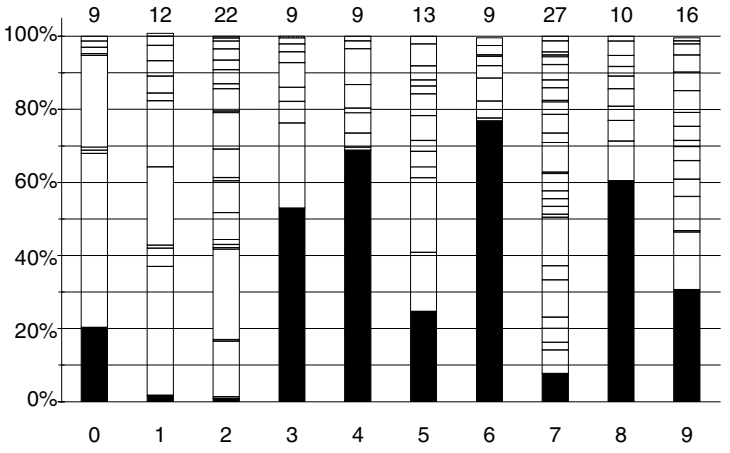



**Fig. 3.** The coverage of the 136 prototypes found on the training set. The sequence of prototypes within each class matches the order they are sequentially generated. Notice the capability of generalization of our system: starting from an average of 234.2 samples per class in the training set, it has found out only 13.6 prototypes in the average, with a compression ratio greater than 17:1.

The number of prototypes per class generated by our algorithm can be read in fig. 3, which shows also the coverage of each prototype: most of the classes are covered at 80% by using only a few number of prototypes per class. These prototypes have a high coverage and capture the major invariants of the character shapes inside a class. The remaining prototypes account for a few characters which, because of noise, are quite dissimilar from the average samples of their class.

**Table 1.** The misclassification matrices evaluated on the test set (null values are not printed): white columns report the results when no form of reject is introduced, while gray ones refer to the discard of all prototypes covering less than 1% of the training samples. The recognition rate of the classes are reported as (bold) values of main diagonal, while the value at row *i* and column *j* (*j*≠*i*) denotes the percentage of samples of class *i* erroneously attributed to class *j*. Last column reports the percentage of samples our system cannot assign to any class. Without rejecting, the overall recognition rate is 81.1%. When accepting some reject, misclassification decreases: e.g., though the recognition rate in the case of the class '2' is significantly lower, the number of samples erroneously classified as '2' has drastically decreased (see the column '2').

| | **0** | | **1** | | **2** | | **3** | | **4** | | **5** | | **6** | | **7** | | **8** | | **9** | | **R** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | **93** | **93** | .14 | .14 | 2.3 | | 1.1 | 1.1 | .14 | | | | .14 | | 2.7 | .57 | | | | | 4.9 |
| **1** | .98 | | **92** | **92** | 3.6 | .14 | .14 | .14 | | | | | .56 | | 2.5 | .56 | | | .14 | | 7.3 |
| **2** | .58 | .44 | 2.3 | 2.3 | **76** | **61** | 3.1 | 2.9 | .73 | .15 | 2.3 | 2.3 | .73 | | 11 | 8.3 | .44 | .44 | 2.8 | 2.2 | 20 |
| **3** | .14 | .14 | .14 | .14 | 7.2 | 1.4 | **89** | **88** | | | 1.3 | 1.3 | .28 | .28 | .56 | .42 | .42 | .42 | 1.4 | 1.4 | 6.3 |
| **4** | | | .14 | .14 | 14 | 1.7 | 2.2 | 2.2 | **74** | **74** | 3.3 | 3.3 | | | 1.7 | .58 | 1.0 | 1.0 | 3.2 | 3.2 | 14 |
| **5** | .28 | .28 | 0 | 0 | 19 | 1.4 | 11 | 10 | .57 | .57 | **65** | **65** | 2.4 | 2.4 | .71 | .57 | .43 | .43 | .71 | .71 | 18 |
| **6** | .70 | .70 | .42 | .42 | 11 | .14 | 3.8 | 3.4 | .28 | .28 | 1.5 | 1.5 | **79** | **79** | .98 | .28 | .84 | .84 | .84 | .84 | 12 |
| **7** | 1.4 | | 2.8 | 2.8 | 2.6 | .28 | .14 | .14 | | | | | 1.6 | | **91** | **78** | | | .28 | .28 | 19 |
| **8** | .15 | .15 | | | 9.6 | 1.5 | 4.7 | 4.7 | .88 | .88 | 2.0 | 2.0 | .88 | .88 | .29 | .15 | **77** | **77** | 4.7 | 4.7 | 8.3 |
| **9** | .29 | .29 | .29 | .29 | 16 | .86 | 4.9 | 3.9 | | | 1.1 | 1.1 | .29 | .29 | 1.6 | .57 | 1.3 | 1.3 | **74** | **74** | .29 17 |

