Analogy on Sequences : a Definition and an Algorithm

Laurent Miclet, Arnaud Delhay

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Analogy on Sequences : a Definition and an Algorithm

Laurent Miclet — Arnaud Delhay

N° 4969

Octobre 2003

THÈME 3

ISSN 0249-6399

Analogy on Sequences : a Definition and an Algorithm

Laurent Miclet*, Arnaud Delhay †

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet CORDIAL

Rapport de recherche n° 4969 — Octobre 2003 — 22 pages

Abstract: We present a definition of analogy on sequences which is based on two principles : the definition of an analogy between the letters of an alphabet and the use of the edit distance between sequences. Our definition generalizes the algorithm given by Lepage and is compatible with another definition of analogy in sequences given by Yvon.

Key-words: Analogy, Analogical Equation, Sequences, Edit Distance, Lazy Learning, Learning by Analogy

* Laurent.Miclet@enssat.fr
† Arnaud.Delhay@univ-rennes1.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Analogie sur les séquences : une définition et un algorithme

Résumé : Nous présentons une définition de l'analogie sur des séquences basée sur deux principes : la définition d'une analogie entre lettres d'un alphabet et l'utilisation de la distance d'édition entre les séquences. Notre définition généralise l'algotithme proposé par Lepage et est compatible avec une autre définition de l'analogie dans les séquences donnée par Yvon.

Mots-clés : Analogie, Equation Analogique, Séquences, Distance d'édition, Apprentissage faiblement supervisé, Apprentissage par analogie

Introduction : Analogy and Sequence mining

As any subfield of data mining, sequence mining has to use both supervised and unsupervised methods of machine learning. We present in this paper a lazy supervised method of learning in the universe of sequences. We assume that there exists a supervised learning set of sequences, composed of sequences associated with class labels. When a new sequence is introduced, a supervised learning algorithm has to infer which label to associate with this sequence.

Lazy learning is a general paradigm that makes no parametric assumption on the data and uses only the learning set. The simplest lazy learning technique is the *nearest neighbour* algorithm : the label attributed to the new sequence is that of the nearest sequence in the learning set. It requires a definition of a distance (or merely of a dissemblance) between sequences.

Analogy is a more complex lazy learning technique, since it is necessary to find three sequences in the learning set and makes use of a more sophisticated argument. Let X be the sequence to which we want to give a label. We have to find three sequences A, B and C in the learning set, with labels $L(A)$, $L(B)$ and $L(C)$, such that "A is to B as C is to X". Then the label of X will be computed as "$L(X)$ is to $L(C)$ as $L(B)$ is to $L(A)$". This is for example the way that we can guess that the past of the verb "to grind" is "ground" knowing that the past of the verb "to bind" is "bound".

Learning by analogy in sequences requires to give a sound definition to the terms "is to" and "as". This is the primary goal of this paper, which is organized as follows. In section 1 we will define more precisely what is analogy, especially on sequences. In section 2 we will give a definition based on a distance between sequences known as the "edit distance". Section 3 will be devoted to explaining an original algorithm which transforms analogies on sequences into analogies on alphabets. Section 4 will discuss this latter problem in algebraic terms. Finally, we will compare our proposition with related works and show that it generalizes previous algorithms. The final section discusses operational problems of this new learning method.

1 What is analogy on sequences

1.1 Analogy

Analogy is a way of reasoning which has been studied throughout the history of philosophy and has been widely used in Artificial Intelligence and Linguistics. Lepage ([7], in French, or [8], in English) has given an extensive description of the history of this concept and its application in science and linguistics.

An analogy between four objects or concepts : `A`, `B`, `C` and `D` is usually expressed as follows : "`A` *is to* `B` *as* `C` *is to* `D`". Depending on what the objects are, analogies can have very different significations. For example, natural language analogies could be : "`a crow` *is to* `a raven` *as* `a merlin` *is to* `a peregrine`" or "`vinegar` *is to* `bordeaux` *as* `a sloe` *is to* `a`

cherry". These analogies are based on the *semantics* of the words. By contrast, in the abstract universe of sequences, analogies such as "abcd *is to* abc *as* abbd *is to* abb" or "g *is to* hh *as* gg *is to* ggg" are *syntactic*.

