Generalisation Operators for Lists Embedded in a Metric Space

V. Estruch C. Ferri J. Hernández-Orallo M.J. Ramírez-Quintana

DSIC, Univ. Politècnica de València
Camí de Vera s/n, 46020 València, Spain.
{vestruch,cferri,jorallo,mramirez}@dsic.upv.es

Abstract. In some application areas, similarities and distances are used to calculate how similar two objects are in order to use these measurements to find related objects, to cluster a set of objects, to make classifications or to perform an approximate search guided by the distance. In many other application areas, we require patterns to describe similarities in the data. These patterns are usually constructed through generalisation (or specialisation) operators. For every data structure, we can define distances. In fact, we may find different distances for sets, lists, atoms, numbers, ontologies, web pages, etc. We can also define pattern languages and use generalisation operators over them. However, for many data structures, distances and generalisation operators are not consistent. For instance, for lists (or sequences), edit distances are not consistent with regular languages, since, for a regular pattern such as $*a$, the covered set of lists might be far away in terms of the edit distance (e.g. *bbbbbba* and *aa*). In this paper we investigate the way in which, given a pattern language, we can define a pair of generalisation operator and distance which are consistent. We define the notion of (minimal) distance-based generalisation operators for lists. We illustrate positive results with two different pattern languages.

Keywords: Distance-based methods, inductive operators, induction with distances, list-based representations

1 Introduction

Distance-based (or more generally, instance-based) methods are a powerful tool in the field of machine learning. Several reasons back its popularity, among them, we must stand out its capability to cope with different data representations: these methods are designed on the basis of a similarity principle (similar examples should share similar properties) which makes them easily adaptable to different datatypes via redefining the similarity (distance) function. In this sense, multiple distances and similarity functions can be found in the literature.

However, in the area of Inductive Programming, the use of distances is still at a very incipient state. Inductive Programming is concerned with the automated

construction of declarative programs from data. We can distinguish several approaches to this problem according to the knowledge representation adopted. For instance, the field known as *Inductive Logic Programming* (ILP) [13] aims to induce consistent first order theories from data represented as first order objects (atoms or clauses). A natural extension of this comes when we move to higher-order logics [1, 12]. The synthesis of functional programs arises when training data consist in a sample of inputs and outputs of a evaluation function [14, 16]. A more generic framework corresponds to the induction of functional-logic theories. This paradigm centres on performing induction within a formal context that combines the strengths of logic and functional programming [7, 10, 9].

In this area, the use of similarity functions and distances has been restricted to ILP, and very specially for machine learning applications of ILP and not for program synthesis. The reason for this limited success of the use of distances in inductive programming is twofold. First, distances and similarities return numerical values which are difficult to integrate with declarative models. A model such as "the sequence *aabb* has been classified as positive since it is similar to the sequence *aaba* which is also positive" cannot reduce the part "similar to" to a traditional declarative notation, since it usually involves an external function $similarity(s_1, s_2)$ and a numerical threshold. In other words, no declarative pattern has been defined to capture the notion of similarity or, at least, to be consistent with the notion. Second, although declarative languages constitute an elegant and powerful framework for program synthesis, they show some limitations when the semantics of the data representation does not match the implicit semantics managed by these declarative languages. An example of this is found when working with lists or sequences. From a declarative point of view, lists are recursively defined in terms of a special item (head) and a tail, which is another (sub)list. This perspective makes it difficult for the search of patterns in data that does not suit this definition. For instance, if we are given the lists *abaca* and *bc*, it is not immediate to learn a pattern of the form $*b*c*$ because of the simple fact that the heads of the lists do not match. Unfortunately, list-based representations appear in many real-world domains, which might put some limits on the applicability of declarative tools. For instance, in bioinformatics, compounds such as amino-acids have a direct representation as sequences of symbols. Furthermore, other much more complex molecules can also be described in terms of sequences by using the so-called 1-D or SMILE representation [17]. Another example is found in text or web mining where documents are usually transformed into sequences of words. Very common software utilities such as command line completion or orthographic correctors work on lists as well.

In general, we could wonder if some of the tools employed in inductive programming (generalisation operators) could be upgraded to deal with list-based representations in a more satisfactory way and overcome this limitation. In [4, 6], we consider the possibility by analysing the relationship between distance and generalisation. In [5] we analysed this framework employing distances and generalisations for graphs.

Note that most of the applications that handle sequences usually employ distances in order to find the most similar sequences in data. Distances (and consequently, metric spaces) play an important role in many inductive techniques that have been developed to date. Similarity offers a well-founded inference principle for learning and reasoning since it is commonly assumed that *similar objects have similar properties*. Given the importance of lists as a datatype for knowledge representation, several distances can be found in the literature, being the edit distance [11] the best-known. The drawback is that these methods do not infer a model (or patterns) from data as declarative inductive (or more general, symbolic) learners do.

Therefore, if we were able to find out a connection between distance and generalisation we could, on the one hand, define more suitable generalisation operators to work with structured data in general and with lists in particular; and on the other hand, we could come up with induction techniques capable of transforming distance-based method outputs into symbolic models, and consequently, more comprehensible explanations for the user.

There might be many different ways to establish a connection between distance and generalisation. Ensuring the consistency between them is a compelling one. Note that if the generalisation process is not driven by the distance, this might result in patterns that do not capture the semantics of the distance, so giving wrong explanations about why objects are similar. Let us see an example of this. If we consider the edit distance over the lists *bbab*, *bab* and *aaba*, we see that the list *ab* is close to the previous lists (distances are 2, 1, and 2 respectively). However, a typical pattern that can be obtained by some model-based methods, **ba**, does not cover the list *ab*. The pattern does cover the list *dededfafbakgagggeewdsc*, which is at distance 20 from the three original lists. The pattern and the distance are up to some point inconsistent since those elements that are most similar to the initial examples which are excluded.

Although there are other important works on hybridisation, they tend to ignore the problem of consistency between the semantic of the model learnt and the semantic of the underlying distance. Basically, what we do is to define some simple conditions that a generalisation operator should have in order to behave in a consistent way wrt. a distance. These operators are called distance-based generalisation operators.

In this paper, we address the problem of inducing patterns from lists of symbols embedded in a metric space. In other words, the work we present here can be seen as an instantiation for lists of the general framework aforementioned. It is noteworthy that, even though first-order logic constitutes an elegant and powerful framework for symbolic knowledge representation, lists have a complex and a little intuitive representation by means of first-order formulas which implies that patterns over lists have also a complex representation. This fact makes rather complicated that the ILP techniques can find patterns over lists by applying a generalisation operator like the *lgg*. In fact, one of the consequences derived of this term-based representation is that we need auxiliary predicates to extract requested information which is packed in a term (like *member, head,*

tail, previous, , ...). Hence, useful patterns might not be learnt if we have not previously defined the correct auxiliary predicates [3].

This paper is organised as follows. Section 2 contains an overview of our proposal. In Section 3, we analyse how our framework could be used to learn symbolic patterns from lists. To this end, we introduce two different pattern languages: $\mathcal{L}_0$ and another more expressive $\mathcal{L}_1$, and study how to define (minimal) distance-based operators in all of them. Finally, conclusions and future work are given in Section 4.

2 Framework

In this section we summarise the main concepts of our setting which integrates distances and generalisation. For a more detailed presentation of it we refer the reader to [3].

The underlying idea in our proposal is that, in order to have a *true* connection between distance and generalisation, the generalisation process have to take the underlying distance into consideration (or at least the two must be consistent). This special relation is formalised through three notions: *reachability*, *intrinsicality* and *minimality*.

Reachability implies that the generalisation of two elements ought to include those paths (a sequence of elements in the metric space) that allow us to reach both elements from each other by making small "steps". The concept of short step must be understood in the sense of the distance.

The second property arises from the observation that the distance between two elements is always given by the length of the shortest paths. Thus, if we want our generalisation to be compatible with the distance, we need the elements belonging to the shortest paths to be covered by the generalisation. This condition is called *intrinsicality*.