Table 1 reports the classification results on the test set; gray columns show the classification performance obtained after removing from the prototype set the prototypes which covered less than 1% of the training samples. These prototypes are more likely influenced by noise in the training set than by actual invariants of the class they represent. This removal causes the rejection of some samples but, as it can be noted, most of the rejected samples were previously misclassified; hence the effect of this pruning is an overall improvement of the classification reliability.

## 5. Concluding Remarks

In this paper we have presented a novel method for learning structural descriptions from examples, based on a formulation of the learning problem in terms of ARG's. Our method, like learning methods based on first-order logic, produces general prototypes easy to understand and to manipulate, but it is based on simpler operations (graph editing and graph matching) leading to a smaller overall computational cost.

At the moment, we are working on the optimization of the heuristic function in order to increase the prototype generality. We are also studying more sophisticated operators for attribute specialization, able to reduce further the learning time. Finally, a preliminary work is being done on a post-processing phase for removing from the prototypes unnecessary constraints introduced due to greedy nature of the algorithm.

## References

1. T. Pavlidis "Structural Pattern Recognition", Springer, New York, 1977.
2. B. T. Messmer and H. Bunke, "A new algorithm for error-tolerant subgraph isomorphism detection", IEEE Trans. on PAMI, vol. 20, n. 5, pp. 493-504, May 1998.
3. P. H. Winston, "Learning Structural Descriptions from Examples", Tech. Report MAC-TR-76, Dep. of Electrical Engineering and Computer Science - MIT, 1970
4. L. P. Cordella, P. Foggia, R. Genna and M. Vento, "Prototyping Structural Descriptions: an Inductive Learning Approach", Advances in Pattern Recognition, Lecture Notes in Computer Science, n. 1451, pp. 339-348, Springer-Verlag, 1998.
5. R. S. Michalski, "Pattern recognition as rule-guided inductive inference", IEEE Trans. on PAMI, vol. 2, n. 4, pp. 349-361, July 1980.
6. N. Lavrac, S. Dzeroski, "Inductive Logic Programming: Techniques and Applications", Ellis Horwood, 1994.
7. A. Pearce, T. Caelly, and W. F. Bischof, "Rulegraphs for graph matching in pattern recognition", Pattern Recognition, vol. 27, n. 9, pp. 1231-1247, 1994.
8. J. Hsu, S. Wang, "A machine learning approach for acquiring descriptive classification rules of shape contours", Pattern Recognition, vol. 30, n. 2, pp. 245-252, 1997.
9. J. R. Quinlan, "Learning Logical Definitions from Relations", Machine Learning, vol. 5, n. 3, pp. 239-266, 1993.
10. L.P. Cordella, P. Foggia, C. Sansone, M. Vento, "Subgraph Transformations for the Inexact Matching of Attributed Relational Graphs", Computing, vol. 12, pp. 43-52, 1998.
11. L.P. Cordella, P. Foggia, C. Sansone, M. Vento, "Performance Evaluation of the VF Graph Matching Algorithm", Proc. 10th ICIAP, 1999, to appear.
12. ETL Character Database, Electrotechnical Laboratory – Japanese Technical Committe for OCR. Distributed during the 2nd ICDAR, 1993.
13. A. Chianese, L.P. Cordella, M. De Santo and M. Vento, "Decomposition of ribbon-like shapes", Proc. 6th SCIA, Oulu, Finland, pp. 416-423, 1989.