Whether syntactic or not, all the examples above show the intrinsic ambiguity in defining an analogy. Some would have good reasons to prefer this : "g *is to* hh *as* gg *is to* gggg" or "g *is to* hh *as* gg *is to* hhg". Obviously, such ambiguities are inherent in semantic analogies, since they are related to the meaning of words (the concepts are expressed through natural language). Hence, it seems easier to focus on formal syntactic properties. But resolving syntactic analogies is also an operational problem in several fields of linguistics, such as morphology and syntax, and provides a basis to learning and data mining by analogy in the universe of sequences.

The first goal of this paper is to give a sound definition of a syntactic analogy between sequences of letters.

1.2 Analogies and analogical equations on sequences

In this section, we focus on concepts that are sequences of letters in a finite alphabet and we are interested in studying what is an analogy on these concepts.

Our development will be based on two basic ideas. Firstly, we will formalize the comparison of sequences through the classical notion of *edit distance* between sequences; we will give an algorithm in section 3 which will be proved to transform the question of analogy between *sequences* into that of analogy between *letters*. This is why we will introduce as a second concept an algebraic definition of the analogy between letters, or more generally between the elements of a finite set (section 4).

Unlike D. Hofstadter *et al.*([3]), we do not *a priori* consider as correct the following analogy : "abc *is to* abd *as* ghi *is to* ghj", since we assume that the alphabet of the sequences is simply a finite set of letters, with no order relation. In that, we will stick to the classical definition of an alphabet in language theory. Of course, adding properties on an alphabet increases the possible number of ambiguities in resolving analogies. If we want to give a sound algorithmic definition, we have to focus our interest on problems with the lowest number of free parameters.

We denote now "A *is to* B *as* C *is to* D" by the equation $A : B \doteq C : D$ and we say informally that *solving an analogical equation* is finding one or several values for X from the relation $A : B \doteq C : X$. We will give a more precise definition at the end of this section.

The classical definition of $A : B \doteq C : D$ as an analogical equation requires the satisfaction of two axioms, expressed as equivalences of this primitive equation with two others equations ([8]) :

Symmetry of the 'as' relation : $C : D \doteq A : B$

Exchange of the means : $A : C \doteq B : D$

As a consequence of these two primitive axioms, five other equations are easy to prove equivalent to $A : B \doteq C : D$:

$$\begin{array}{rl} \textit{Inversion of ratios}: & B : A \doteq D : C \\ \textit{Exchange of the extremes}: & D : B \doteq C : A \\ \textit{Symmetry of reading}: & D : C \doteq B : A \\ & B : D \doteq A : C \\ & C : A \doteq D : B \end{array}$$

Another possible axiom (*determinism*) requires that one of the following trivial equations has a unique solution (the other being a consequence) :

$$\begin{array}{lcl} A : A \doteq B : X & \Rightarrow & X = B \\ A : B \doteq A : X & \Rightarrow & X = B \end{array}$$

We can give now a definition of a solution to an analogical equation which takes into account the axioms of analogy.

Definition 1. *X is a* correct solution *to the analogical equation*

$$A : B \doteq C : X$$

if X is a solution to this equation and is also a solution to the two others equations :

$$C : X \doteq A : B$$

$$A : C \doteq B : X$$

2 The edit distance between two sequences

In this section, we give more notations and definitions in elementary language theory ([1]) and we recall what is the *edit distance* between sequences.

2.1 Basic definitions

Definition 2. *Let Σ be a finite set that we will call an* alphabet*. We call* letters *$a, b, c, ...$ the elements of Σ. We denote $u, v, \ldots$ or $A, B, \ldots$ the elements of Σ^*, called* sequences *or* sentences *or* words*.*