The two above properties have been defined for two elements since they are established in terms of the distance which is a binary function. But generalisation operators are not binary, thus for more than two elements, the connection between distance and generalisation turns a bit unclear. It seems that the properties of reachability and intrinsicality must be extended for this generic case. Distance-based algorithms suggest that it would make sense to impose the notion of intrinsicality for *some* pairs of elements. The pairs of elements that will have to comply with the intrinsicality property will be set by a path or connected graph which we will call *nerve*. Furthermore, we obtain with this a more generic notion of reachability since all the elements in the set are reachable from any of them by moving from one element to another through combinations of (intrinsical) paths.

In Figure 1, generalisations $G1$ and $G2$ do not connect the three elements to be generalised. Only the generalisations $G3$ and $G4$ connect the three elements through combinations of straight segments.

Finally, the last property concerns with the notion of *minimality*, which is understood not only in terms of fitting the set (i.e., semantic minimality) but

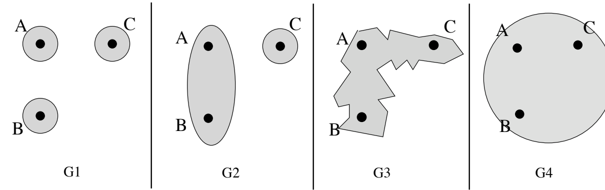

Fig. 1. Generalising the elements $E = \{A, B, C\}$. Elements in E are not reachable through a path of segments in generalisations $G1$ and $G2$. For any two elements in E, generalisations $G3$ and $G4$ include a path of segments connecting them.

also as the simplicity of the pattern (i.e., syntactic minimality). In Figure 1, $G3$ is an example of a very specific and rather complicated generalisation of A, B and C.

2.1 Distance-based Inductive Operators

Next, we formally show how the three previous notions are employed in order to define the so-called distance-based generalisation operators.

A generalisation of a finite set of elements $E \subset X$ could be seen as any superset of E in X. Therefore, a generalisation operator (denoted by Δ) simply maps sets of elements E into supersets. As known, this superset can be extensionally or intensionally defined, being the latter one more useful from a predictive/explanatory point of view. Symbolic patterns constitute a widely-spread manner of representing intensional generalisations. For instance, the pattern $a*$ denotes all the lists headed by the symbol a. We denote by $\mathcal{L}$ the pattern language and by $Set(p)$ the set of all the elements in X that the pattern $p \in \mathcal{L}$ represents. For instance, $Set(a*) = \{a, aa, ab, \ldots\}$. If necessary, $\mathcal{L}$ expressiveness can always be increased by combining patterns via logical operators (e.g. pattern disjunction). In this work, disjunction is denoted by the symbol $+$ and the expression $p_1 + p_2$ represents the set $Set(p_1) \cup Set(p_2)$. For simplicity, the pattern $p = p_1 + \ldots + p_n$ will be expressed as $p = \sum_{i=1}^{n} p_i$.

Now, we can already introduce the definition of binary distance-based pattern and binary distance-based generalisation operator.

Definition 1. ***(Binary distance-based pattern and binary distance-based generalisation operator)*** *Let (X, d) be a metric space, $\mathcal{L}$ a pattern language, and a set of elements $E = \{e_1, e_2\} \subset X$. We say that a pattern $p \in \mathcal{L}$ is a binary distance-based (db) pattern of E if p covers all the elements between e_1 and e_2*[1]*. Additionally, we say that Δ is a binary distance-based generalisation (dbg) operator if $\Delta(e_1, e_2)$ always computes a binary distance-based pattern.*

[1] Given a metric space (X, d) and two elements $e_1, e_2 \in X$, we say that an element $e_3 \in X$ is between e_1 and e_2, or is an intermediate element wrt. d, if $d(e_1, e_2) = d(e_1, e_3) + d(e_3, e_2)$

As previously said, for the case of more than two elements to be generalised, the concept of "nerve" of a set of elements E is needed to define non-binary *dbg* operators. Informally, a nerve of E is simply a connected[2] graph whose vertices are the elements belonging to E. Observe that if $E = \{e_1, e_2\}$, the only possible nerve is a one-edged graph. Formally,

Definition 2. ***(Nerve function)*** *Let (X, d) be a metric space and let S_G be the set of undirected and connected graphs over subsets of X. A nerve function $N : 2^X \to S_G$ maps every finite set $E \subset 2^X$ into a graph $G \in S_G$, such that each element e in E is inequivocally represented by a vertex in G and vice versa. We say the obtained graph $N(E)$ is a nerve of E.*

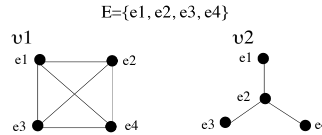

Fig. 2. Two nerves for the set E. **(Left)** ν_1 is a complete graph. **(Right)** ν_2 is a 3-star graph.

Some typical nerve functions are the complete graph, and a radial/star graph around a vertex (see Figure 2).

Recall that the nerve corresponds to the notion of reachability and indicates which intermediate elements must be covered by the generalisations. In a more precise way,

Definition 3. ***(Skeleton)*** *Let (X, d) be a metric space, $\mathcal{L}$ a pattern language, a set $E \subseteq X$, and ν a nerve of E. Then, the skeleton of E wrt. ν, denoted by skeleton(ν), is defined as a set which only includes all the elements $z \in X$ between x and y, for every $(x, y) \in \nu$.*

Consequently, we look for generalisations that include the skeleton. From here, we can define the notion of distance-based pattern wrt. a nerve.

Definition 4. ***(Distance-based pattern and distance-based pattern wrt. a nerve ν)*** *Let (X, d) be a metric space, $\mathcal{L}$ a pattern language, E a finite set of examples. A pattern p is a db pattern of E if there exists a nerve ν of E such that skeleton$(\nu) \subset Set(p)$. If the nerve ν is known, then we will say that p is a db pattern of E wrt. ν.*

And, from here, we have:

[2] Here, the term connected refers to the well-known property for graphs.

Definition 5. ***(Distance-based generalisation operator)*** *Let (X, d) be a metric space and $\mathcal{L}$ be a pattern language. Given a generalisation operator Δ, we will say that Δ is a dbg operator if for every $E \subseteq X$, $\Delta(E)$ is a db pattern of E.*

The above definition can be characterised for one nerve function in particular.

Definition 6. ***(Distance-based generalisation operator wrt. a nerve function)*** *Let (X, d) be a metric space and $\mathcal{L}$ a pattern language. A generalisation operator Δ is a dbg operator wrt. a nerve function N if for every $E \subseteq X$ then $\Delta(E)$ is a db pattern of E wrt. $N(E)$.*

In general it is quite hard to prove that a generalisation operator is *db* wrt. any nerve function. Fortunately, for most of the applications it is enough to exist a particular nerve function wrt. Δ is distance-based. If the nerve is known beforehand, we speak of *distance-based generalisation operators wrt. a nerve function N*.

Proposition 1. *Let $\mathcal{L}$ be a pattern language endowed with the operation $+$ and let Δ^b be a binary dbg operator in $\mathcal{L}$. Given a finite set of elements E and a nerve function N, the generalisation operator Δ_N defined as follows is a dbg operator wrt. N.*

$$\Delta_N(E) = \sum_{\forall (e_1, e_j) \in N(E)} \Delta^b(e_i, e_j)$$

Proof. It follows from the definition of *dbg* operator.

2.2 Minimality

Given the definition of *dbg* operator in the previous section, we can now guarantee that a pattern obtained by a *dbg* operator from a set of elements ensures that all the original elements are reachable inside the pattern through intrinsic (direct) paths. However, the generalisation can contain many other, even distant, elements.

An abstract, well-founded and widely-used principle that connects the notions of fitness and simplicity is the well-known MDL/MML principle [15, 19]. According to this principle, in our framework, the optimality of a generalisation will be defined in terms of a cost function, denoted by $k(E, p)$, which considers both the complexity of the pattern p and how well the pattern p fits E in terms of the underlying distance.

From a formal viewpoint, a cost function $k : 2^X \times \mathcal{L} \to \mathbf{R}^+ \cup \{0\}$ is a mapping where we assume that E is always finite, p is any pattern covering E and $k(E, p)$ can only be infinite when $Set(p) = X$.

As usual in MDL/MML approaches, most of the $k(E, p)$ functions will be expressed as the sum of a complexity (syntactic) function $c(p)$ (which measures how complicated the pattern is) and a fitness function $c(E|p)$ (which measures

how the pattern fits the data E). As said, the most novel point here is that $c(E|p)$ will be expressed in terms of the distance employed.

As $c(p)$ measures how complex a pattern is, this function will strongly depend on the sort of data and the pattern space $\mathcal{L}$ we are dealing with. For instance, if the generalisation of two real numbers is a closed interval containing them, then a simple choice for $c(p)$ would be the length of the interval.

As $c(E|p)$ must be based on the underlying distance, a lot of definitions are based on or inspired by the well-known concept of border of a set[3]. But as the concept of border of a set is something intrinsic to metric spaces, several general definitions of $c(E|p)$ can be given independently from the datatype as shown in Table 1.

	$\mathcal{L}$	$c(E\|p)$
1	Any	$\sum_{\forall e \in E} r_e$ $r_e = inf_{r \in \mathbf{R}} B(e, r_e) \not\subset Set(p)$
2	Any	$\sum_{\forall e \in E} r_e$ $r_e = sup_{r \in \mathbf{R}} B(e, r_e) \subset Set(p)$
3	Any	$\sum_{\forall e \in E} min_{e' \in \partial Set(p)} d(e, e')$
4	$Set(p)$ is a bound set	$\sum_{\forall e \in E} min_{e' \in \partial Set(p)} d(e, e')$ $+ max_{e'' \in \partial Set(p)} d(e, e'')$

Table 1. Some definitions of the function $c(E|p)$: 1-Infimum of uncovered elements, 2-Supremum of covered elements, 3-Minimum to the border, 4-Minimum and maximum to the border.

Now, we can introduce the definition of minimal distance-based generalisation operator and minimal distance-based generalisation operator relative to one nerve function.

Definition 7. (Minimal distance-based generalisation operator and minimal distance-based generalisation operator relative to one nerve function N**)** *Let (X, d) and N be a metric space and a nerve function, and let Δ be a dbg operator wrt. N defined in X using a pattern language $\mathcal{L}$. Given a finite set of elements $E \subset X$ and a cost function k, we will say that Δ is a minimal distance-based generalisation (mdbg) operator for k in $\mathcal{L}$ relative to N, if for every dbg operator Δ' wrt. N,*

$$k(E, \Delta(E)) \leq k(E, \Delta'(E)), \textit{for every finite set } E \subset X. \quad (1)$$

In similar terms, we say that a dbg operator Δ wrt. a nerve function N is a mdbg operator relative to N if the expression (1) holds for every dbg operator Δ' wrt. N.

The previous definition says nothing about how to compute the *mdbg* operator, and as we will see later, this might be difficult. A way to proceed is to first try

[3] Intuitively, if a pattern p_1 fits E better than a pattern p_2, then the border of p_1 (∂p_1) will somehow be nearer to E than the border of p_2 (∂p_2).

to simplify the optimisation problem as much as possible, as the next definition shows:

Definition 8. (Skeleton generalisation operator wrt. a nerve function N) *Let (X, d) be a metric space and N a nerve function. The skeleton generalisation operator $\bar{\Delta}_N$ is defined for every set $E \subset X$ as follows:*

$$\bar{\Delta}_N(E) = argmin_{\forall p \in \mathcal{L}: skeleton(N(E)) = Set(p)} \ k(E, p)$$

which means the simplest pattern that covers the skeleton of the evidence (given a nerve) and nothing more. Clearly, it is a *dbg* operator because it includes the skeleton, but it might not exist because it cannot be expressed.

The following section is devoted to defining *db* and *mdbg* operators for the list data type.

3 Inductive Operators for Lists

Lists or sequences is a widely-used datatype for data representation in different fields of automatic induction such as structured learning, bioinformatics or text mining. In this section, we apply our framework to finite lists of symbols by introducing two cost functions and two pattern languages for this sort of data and studying different *dbg* and *mdbg* operators for each particular combination of language and cost function. Due to space limitations as well as comprehensibility's sake, we sketch those proofs that are excessively long and would make the reading unnecessarily difficult. If needed, a complete detail of them can be found in [3].

3.1 Metric space, pattern languages and cost functions

Several distance functions for lists have been proposed in the literature. For instance, the Hamming distance defined for equally-length lists in [8], or the distance in [2], defined for infinite-length lists but which can easily be adapted for finite lists.

However, the most widely used distance function for lists is the edit distance (or Levenshtein distance [11]), which is the one we are working with. Specifically, we set the edit distance in such a way that only insertions and deletions are allowed (a substitution can be viewed as a deletion followed by an insertion or vice-versa).

Two different pattern languages $\mathcal{L}_0$ (single-list pattern language) and $\mathcal{L}_1$ (multiple-list pattern language) will be introduced in this section. The patterns in $\mathcal{L}_0$ are lists that are built from the extended alphabet $\Sigma' = \{\lambda\} \cup \Sigma \cup V$ where λ denotes the empty list, $\Sigma = \{a, b, c, \ldots\}$ is the alphabet (also called ground symbols) from which the lists to be generalised are defined, and $V = \{V_1, V_2, \ldots\}$ is a set of variables. The same variable cannot appear twice in a pattern. Each variable in a pattern represents a symbol from $\{\lambda\} \cup \Sigma$. Finally, the pattern language $\mathcal{L}_1$ is defined from $\mathcal{L}_0$ by means of the operation $+$ (see Subsection 2.1)

and aims to improve the expressiveness of $\mathcal{L}_0$. For instance, if we let $\Sigma = \{a, b\}$, then, the patterns $p_1 = aV_1V_2$ and $p_2 = bV_1V_2b$ belong to $\mathcal{L}_0$ where $Set(p_1) = \{aaa, aab, aba, abb, aa, ab, a\}$ and $Set(p_2) = \{baab, babb, bbab, bbbb, bab, bbb, bb\}$. In other words, the pattern p_1 denotes all those lists headed by the symbol a whose length ranges between 1 and 3. In a similar way, p_2 contains all the lists headed and ended by b whose length ranges between 2 and 4. Likewise, the pattern $p_3 = p_1 + p_2$ belongs to $\mathcal{L}_1$ and $Set(p_3) = Set(p_1) \cup Set(p_2) = \{aaa, aab, aba, abb, aa, ab, a, baab, babb, bbab, bbbb, bab, bbb, bb\}$.

With regard to the cost function, it is convenient to discuss some issues about the computation of the semantic cost function $c(\cdot|\cdot)$ for this particular setting. We will do this by means of an example. Suppose we are given the pattern $p = V_1V_2V_3V_4aV_5V_6V_7V_8$ and the element $e = ccaba$ which is covered by p. The computation of $c(e|p)$ is equivalent to find one of the nearest elements to e, namely e', which is not covered by p. Note that e' is not covered by p when the symbol a does not occur in e' (e.g. $e' = ccb$) or the number of symbols before or after each occurrence of a in e' is greater than 4 (e.g. $e' = ccbbbaba$). From this two possibilities, it is clear in this case that $e' = ccb$ is the nearest element to e not covered by p. This simple example allows us to affirm that the calculus $c(e|p)$ can be as complicated as determining the number of times a sequence s_1 occurs in a sequence s_2. Generally speaking, if s_p is the sequence of ground symbols in a pattern p and e' is the nearest element to e not covered by p, then e' will be a supersequence or a subsequence of e which will be obtained by modifying all the occurrences of s_p in e. Of course, as for the general form $c(E|p)$, this operation must be repeated for all the elements in E.