- *A sequence $u = u_1 u_2 \ldots u_{|u|}$ is an ordered list of letters of Σ ; its length is denoted $|u|$.*
- *ϵ, the* empty sequence*, is the sequence of null length.*
- *The sentence $u = u_1 u_2 \ldots u_{|u|}$ and the sequence $v = v_1 v_2 \ldots v_{|v|}$ can be* concatenated *to give a new sentence $uv = u_1 u_2 \ldots u_{|u|} v_1 v_2 \ldots v_{|v|}$. The concatenation operation is associative and generally not commutative. The null element for this operation is ϵ.*
- *u is a* prefix *of v if there exists w such as $uw = v$.*
- *z is a* subword *of the sequence x if it exists u and t in Σ^* such that : $x = uzt$.*

A subword z of the sequence x is a special type of *subsequence* of x, since we can also extract sequences from x in which the letters have not to be contiguous in x. The general definition of a subsequence is rather tedious to write, but quite easy to understand on an example :

> `angle` is a subword (and a subsequence) of `triangle`, but `tingle` is only a subsequence of `triangle`.

2.2 Replacement of letters and transformation of sequences.

Firstly we define a positive function δ on $\Sigma \cup \{\epsilon\} \times \Sigma \cup \{\epsilon\} \longmapsto \mathbb{R}$, which can be interpreted as the *replacement cost* of a letter by another letter, the term replacement including the *deletion* (replacement of a letter of Σ by ϵ), the *insertion* (replacement of ϵ by a letter of Σ) and the *substitution* (replacement of a letter of Σ by another letter of Σ or by itself). Substituting a letter by itself will always have a null cost.

> For example, substituting in $x = 00011$ the letter $x_3 = 0$ by the letter 1 produces the sequence 00111, with a cost of $\delta(0, 1)$.
>
> Deleting x_4 in x produces the sequence 0001, with a cost of $\delta(1, \epsilon)$.
>
> Inserting the letter 0 immediatly after x_2 in x produces the sequence 000011, with a cost of $\delta(\epsilon, 0)$.

We can now apply several replacements to a sequence, and it is easy to see that a finite number of replacements can transform any sequence x into any sequence. There are an infinite number of such possibilities. These remarks lead to the following definitions.

Definition 3. *A* transformation *of x to y is a series of replacements changing the sequence x into the sequence y. The* cost *of this transformation is the sum of the costs of the replacements that compose the transformation.*

The edit distance *$D(x, y)$ is the cost of the transformation of x into y which has the lowest cost.*

We will see in section 2 that the word "distance" is used here in its mathematical meaning[1].

[1] A distance Δ on E is a mapping $E \times E \longmapsto \mathbb{R}^+$ with the properties :

- $\forall x, y \in \Sigma, \Delta(x, y) \geq 0$ (positivity)
- $\Delta(x, y) = 0 \Longleftrightarrow x = y$
- $\forall x, y \in \Sigma, \Delta(x, y) = \Delta(y, x)$ (symmetry)
- $\forall x, y, z \in \Sigma \Delta(x, y) \leq \Delta(x, z) + \Delta(z, y)$ (triangular inequality)

2.3 Traces as a particular case of transformation.

To compute the edit distance, we have to give more definitions and quote a theorem, demonstrated by Wagner and Fisher in [12]. The first definition is the classical definition of a *trace* ([12]) between two sequences x and y.

Definition 4. *A* trace *is a transformation from x to y with the following constraints :*

- *every letter x_i and every letter y_j is concerned by at most one substitution.*
- *if (x_i, y_j) and (x_k, y_l) are two couples of letters in x and y and there is a substitution of x_i to y_j and of x_k to y_l, then : $i < k \Rightarrow j < l$*

For example, the transformation of $x = abef$ into $y = acde$ is a trace if it is realized with the following sequence of substitutions, insertions and deletions :

$$\begin{array}{ccccccc} x & = & a & b & & e & f \\ & & | & | & & | & \\ y & = & a & c & d & e & \end{array}$$

Introducing the insertions and deletions, we obtain a complete and better representation of this trace with all the replacements :

$$\begin{array}{ccccccc} x^1 & = & a & b & \epsilon & e & f \\ & & | & | & | & | & | \\ y^1 & = & a & c & d & e & \epsilon \end{array}$$

Note that the two words have now the same length, if we count the number of ϵ that have been included in x and y. In the following, we will denote $x^1 = ab\epsilon bf$ and $y^1 = acde\epsilon$ the two sentences created by the optimal trace between x and y. The sentences x and x^1, on one hand, and y and y^1, on the other hand, have the same semantics in language theory.