Therefore, if the learning problem requires the use of a cost function (e.g. because we are interested in minimal generalisations), it might be more convenient to approximate $c(E|p)$, instead of handling the original definition. For instance, we propose a naive but intuitive approximation of c inspired on the one we introduced in [3] for sets:

$$c'(E|p = \sum_{i=1}^{n} p_i) = \begin{cases} |E - E_1| + c(E_1|p_k), \exists p_k = V_1 \dots V_j \\ \quad \text{and } E_1 = \{e \in E : \text{length of } e \leq j\} \\ |E|, \text{ otherwise.} \end{cases}$$

The justification is as follows. If there exists a pattern $p_k = V_1 \dots V_j$ in p, then it is immediate that for every element e such that its length l is equal to or less than j, its nearest element not covered by p is, at least, at a distance $j - l + 1$, which is the value computed by $c(e|V_1 \dots V_j)$. Otherwise, we assume that the nearest element of e is, at least, at a distance of 1. Implicitly, we are assuming that the nearest element to e can be obtained by removing (or adding) one specific ground symbol from (to) e.

The simplicity of $c'(\cdot|\cdot)$ will help us to study and compare the computation of the *mdbg* in $\mathcal{L}_0$ and $\mathcal{L}_1$. As for $\mathcal{L}_0$, the cost function is directly defined as $k_0(E, p) = c'(E|p)$ (that is, the complexity of the pattern is disregarded). As for $\mathcal{L}_1$, we use $k_1(E, p) = c_1(p) + c'(E|p)$ where $c_1(p)$ measures the complexity of a pattern $p \in \mathcal{L}_1$ by counting both the ground and variable symbols in p.

3.2 Notation and previous definitions

The function $Seq(\cdot)$ defined over a pattern $p \in \mathcal{L}_0$ returns the sequence of ground symbols in p. For example, setting $p = V_1 aaV_2 b$, then $Seq(p) = aab$. The bar notation $|\cdot|$ denotes the length of a sequence (here a sequence can be an element, a pattern, etc.). For instance, in the previous case, $|p| = 5$. The i-th symbol in a sequence p is denoted by $p(i)$. Following with the example, $p(1) = V_1$, $p(2) = a$, $\ldots, p(5) = b$. Any sequence is indexed starting from 1. The set of all the indices of p is denoted by $I(p)$. Thus, $I(p) = \{1, 2, 3, 4, 5\}$. We sometimes use superscript as a shorthand notation to write sequences and patterns. For instance, $V^5a^3V^2$ is equivalent to $V_1 \ldots V_5aaaV_6V_7$, and $V^2(ab)^3c$ is the same as $V_1V_2abababc$. Finally, we will often introduce mappings that are defined from one sequence to another. By $Dom(\cdot)$ and $Im(\cdot)$ we denote the domain and the image, respectively, of a mapping.

The first concept that is required is:

Definition 9. (Maximum common subsequence) *Given a set of sequences* $E = \{e_1, \ldots, e_n\}$*, and according to [18], the maximum common subsequence (mcs, to abbreviate) is the longest (not necessarily continuous) subsequence of all the sequences in* E*.*

This concept is already widely used in pattern recognition. Note that the *mcs* of a group of sequences is not necessarily unique. The following definitions will let us work with the concept of common subsequence in a more algebraic fashion.

Definition 10. (Alignment) *Given two elements* e_1 *and* e_2*, we say that the mapping* $M^{e_1}_{e_2} : I(e_1) \rightarrow I(e_2)$ *is an alignment of* e_1 *with* e_2 *if:*

i) $\forall i \in Dom(M^{e_1}_{e_2}),\ e_1(i) = e_2(M^{e_1}_{e_2}(i))$
ii) $M^{e_1}_{e_2}$ *is a strictly increasing function in* $Dom(M^{e_1}_{e_2})$*.*

(Remark 1) If $Dom(M^{e_1}_{e_2}) = \emptyset$, we say that $M^{e_1}_{e_2}$ is the empty alignment of e_1 with e_2. Thus, for every pair of elements we can affirm that there is always at least one alignment between them.
(Remark 2) Note that the alignment definition does not exclude the case $e_1 = e_2$.
(Remark 3) We call $e_1(i) = e_2(M^{e_1}_{e_2}(i))$ a (symbol) matching. Thus, $|Dom(M^{e_1}_{e_2})|$ (or equivalently, $|Im(M^{e_1}_{e_2})|$) is the number of matchings between e_1 and e_2 captured by $M^{e_1}_{e_2}$, and the subsequence obtained by considering the i-th symbols of e_1 where $i \in Dom(M^{e_1}_{e_2})$ is the sequence of matchings. For the sake of simplicity, we denote this sequence by $Seq(M^{e_1}_{e_2})$.

Definition 11. (Optimal alignment) *Given two elements* e_1 *and* e_2*, if* $Seq(M^{e_1}_{e_2})$ *is a mcs of* e_1 *and* e_2*, then we say that* $M^{e_1}_{e_2}$ *is an optimal alignment.*

Since $I(e_1)$ and $I(e_2)$ are finite sets, an alignment $M^{e_1}_{e_2}$ can be written as a $2 \times n$ matrix where n (which we denote as $Rang(M^{e_1}_{e_2})$) is the number of matchings. Hence,

$$M_{e_2}^{e_1} = \begin{pmatrix} a_{11} \dots a_{1n} \\ a_{21} \dots a_{2n} \end{pmatrix}$$

where $e_1(a_{1i}) = e_2(a_{2i})$ for all $1 \leq i \leq n$ (condition i) from Definition 10) and $a_{1i} < a_{1(i+1)}$ and $a_{2i} < a_{2(i+1)}$ for all $1 \leq i \leq (n-1)$ (condition ii) from Definition 10). An element of $M_{e_2}^{e_1}$ placed at row i and column j is denoted by $(M_{e_2}^{e_1})_{ij}$.

Let us illustrate all these ideas by means of an example.

Example 1. Given the elements $e_1 = caabbc$ and $e_2 = aacd$ where $I(e_1) = \{1, 2, 3, 4, 5, 6\}$ and $I(e_2) = \{1, 2, 3, 4\}$. An alignment $M_{e_2}^{e_1}$ (M in short) is

$$M = \begin{pmatrix} 2\ 3\ 6 \\ 1\ 2\ 3 \end{pmatrix} \equiv \begin{matrix} c & a & a & b & b & c & \\ & a & a & & & c & d \end{matrix}$$

Note that M satisfies both conditions from Definition 10. Following with M, we have that $Dom(M) = \{2, 3, 6\}$, $Im(M) = \{1, 2, 3\}$, $Rang(M) = 3$ and $Seq(M) = aac$. Finally, M is an optimal alignment.

Given that different optimal alignments can be defined over two elements e_1 and e_2, we might be interested in obtaining a concrete optimal alignment. To do this, we define a total order over all of them which lets us formally specify which optimal alignment we want.

Definition 12. ***(Total order for optimal alignments)*** *Given two elements e_1 and e_2 and given the optimal alignments $M_{e_2}^{e_1}$ (M in short) and $N_{e_2}^{e_1}$ (N in short) defined as*

$$M = \begin{pmatrix} a_{11} \dots a_{1n} \\ a_{21} \dots a_{2n} \end{pmatrix} \; N = \begin{pmatrix} b_{11} \dots b_{1n} \\ b_{21} \dots b_{2n} \end{pmatrix}$$

we say that $M < N$ iff $(a_{11}, \dots, a_{1n}, a_{21}, \dots, a_{2n}) <_{LO} (b_{11}, \dots, b_{1n}, b_{21}, \dots, b_{2n})$ where $<_{LO}$ is the Lexicographical Order for numerical tuples.

Example 2. Given $e_1 = aab$ and $e_2 = ab$, we define the optimal alignments

$$M_{e_2}^{e_1} = \begin{pmatrix} 1\ 3 \\ 1\ 2 \end{pmatrix} \; N_{e_2}^{e_1} = \begin{pmatrix} 2\ 3 \\ 1\ 2 \end{pmatrix}$$

Then $M_{e_2}^{e_1} < N_{e_2}^{e_1}$.

Every alignment between two elements e_1 and e_2 induces a special pattern p which covers both e_1 and e_2. This pattern is unique and we call it the pattern associated to an alignment.