Now we can give a slightly modified definition of a trace, which will be more easy to use.

Definition 5. *A* trace *is a transformation from x^1 to y^1, obtained from x and y by possibly inserting some ϵ, with the following constraints :*

- *every letter x_i^1 and every letter y_j^1 is concerned by exactly one substitution*
- *if (x_i^1, y_j^1) is a couple of letters in x^1 and y^1 and there is a substitution of x_i^1 to y_j^1 , then : $i = j$*

The following theorem ([12]), states that the only transformations to be considered for computing the edit distance are the traces.

Theorem 1. *If δ is a distance, then the* edit distance *can be computed as the cost of a* trace *with the lowest cost than transforms x into y.*

A trace corresponding to the edit distance, that of lowest cost, is often called *optimal*. It may not be unique.

It is now possible to give the classical Wagner and Fisher algorithm ([12]) which computes the edit distance and the optimal trace.

2.4 Computing the edit distance and finding an optimal trace

We take two sentences x, with $|x| = n$ and y, with $|y| = m$. The algorithm to compute the edit distance $D(x, y)$ between the two sequences x and y takes as input the distance δ on $\Sigma \cup \epsilon$ and fills up a matrix $M(0:n, 0:m)$ of size $(n+1) \times (m+1)$. The element $M(i, j)$ is defined as :

$$M(i, j) = D(x(1:i), y(1:j))$$

where $x(1:i)$ denotes the prefix of length i of x and $y(1:j)$ denotes the prefix of length j of y.

The algorithm is based on a dynamic programming recurrence and is described in the next figure. Its time complexity is in $\mathcal{O}(n \times m)$.

Algorithm 1 Computing the edit distance $D(x, y)$ between two sequences x and y.

begin
$M(0, 0) \leftarrow 0$;
for $i = 1, m$ **do**
  $M(i, 0) \leftarrow \sum_{k=1}^{k=i} \delta(x_k, \epsilon)$;
end for
for $j = 1, n$ **do**
  $M(0, j) \leftarrow \sum_{k=1}^{k=j} \delta(\epsilon, y_k)$;
end for
for $i = 1, m$ **do**
  for $j = 1, n$ **do**

$$M(i, j) \leftarrow \min \begin{cases} M(i-1, j) + \delta(x_i, \epsilon) \\ M(i, j-1) + \delta(\epsilon, y_j) \\ M(i-1, j-1) + \delta(x_i, y_j) \end{cases}$$

  end for
end for
$D(x, y) \leftarrow M(m, n)$
end

A consequence of this algorithm is the following remarkable result([2]), which justifies the name of edit *distance* :

Theorem 2. *If δ is a distance on $(\Sigma \cup \{\epsilon\})$ then D is also a distance on Σ^*.*

The algorithm can be completed to construct x^1 and y^1 from x and y and to produce the optimal trace, or all the optimal traces if there are more than one. This is easily done by memorizing more information during the computation and by backtracking on an optimal path in the final matrix ([11]).

2.5 A degenerate case : the edit distance and the longest common subsequence

If the replacement of a letter is restricted to the insertion, the deletion and the substitution of a letter by itself (i.e. the substitution of a letter by another is forbidden), the Wagner and Fisher algorithm is exactly equivalent to another algorithm on strings which finds the *longest common subsequence* of two sentences[2]. Once again, an example is better than a formal definition to understand what happens in that case.

Let $x = abdefgh$ and $y = bcdfhigc$. The longest common subsequence of x and y is : $lcs(x, y) = bdfg$.

The trace corresponding to $lcs(x, y)$ has no substitution of one letter by another different letter :

$$\begin{array}{cccccccccc} a & b & \epsilon & d & f & \epsilon & \epsilon & g & \epsilon & h \\ | & | & | & | & | & | & | & | & | & | \\ \epsilon & b & c & d & f & h & i & g & c & \epsilon \end{array}$$

The edit distance between x and y can be analytically expressed in that case by the following formula ([11]):

$$D(x, y) = |x| + |y| - 2|lcs(x, y)|$$

This particular distance[3] is used by both Lepage([5]) and Yvon ([9]) for defining analogy between sequences.