Definition 13. (Pattern associated to an alignment and optimal alignment pattern) *Let e_1 and e_2 be two elements in Σ^* and let $M_{e_2}^{e_1}$ (M in short) be an alignment of e_1 with e_2. We say that a pattern $p \in \mathcal{L}_0$ is a pattern associated to the alignment M (denoted by p_M), if*

i) $Seq(M) = Seq(p)$
ii) the variable symbols in p *are distributed as follows (letting* $n = Rang(M)$, $l_1 = |e_1|$, $l_2 = |e_2|$*):*

- *The number of variables in the pattern* p *before the first ground symbol is equal to*
$$((M)_{11} - 1) + ((M)_{21} - 1)$$
- *The number of variables between whatever two ground symbols* $p(i)$ *and* $p(j)$ $(i < j)$ *in* $Seq(p)$ *such that there does not exists* $i < k < j$ *where* $p(k)$ *is a ground symbol, is equal to*
$$((M)_{1(i+1)} - (M)_{1i} - 1) + ((M)_{2(i+1)} - (M)_{2i} - 1)$$
- *The number of variables after the last ground symbol in* p *is equal to*
$$(l_1 - (M)_{1n}) + (l_2 - (M)_{2n})$$

If $M_{e_2}^{e_1}$ *is an optimal alignment of* e_1 *with* e_2*, we say that* $p_{M_{e_2}^{e_1}}$ *is an optimal alignment pattern.*

For instance, the pattern associated to the alignment M in Example 1 is $p_M = V_1aaV_2V_3cV_4$, which is an optimal alignment pattern because M is an optimal alignment. Note that if M is the empty alignment then $p_M = V^{l_1+l_2}$ and $Seq(M) = \lambda$.

The alignment and optimal alignment concepts (Definitions 10 and 11) can be easily extended to cope with patterns. Given two patterns p_1 and p_2, $M_{p_2}^{p_1}$ is an alignment of p_1 with p_2 where only matchings between ground symbols are taken into account, that is, $\forall i \in Dom(M_{p_2}^{p_1}), p_1(i) = p_2(M_{p_2}^{p_1}(i))$, $p(i) \in \Sigma$ and $p_2(M_{p_2}^{p_1}(i)) \in \Sigma$. Analogously, $M_{p_2}^{p_1}$ is an optimal alignment if $Seq(M_{p_2}^{p_1})$ is a *msc* of p_1 and p_2.

To conclude, we introduce a binary bottom-up generalisation operator (called $\uparrow$-transformation) defined over $\mathcal{L}_0$, which allows us to move through the pattern language.

Definition 14. *Given two patterns* p_1 *and* p_2 *in* $\mathcal{L}_0$ *we define the binary mapping*
$$\begin{aligned} \uparrow(\cdot,\cdot) : \mathcal{L}_0 \times \mathcal{L}_0 &\rightarrow \mathcal{L}_0 \\ (p_1, p_2) &\rightarrow \uparrow(p_1, p_2) = p, \quad \textit{such that} \end{aligned}$$

1. *Let* $M_{p_2}^{p_1}$ *(*M *in short) be the minimum optimal alignment of* p_1 *with* p_2*, then* $Seq(p) = Seq(M)$*.*
2. *If* $Seq(M) = \lambda$ *then* $p = V^{max\{|p_1|,|p_2|\}}$*. Otherwise, the distribution of the variables in* p *is:*
   - *Before the first ground symbol in* p*, the number of variable is equal to:*
$$max\{(M)_{11} - 1, (M)_{21} - 1\}$$

– *Between two consecutive ground symbols in p, the number of variables is equal to:*

$$max\{(M)_{1(i+1)} - (M)_{1i} - 1, (M)_{2(i+1)} - (M)_{2i} - 1\}$$

– *After the last ground symbol in p, the number of variables is equal to (letting $n = Rang(M)$, $l_1 = |p_1|$ and $l_2 = |p_2|$):*

$$max\{l_1 - (M)_{1n}, l_2 - (M)_{2n}\}$$

Example 3. Given the patterns $p_1 = abcV_1$, $p_2 = V_1abcccV_2$ and $p_3 = dV_1$, then $\uparrow(p_1, p_2) = VabcV^3$ and $\uparrow(p_1, p_3) = V^4$.

Proposition 2. *For every pair of patterns p_1 and p_2 in $\mathcal{L}_0$, if $p = \uparrow(p_1, p_2)$ then $Set(p_1) \subset (p)$ and $Set(p_2) \subset (p)$.*

Proof. It directly comes from the definition of the $\uparrow$-transformation.

Next, we explain how to define *dbg* operators for the different pattern languages, and we study the possibility of finding *mdbg* operators for $(\mathcal{L}_0, k_0)$ and $(\mathcal{L}_1, k_1)$.

3.3 Single list pattern language ($\mathcal{L}_0$)

One would expect that if $\Delta(E)$ computes a pattern p such that $Seq(p)$ is a *mcs* of the lists in E, then $\Delta(\cdot)$ is a *dbg* operator. However, we find that this operator is not, in general, distance-based. The following example illustrates this:

Example 4. Let $E = \{e_1, e_2, e_3\}$ where $e_1 = c^5a^3b^3$, $e_2 = c^5a^2d^4$ and $e_3 = a^3b^3d^4c^5$ are the elements to be generalised. Initially, we are going to fix a nerve for these elements, namely, the complete nerve (see Figure 3).

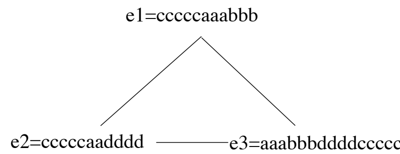

Fig. 3. A complete nerve ν for the evidence $E = \{e_1, e_2, e_3\}$.

The pattern $p = V^{10}c^5V^6$ generalises E, and $Seq(p)$ is a *mcs* of the lists in E. However, this pattern is not a *db* pattern of E since, for example, the element a^3b^3 (which is between e_1 and e_3) and the element a^2d^4 (which is between e_2 and e_3) are not covered by p. As a matter of fact, no pattern containing the ground symbol c will be *db* and this result is independent of the nerve chosen.

The explanation for this apparently counterintuitive result is based on how the distance between the different pairs of elements e_i and e_j is calculated. In fact, although all the lists in E have subsequence c^5 in common, this subsequence is never taken into account to compute the distance $d(e_i, e_j)$, for any pair (e_i, e_j) in ν. Therefore, the operator definition we propose next not only uses the concept of *mcs* but also uses others such as the $\uparrow$-transformation and the concept of nerve which ensures the condition of being *db*. First, we deal with the binary generalisation operator, and then we extend it for the n-ary case.

In the first stage, for any two elements e_1 and e_2 to be generalised, we need to somehow find out which patterns in $\mathcal{L}_0$ can cover those elements between e_1 and e_2.

Proposition 3. *Given the elements e_1, e_2 and e, if e is between e_1 and e_2, then there exists an optimal alignment pattern p associated to an optimal alignment of e_1 and e_2 such that $e \in Set(p)$.*

Proof. **(Sketch)** Let $M_e^{e_1}$ and $M_{e_2}^{e}$ be the optimal alignments of e_1 with e and e with e_2, respectively. We define the mapping M between e_1 and e_2 as the composition of $M_e^{e_1}$ and $M_{e_2}^{e}$. The goal is to prove first that M is an optimal alignment of e_1 with e_2 and then, see that the associated pattern $p_{_M}$ covers e. For this last step we distinguish two cases: *i)* M is the empty alignment and consequently $p_{_M} = V^{|e_1|+|e_2|}$. According to Proposition 21 in [3], if e is between e_1 and e_2, then $|e| \leq |e_1| + |e_2|$, hence $e \in Set(p_{_M})$. *ii)* M is not empty and we aim to prove that the variable symbols in M are distributed in such a way that we can ensure that $e \in Set(p_{_M})$.

We will use the proposition above along with the $\uparrow$-transformation to define binary *db* operators.

Corollary 1. *Given the elements e_1 and e_2, if $\{p_i\}_{i=1}^n$ is the set of all the optimal alignment patterns of e_1 and e_2, then the generalisation operator defined as follows is db.*

$$\Delta^b(e_1, e_2) = \uparrow(p_1, \uparrow(p_2, \ldots \uparrow(p_{n-1}, p_n)) \ldots)$$

Proof. For every optimal alignment pattern, we know from Proposition 2, that

$$Set(p_i) \subset Set(\Delta^b(e_1, e_2)) \tag{2}$$

Then, from Proposition 3, we can write that

$$\forall \text{ element } e \text{ between } e_1 \text{ and } e_2 \ \Rightarrow \exists p_i : e \in Set(p_i) \tag{3}$$

Now, combining (2) and (3), we can affirm that

$$\forall \text{ element } e \text{ between } e_1 \text{ and } e_2 \ \Rightarrow e \in Set(\Delta^b(e_1, e_2)) \tag{4}$$

Hence, the generalisation operator is distance-based.

Next, we extend Corollary 1 for an arbitrary number of elements.

Corollary 2. *Given a finite set of elements $E \subset X$ and a function nerve N, the generalisation operator Δ defined in Algorithm 1 (where Δ^b is defined in Corollary 1) is db wrt. N.*

Proof. For every $(e_i, e_j) \in N(E)$, $Set(\Delta^b(e_i, e_j)) \subset Set(\Delta(E))$ by the definition of the $\uparrow$-transformation. Therefore, for every finite set E, $\Delta(E)$ is distance-based w.r.t. $N(E)$.

```
   Data: E = {e_1, ..., e_n}, Δ^b (binary dbg operator) and ν (a nerve of E)
   Result: Distance-based pattern of E wrt. ν
 1 begin
 2     k ← 0;
 3     L ← []/ * empty list * /;
 4     for (e_i, e_j) ∈ N(E) do
 5         L[k] ← Δ^b(e_i, e_j);
 6         k ← k + 1;
 7     end
 8     S ← {a_i ∈ Σ : ∀0 ≤ j ≤ k : a_i ∈ Seq(L[j]);
 9     if S = ∅ then return V^{max{|L[j]|:∀0≤j≤k}} ;
10     else
11         p ← First(L);
12         Remove(L, p);
13         while L ≠ ∅ do
14             Find p_i ∈ L: ∃a_j ∈ S, a_j ∈ Seq(↑ (p, p_i));
15             p ←↑ (p, p_i);
16             Remove(L, p_j);
17         end
18         return p;
19     end
20 end
```

Algorithm 1: An algorithm to compute a *db* pattern of a set of lists E wrt. a nerve ν.

Algorithm 1 returns a pattern p such that $Set(\Delta^b(e_i, e_j)) \subset Set(p)$, for every pair of elements in $N(E)$, by iteratively applying the $\uparrow$-transformation over all the patterns $\Delta^b(e_i, e_j)$. The *else*-block is important since it ensures that $Seq(p) \neq \lambda$, if all the sequences $Seq(\Delta^b(e_1, e_j))$ have a subsequence in common. Let us see an example of this.

Example 5. Given $E = \{e_1, e_2, e_3, e_4\}$ where $e_1 = abc$, $e_2 = cabcd$, $e_3 = c$, $e_4 = cab$ and the nerve $N(E) = \{(e_1, e_2), (e_2, e_3), (e_2, e_4)\}$. The binary distance-

based generalisations (lines 5-7 in the algorithm) are:

$$\begin{aligned} L[0] &= \Delta^b(e_1, e_2) = VabcV \\ L[1] &= \Delta^b(e_2, e_3) = VcabV \\ L[2] &= \Delta^b(e_2, e_4) = V^3cV^4 \end{aligned}$$

If we applied the $\uparrow$-transformation in any arbitrary order over the set of binary patterns, we could obtain for example:

$$\begin{aligned} &p \leftarrow VabcV \\ &p \leftarrow \uparrow (p, VcabV) = V^2abV^2 \\ &p \leftarrow \uparrow (p, V^3cV^4) = V^9 \end{aligned}$$

However, if the $\uparrow$-transformation is applied as the algorithm indicates (lines 8-17), then $S = \{c\}$ and the patterns would be merged in the following order:

$$\begin{aligned} &p \leftarrow \uparrow VabcV \\ &p \leftarrow \uparrow (p, V^3cV^4) = V^3cV^4 \\ &p \leftarrow \uparrow (p, VcabV) = V^3cV^4 \end{aligned}$$

With regard to the computation of *mdbg* operators in $(\mathcal{L}_0, k_0)$, the algorithm above always return the *mdbg*. On the one hand, if all the binary patterns have a subsequence in common, the algorithm computes a distance-based pattern p such that $Seq(p) \neq \lambda$ and the function $c'(E|p) = |E|$ which attains a minimum value. On the other hand, the algorithm returns a pattern with variable symbols only, and whose length is the minimum length required to be distance-based. Therefore, p is minimal as well.

3.4 Multiple list pattern language ($\mathcal{L}_1$)

We will define *dbg* operators in $\mathcal{L}_1$ via Δ_N (Proposition 1). The binary operator Δ^b required by Δ_N is the one introduced in Corollary 1. An example of how this operator works is shown below:

Example 6. Given a finite set of elements $E = \{e_1, e_2, e_3, e_4\}$ where $e_1 = a^2b^2d$, $e_2 = da^2c^2$, $e_3 = c^2db^2$ and $e_4 = ad$ and the nerve $N(E) = \{(e_1, e_2), (e_1, e_3), (e_1, e_4)\}$.

$$\begin{aligned} \Delta^b(e_1, e_2) &= p_1 = Va^2V^5 \\ \Delta^b(e_1, e_3) &= p_2 = V^5b^2 \\ \Delta^b(e_1, e_4) &= p_3 = VaV^3d \end{aligned}$$

Finally,

$$\Delta_N(E) = Va^2V^5 + V^5b^2 + VaV^3d$$

Observe that the solution for this example in $\mathcal{L}_0$ is just a pattern consisting of variable symbols only, which shows the utility of $\mathcal{L}_1$. Next, let us see how to obtain *mdbg* operators in $\mathcal{L}_1$.

Since the only way we know to define a distance-based operator in $\mathcal{L}_1$ consists in fixing a nerve beforehand, it is reasonable to study how to define *mdbg* operators relative to a nerve function. However, the calculus of the *mdbg* operator is not easy at all. Basically, the question is whether the *mdbg* operators relative to a nerve function N can be defined in terms of Δ_N and the $\upharpoonleft$-transformation. However, this result seems hard to be established. On the one hand, we ignore how to explicitly define most of the Δ^b operators (since Corollary 1 only establishes a sufficient condition) and on the other hand, we must take into consideration some inherent limitations of the $\upharpoonleft$-transformation:

1. The *mdb* pattern might not be found by applying the $\upharpoonleft$-transformation over Δ_N if this one uses the binary operator Δ^b defined in Corollary 1: we will illustrate this by means of an example.

   Example 7. Given the set $E = \{e_1, e_2, e_3\}$, where $e_1 = a_1a_2a_3$, $e_2 = a_1a_6a_7$ and $e_3 = a_2a_4a_5$, and $N(E) = \{(e_1, e_2), (e_1, e_3)\}$. The optimal alignment patterns which are associated to (e_1, e_2) and (e_1, e_3), respectively, are a_1V^4 and Va_2V^3. Then a_1V^4 is a *db* pattern of (e_1, e_2) (since it is the only optimal alignment pattern) and Va_2V^3 is a *db* pattern of (e_2, e_3) (since it is the only optimal alignment pattern). Hence, the pattern $p = a_1V^4 + Va_2V^3$ is *db* w.r.t. $N(E)$. However, the pattern $p' = a_1V^4 + a_2V^3$ is distance-based (the only element between e_1 and e_2, which is not covered by a_2V^3, is $a_1a_2a_4a_5$ but this is covered by a_1V^4) but $Set(p') \not\subset Set(p)$. The *mdb* pattern for E will have $|p'|$ or even fewer symbols and this will never be achieved by the $\upharpoonleft$-transformation over the optimal alignment patterns.