3 An algorithm for solving analogical equations on sequences

We propose in this section an algorithm for solving analogical equations on sequences, based on two concepts : firstly the notion of *edit distance*, as seen in the last section. Secondly, we will see that this algorithm uses the hypothesis that there exists a correct solution for analogical equations on the alphabet of the sequences. This point will be discussed in section 4. We will also show how our algorithm can be considered as a generalization of that of Lepage ([5]) and how it inserts into the language theory framework of Yvon ([13], [9]).

3.1 Presentation of the algorithm

Let A, B, and C be three sequences on the same alphabet Σ. We want to find a solution X to the analogical equation : $A : B :: C : X$.

We assume that the optimal trace between A and B, corresponding to their edit distance has been established, producing two new sequences A^1 and B^1 of same length. We also

[2] Or, in a equivalent manner, in giving to the substitution of a letter by another letter a cost superior to the sum of the destruction of the first letter and the cost of the insertion of the second. In that case, δ is no more a distance, because it does not verify the triangular inequality.

[3] It is actually a distance in the mathematical meaning of the word.

assume that it is unique (we will relax this constraint in section 3.4). A^1 and B^1 are semantically equal to A and B, some ϵ being possibly inserted according to the optimal trace, as showed in section 2.3.

In the same manner, we assume that A and C have been processed and that the optimal trace has been established, giving two new sentences A^2 and C^1 of same length.

The result of the new algorithm that we give here will be three sequences, denoted A^3, B^3 and C^3, of same length, such that A^3 is semantically equal to A (it is constructed in inserting some ϵ in A^2) and B^3 is semantically equal to B. Every triple of letters with the same index, $A^3[l]$, $B^3[l]$ and $C^3[l]$ will correspond to an elementary analogical equation on letters. More formally :

- $|A^3| = |B^3| = |C^3|$
- Each couple of symbols of $\Sigma \cup \{\epsilon\}$ in A^3 and B^3 which have the same rank in A^3 and B^3 are linked by a replacement in the optimal trace between A and B or are a couple (ϵ, ϵ).

We will firstly give the algorithm and then run an example. Then we will show that if we know how to solve analogical equations on letters, this algorithm ensures that we know how to solve analogical equations on sequences of letters. Moreover, we will show that the algorithm by Lepage ([5]) is a particular case of our algorithm and that our algorithm is also compatible with the theory of analogy on sequences by Yvon ([13]).

3.2 The algorithm

For readibility, we use now the following notations : the i^{th} letter of a sequence S is denoted $S[i]$. The index i will be used for the sequence A^1, the index i' for the sequence A^2, the index j for the sequence B^1 and the index k for the sequence C^1.

There are in principle sixteen different situations corresponding to the fact that $A^1[i]$, $A^2[i']$, $B^1[j]$ and $C^1[k]$ may be equal to ϵ or not. Only eleven are actually met, as a consequence of the fact that A^1 and A^2 are explored together : whenever the algorithm meets $A^1[i] \neq \epsilon$ and $A^2[i'] \neq \epsilon$, then $A^1[i]$ is the same letter as $A^2[i']$. In this case, we notice that there are four common situations for which we do the same operations, that is :

- storing the values of $A^1[i] = A^2[i']$ in $A^3[l]$, of $B^1[j]$ in $B^3[l]$, of $C^1[k]$ in $C^3[l]$, and
- incrementing all indices.

The situation where $A^1[i] = A^2[i'] = \epsilon$ and $B^1[j] \neq \epsilon$ and $C^1[k] \neq \epsilon$ is a particular case. This situation corresponds to an insertion from $A^1[i]$ into $B^1[j]$ and from $A^2[i']$ into $C^1[k]$ and we have to keep both ϵ form the two insertions. As a consequence, we treat this situation in two steps by considering one insertion after the other. But the choice of the first insertion to treat can lead to wrong results. The idea is to prioritize the insertion that will respect chunks in the sequences. To achieve this goal, we look forward in the remaining subsequences of A^1 and A^2 the next common symbol that is not ϵ and treat the sequence

Algorithm 2 Computing the solution to an analogical equation on sequences.

begin
$i \leftarrow 1$; $i' \leftarrow 1$;
$j \leftarrow 1$; $k \leftarrow 1$; $l \leftarrow 1$.
while $(A^1[i] \neq \epsilon)$ OR $(A^2[i'] \neq \epsilon)$ OR $(B^1[j] \neq \epsilon)$ OR $(C^1[k] \neq \epsilon)$ **do**
  According to the values of $A^1[i]$, $A^2[i']$, $B^1[j]$ and $C^1[k]$,
    assign a value to $A^3[l]$, $A^3[l]$, $B^3[l]$ and $C^3[l]$
    and increase the values of i, i', j and k by 0 or 1
  $l \leftarrow l + 1$
end while
for $l = 1, |A^3|$ **do**
  Solve on the alphabet the analogical equation :
    $A^3[l] : B^3[l] \doteq C^3[l] : X^3[l]$
end for
end

where this letter is replaced by itself rather than the sequence where it is a real substitution (by another letter).

Finally we actually get only four different cases in this algorithm that are described in the following list :

Case 1.
We have $A^1[i] = A^2[i']$.
$A^3[l] \leftarrow A^1[i] = A^2[i']$; $B^3[l] \leftarrow B^1[j]$; $C^3[l] \leftarrow C^1[k]$.
$i \leftarrow i + 1$; $i' \leftarrow i' + 1$; $j \leftarrow j + 1$; $k \leftarrow k + 1$; $l \leftarrow l + 1$.

Case 2.
We have $A^1[i] = \epsilon$ and $A^2[i'] \neq \epsilon$.
$A^3[l] \leftarrow \epsilon$; $B^3[l] \leftarrow B^1[j]$; $C^3[l] \leftarrow \epsilon$.
$i \leftarrow i + 1$; $j \leftarrow j + 1$; $l \leftarrow l + 1$.

Case 3.
We have $A^1[i] \neq \epsilon$ and $A^2[i'] = \epsilon$.
$A^3[l] \leftarrow \epsilon$; $B^3[l] \leftarrow \epsilon$; $C^3[l] \leftarrow C^1[k]$.
$i' \leftarrow i' + 1$; $k \leftarrow k + 1$; $l \leftarrow l + 1$.

Case 4.
We have $A^1[i] = \epsilon$ and $A^2[i'] = \epsilon$ and $B^1[j] \neq \epsilon$ and $C^1[k] \neq \epsilon$.
Choose the sequence to treat (where next symbol $\neq \epsilon$ is replaced by itself)

- If A^1 is chosen : $A^3[l] \leftarrow \epsilon$; $B^3[l] \leftarrow B^1[j]$; $C^3[l] \leftarrow \epsilon$.
  $i \leftarrow i + 1$; $j \leftarrow j + 1$; $l \leftarrow l + 1$.
- If A^2 is chosen : $A^3[l] \leftarrow \epsilon$; $B^3[l] \leftarrow \epsilon$; $C^3[l] \leftarrow C^1[k]$.
  $i' \leftarrow i' + 1$; $k \leftarrow k + 1$; $l \leftarrow l + 1$.

The following table represents the dispatching of all the situations on the three cases :

Number of the case	$A^1[i] \neq \epsilon$ $B^1[j] \neq \epsilon$	$A^1[i] \neq \epsilon$ $B^1[j] = \epsilon$	$A^1[i] = \epsilon$ $B^1[j] \neq \epsilon$	$A^1[i] = \epsilon$ $B^1[j] = \epsilon$
$A^2[i'] \neq \epsilon$ $C^1[k] \neq \epsilon$	1	1	2	
$A^2[i'] \neq \epsilon$ $C^1[k] = \epsilon$	1	1	2	
$A^2[i'] = \epsilon$ $C^1[k] \neq \epsilon$	3	3	4	End of algo
$A^2[i'] = \epsilon$ $C^1[k] = \epsilon$			End of algo	