   Therefore, given that Δ^b is defined from the concept of optimal alignment patterns and Δ_N is defined from Δ^b, it is not possible that the *mdbg* operator can be expressed in terms of the $\upharpoonleft$-transformation and Δ_N.
2. The *mdbg* pattern might not be found by applying the $\upharpoonleft$-transformation over $skeleton(N(E))$: from the previous point, we could think that the *mdb* pattern cannot be found because the optimal alignment patterns are excessively general. However, if it was so, it would mean that starting the search from something extremely specific, namely the skeleton, the *mdb* pattern should be found. However, this is not true as the next example reveals:

   Example 8. Given $E = \{e_1, e_2, e_3, e_4, e_5\}$ where $e_1 = ac^3b^2$, $e_2 = ab^2$, $e_3 = ab^2ce$, $e_4 = d$ and $e_5 = fgh$ and the nerve depicted below:
   If we group the elements according to its similarity and then apply the $\upharpoonleft$-transformation over the different groups, the pattern obtained would attain a lower value for $k_1(E, \cdot)$. Taking this strategy into account, we can distinguish several meaningful grouping criteria. For instance, those elements which contain the subsequence *abb* (G_1) and those which do not (G_2). That is,

   $$G_1 = \{ac^3b^2, acb^2, ac^2b^2, ab^2, \ldots, ab^2d\}$$
   $$G_2 = \{dfgh, fdgh, fgdh, fghd\}$$

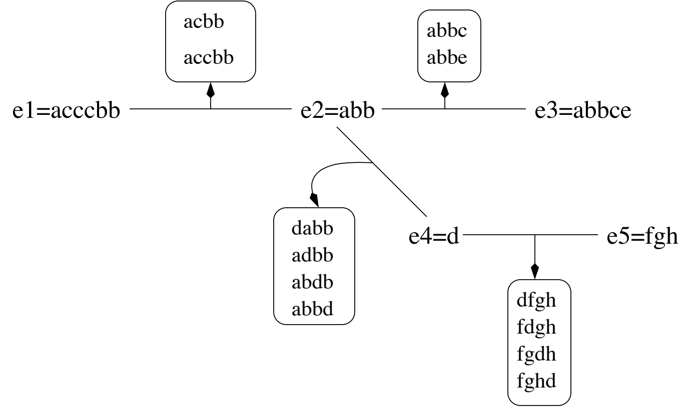

Fig. 4. A naive generalisation of the set E w.r.t. the nerve N(E). Circled elements are the intermediate elements.

In this particular case, it does not matter how the elements in the groups are ranked in order to apply the $\upharpoonleft$-transformation since the final result remains invariable. Thus, we can write

$$p_1 = \upharpoonleft (G_1) + \upharpoonleft (G_2) = VaV^3bVbV^2 + VfVgVhV$$

For any other binary splitting, we would have elements having no subsequence in common in the same group (e.g. abb and $dfgh$). The shortest patterns would be

$$p_2 = aV^3b^2V^2 + V^4$$
$$p_3 = V^6$$

Using three groups, another interesting possibility can be explored. For instance, $G_1 = \{fgh\}$, those elements containing the subsequence d (G_2) and the remaining ones (G_3). Depending on the order of the elements in G_2 we could obtain by applying the $uparrow$-transformation.

$$p_4 = V^5 + aV^3b^2V^2$$
$$p_5 = V^3dV^3 + aV^3b^2V^2 + fgh$$

Finally, it is not worth using more than three groups because of the excessive length of the pattern obtained. Evaluating the different patterns, we have that:

$$k_1(E, p_1) = c(p_1) + c'(E|p_1) = 17 + 5 = 22$$
$$k_1(E, p_2) = c(p_2) + c'(E|p_2) = 12 + 10 = 22$$
$$k_1(E, p_3) = c(p_3) + c'(E|p_3) = 6 + 17 = 23$$
$$k_1(E, p_4) = c(p_4) + c'(E|p_4) = 13 + 13 = 26$$
$$k_1(E, p_5) = c(p_5) + c'(E|p_5) = 18 + 5 = 23$$

But the following patterns are also distance-based for E:

$$p_6 = V^3cV^2 + V^4$$
$$p_7 = aV^5 + V^4$$

where

$$k_1(E, p_6) = c(p_6) + c'(E|p_6) = 10 + 10 = 20$$
$$k_1(E, p_7) = c(p_7) + c'(E|p_7) = 10 + 10 = 20$$

However, neither p_6 nor p_7 can be derived from a $\uparrow$-transformation since this tends to extract the longest common subsequence. Observe that all the elements which have the subsequence c or a also contain the subsequence abb in common.

From this previous analysis, we can conclude that the $\uparrow$-transformation is not enough in itself to explore the search space. We need a generalisation tool which is not based on the concept of the longest common subsequence. For this purpose, we introduce the so-called inverse substitution.

Definition 15. ***(Inverse substitution)*** *Given a pattern p in $\mathcal{L}_0$ or in $\mathcal{L}_1$ an inverse substitution σ^{-1} is a set of indices where each index denotes a ground symbol in p to be changed by a variable. Thus, $p\sigma^{-1}$ represents the new pattern which is obtained by applying σ^{-1} over p.*

Basically, an inverse substitution just changes ground symbols by variables. For example, given $p = VaabV$ and $\sigma^{-1} = \{2, 4\}$ then $p\sigma^{-1} = V^2aV^2$. Now, we are in conditions to introduce the next proposition:

Proposition 4. *Given a finite set of elements $E = \{e_1, \ldots, e_n\}$ and a nerve function N. If we set $S = skeleton(N(E))$ then there exists a partition P of the set S and a collection of inverse substitutions $\{\sigma_1^{-1}, \ldots, \sigma_n^{-1}\}$ such that the pattern*

$$p = \sum_{\forall P_i = \{e_{k_i}\}_{k_i=1}^{m_i} \in P} \uparrow (\{e_{k_i}\sigma_{k_i}^{-1}\}_{k_i=1}^{m_i})$$

is a mdb pattern of E relative to $N(E)$.

Proof. **(Sketch)**. We can assume that there exists a pattern $p = \sum_{i=1}^{n} p_i$ such that $k(E, p)$ attains a minimum value. The pattern p induces a partition of $E = \cup E_i$ in such a way that $e_i \in E_i$ iff $e_i \in Set(p_i)$. Next, we remove repeated elements in the different E_i in order to make sure that the subsets E_i are pairwise disjoints. Finally, the proposition can be proved using the concepts of inverse substitution and $\uparrow$-transformation over the partition we have set.

This latter proposition leads to an exhaustive search algorithm in order to compute the *mbdg* operator. This algorithm turns out to be useless in general due to the size of the search space (the number of different possibilities for the partition of $skeleton(N(E))$ and substitutions). In fact, for a particular version of $\mathcal{L}_1$, we have proved that this optimisation problem is NP-Hard (see [3]).

Hence, the other option is to approximate the calculus of the *mdb* patterns. To do this, we use a greedy search schema driven by the cost function. That is, for each iteration, the $\upharpoonleft$-transformation is applied over the pair of patterns that reduces th cost function most. This idea is formalised in the Algorithm 2 and illustrated in Example 9.

Input: $E = \{e_1, \ldots, e_n\}$, Δ^b (binary *dbg* operator) and N (nerve function)
Output: A pattern which approximates a *mdb* pattern of E w.r.t. $N(E)$

1 $\tilde{\Delta}_N(E)$
2 **begin**
3 $\quad k \leftarrow 1$;
4 $\quad$ **for** $(e_i, e_j) \in N(E)$ **do**
5 $\quad\quad p_k \leftarrow \Delta^b(e_i, e_j)$;
6 $\quad\quad k \leftarrow k + 1$;
7 $\quad$ **end**
8 $\quad p = \sum_{k=1}^{n} p_k$;
9 $\quad$ **do**
10 $\quad\quad k_p \leftarrow k_1(E, p)$;
11 $\quad\quad p' \leftarrow argmin\{k_1(E, p_{ij}) : \forall 1 \leq i, j, \leq n, p_{ij} = \upharpoonleft (\{p_i, p_j\}) + (p - p_i - p_j)\}$;
12 $\quad\quad k'_p \leftarrow k_1(E, p')$;
13 $\quad\quad$ **if** $k_{p'} < k_p$ **then** $p \leftarrow p'$;
14 $\quad$ **while** $k_{p'} < k_p$
15 $\quad$ return p;
16 **end**
17 //The notation $p - p_i - p_j$ employed in the algorithm means all the patterns in p except p_i and p_j.;

Algorithm 2: A greedy algorithm which approximates the *mdbg* operator.

Example 9. Let E and $N(E)$ be the set of examples and the nerve employed in Example 6. Remember that,

$$\begin{aligned} p_1 &= \Delta^b(e_1, e_2) = Va^2V^5 \\ p_2 &= \Delta^b(e_1, e_3) = V^5b^2 \\ p_3 &= \Delta^b(e_1, e_4) = VaV^3d \end{aligned}$$

and

$$p = Va^2V^5 + V^5b^2 + VaV^3d$$

see lines 4-8 in the algorithm. Next, we have to apply the $\upharpoonleft$-transformation over each pair of binary generalisations and we choose the one which attains a lower value of $k_1(E, \cdot)$ (see lines 9-14). In our case, we must consider two possibilities:

$$\begin{aligned} p_1 &= \upharpoonleft (Va^2V^5, V^5b^2) + VaV^3d = V^8 + VaV^3d \\ &= V^8 \\ p_2 &= \upharpoonleft (Va^2V^5, VaV^3d) + V^5b^2 = VaV^6 + V^5b^2 \end{aligned}$$

Since $k_1(E, p_2) = 19$ is less than $k_1(E, p_1) = 27$, we choose the pattern p_2. The process stops when the pattern cannot be further improved. Note that the next iteration leads to

$$\uparrow (VaV^6, V^5b^2) = V^8$$

which performs worse than p_2. Therefore, the algorithm returns p_2.

4 Conclusions and Future Work

We have followed the connection between two major concepts in inductive programming, the concept of distance and generalisation, when applied to lists. This work is based in a correct integration of distance-based methods with symbolic inductive learners we introduced in [4][6]. This proposal relies on the novel concept of (minimal) distance-based generalisation operator, which aims to induce consistent (minimal) patterns from data embedded in a metric space.

The main contribution of this paper consists in studying how to apply our framework in order to infer consistent symbolic patterns from a particular structured data type (lists) and a distance function (edit distance). More concretely, we have seen how to define (minimal) distance-based generalisation operators for this domain. To do this, we have introduced two different pattern languages $\mathcal{L}_0$ and $\mathcal{L}_1$. The first language is made up of patterns which consist of finite sequences of ground and variable symbols. The language $\mathcal{L}_1$ extends $\mathcal{L}_0$ in that the disjunction of patterns is permitted. Additionally, we have defined a cost function for each language in order to study the minimality of the patterns we can obtain.

We have proved that for more than two sequences, the widely-used concept of *maximum common subsequence* does not necessarily lead to distance-based generalisation operators. In order to obtain this sort of operators, we need to introduce a new concept: namely, the concept of sequence associated to an optimal alignment. This kind of sequences leads to certain patterns that when combined, allows us to define distance-based operators. As for the minimality of these operators, we have shown this is a computational hard problem in $\mathcal{L}_1$. For this reason, we have introduced a greedy search algorithm which allows us to approximate minimal generalisations.

There are some work ahead to ease the integration of these generalisation operators into inductive programming tools. For instance, the computational complexity of the greedy search algorithm which approximates minimal patterns is a concern. This has a quadratic complexity with the number of subpatterns in the pattern obtained by Proposition 1. Unfortunately, this operation still has a high cost, if we want to run our algorithm over large data sets. Thus, it would be convenient to try other heuristics with a lower complexity that ensure a good approximation. Another one is devoted to the pattern languages that have been investigated. Note that both $\mathcal{L}_0$ and $\mathcal{L}_1$ are subfamilies of regular languages. A very interesting line of work would consist in extending all the results presented in this paper in order to include pattern representations based on other

more expressive subfamilies of regular languages. By doing this, we could obtain not only new grammar inference algorithms but also new grammar learners that would ensure the consistency of the inferred model wrt. the underlying distance, something which does not happen when traditional grammar learners are applied.

5 Acknowledgments

This work was partially supported by the EU (FEDER) and the Spanish Government MEC/MICINN, under grant TIN 2007-68093-C02, the Spanish project "Agreement Technologies"(CONSOLIDER-INGENIO CSD2007-00022) and the Valencian project PROMETEO/2008/051.

References

1. A.F. Bowers, C. G. Giraud-Carrier, and J. W. Lloyd. Classification of individuals with complex structure. In *Proc. of the 17th International Conference on Machine Learning (ICML'00)*, pages 81–88. Morgan Kaufmann, 2000.
2. G. A. Edgar. *Measure, Topology and Fractal Geometry.* Springer-Verlag, 1990.
3. V. Estruch. Bridging the gap between distance and generalisation: Symbolic learning in metric spaces. PhD Thesis, DSIC-UPV http://www.dsic.upv.es/ vestruch/thesis.pdf, 2008.
4. V. Estruch, C. Ferri, J. Hernández-Orallo, and M.J. Ramírez-Quintana. Distance based generalisation. In *Proc. of the 15th Int. Conf. on ILP*, volume 3625 of *LNCS*, pages 87–102, 2005.
5. V. Estruch, C. Ferri, J. Hernández-Orallo, and M.J. Ramírez-Quintana. Distance based generalisation for graphs. In *Proc. Work. of Machine and Learning with Graphs*, pages 133–140, 2006.
6. V. Estruch, C. Ferri, J. Hernández-Orallo, and M.J. Ramírez-Quintana. Minimal distance-based generalisation operators for first-order objects. In *In Proc. of the 16th Int. Conf. on ILP*, pages 169–183, 2006.
7. C. Ferri, J. Hernández-Orallo, and M.J. Ramírez-Quintana. Incremental learning of functional logic programs. In H. Kuchen and K. Ueda, editors, *FLOPS*, volume 2024 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2001.
8. R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal.*, 26(2):147–160, 1950.
9. J. Hernández-Orallo and M.J. Ramírez-Quintana. Inverse narrowing for the induction of functional logic programs. In *1998 Joint Conference on Declarative Programming, APPIA-GULP-PRODE'98, A Coruña, Spain, July 20-23, 1998*, pages 379–392, 1998.
10. J. Hernández-Orallo and M.J. Ramírez-Quintana. A strong complete schema for inductive functional logic programming. In *Proceedings of the 9th International Workshop on Inductive Logic Programming*, volume 1634 of *Lecture Notes in Artificial Intelligence*, pages 116–127. Springer-Verlag, 1999.
11. V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady.*, 10:707–710, 1966.
12. J. W. Lloyd. Learning comprehensible theories from structured data. In *Advanced lectures on machine learning*, pages 203–225. Springer-Verlag, 2003.

13. S. H. Muggleton. Inductive logic programming: Issues, results, and the challenge of learning language in logic. *Artificial Intelligence*, 114(1–2):283–296, 1999.
14. R. Olsson. Inductive functional programming using incremental program transformation. *Artifificial Intelligence*, 74(1):55–81, 1995.
15. J. Rissanen. Hypothesis selection and testing by the MDL principle. *The Computer Journal*, 42(4):260–269, 1999.
16. U. Schmid. *Inductive synthesis of Functional Programs-Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*. Springer, 2003.
17. S.H. Swamidass, J. Chen, J. Bruand, P. Phung, L. Ralaivola, and P. Baldi. Kernels for small molecules and the prediction of mutagenecity, toxicity and anti-cancer activity. *Bioinformatics*, 21:359–368, 2005.
18. R. Rivest T.H. Cormen, C. Leiserson and C. Stein, editors. *Introduction to Algorithms*. The MIT Press, 2000.
19. C. S. Wallace and D. L. Dowe. Minimum Message Length and Kolmogorov Complexity. *Computer Journal*, 42(4):270–283, 1999.