3.3 Running an example

Let $A = wolf$, $B = wolves$, $C = leaf$

We do not know the distance δ on the alphabet, but it does not matter. We assume that, for example, the sequences A^1 and A^2 and the optimal trace between A and B and A and C are as follows :

$$\begin{array}{lcccccc} A^1 = & w & o & l & \epsilon & f & \epsilon \\ & | & | & | & | & | & | \\ B^1 = & w & o & l & v & e & s \end{array} \qquad \begin{array}{lccccc} A^2 = & w & o & l & \epsilon & f \\ & | & | & | & | & | \\ C^1 = & \epsilon & l & e & a & f \end{array}$$

Running the algorithm will take 7 steps and produce three sequences A^3, B^3 and C^3 of length 7.

Initialisation : $i \leftarrow 1$; $i' \leftarrow 1$; $l \leftarrow 1$

Step 1 : $A^1[i] = w$, $A^2[i'] = w$, $B^1[i] = w$, $C^1[i'] = \epsilon$
Case 1 : $i \leftarrow 2$; $i' \leftarrow 2$; $l \leftarrow 2$

Step 2 : $A^1[i] = o$, $A^2[i'] = o$, $B^1[i] = o$, $C^1[i'] = l$
Case 1 : $i \leftarrow 3$; $i' \leftarrow 3$; $l \leftarrow 3$

Step 3 : $A^1[i] = l$, $A^2[i'] = l$, $B^1[i] = l$, $C^1[i'] = e$
Case 1 : $i \leftarrow 4$; $i' \leftarrow 4$; $l \leftarrow 4$

Step 4 : $A^1[i] = \epsilon$, $A^2[i'] = \epsilon$, $B^1[i] = v$, $C^1[i'] = a$
Case 4 : look for the first letter $\neq \epsilon$. For C, no substitution then go on with C.
$i \leftarrow 4$; $i' \leftarrow 5$; $l \leftarrow 5$

Step 5 : $A^1[i] = \epsilon$, $A^2[i'] = f$, $B^1[i] = v$, $C^1[i'] = f$
Case 2 : $i \leftarrow 5$; $i' \leftarrow 5$; $l \leftarrow 6$

Step 6 : $A^1[i] = f$, $A^2[i'] = f$, $B^1[i] = e$, $C^1[i'] = f$
Case 1 : $i \leftarrow 6$; $i' \leftarrow 6$; $l \leftarrow 7$

Step 7 : $A^1[i] = \epsilon$, $A^2[i] = \epsilon$ (finished), $B^1[i] = s$, $C^1[i'] = \epsilon$ (finished)
End of Algo : $i \leftarrow 7$; $i' \leftarrow 7$; $l \leftarrow 8$

End /* End of A^1 and $A^2 \Rightarrow$ End of the algorithm */

Result :

$$\begin{array}{lcccccccc} A^3 & = & w & o & l & \epsilon & \epsilon & f & \epsilon \\ B^3 & = & w & o & l & \epsilon & v & e & s \\ C^3 & = & \epsilon & l & e & a & \epsilon & f & \epsilon \end{array}$$

3.4 The case of several optimal traces

We have noticed that an optimal trace giving the edit distance may not be unique (section 3.1). A simple example of this phenomenon is in the case of the degenerated edit distance : for example, the sentences $x = abcdef$ and $y = bafgc$ have four longest common subsequences of length 2, ac, bf, af and bc.

In that case, the brute force treatment is to enumerate all the p traces and to run the algorithm p times. The algorithm therefore gives all the correct solutions to the analogical equations[4].

3.5 Properties of the algorithm.

We have defined in section 1 X as a *correct solution* to the analogical equation

$$A : B \doteq C : X$$

if X is a solution to this equation and is also a solution to the two others equations :

$$C : X \doteq A : B \qquad \text{and} \qquad A : C \doteq B : X$$

We will see now that our algorithm gives a solution which is coherent with this definition.

Property 1. *Assuming that we know how to correctly solve analogical equations on alphabets, if for every triple a, b, c of letters of $\Sigma \cup \{\epsilon\}$ there exists a correct solution to the analogical equations : $a : b \doteq c : x$ then our algorithm gives a correct solution to any analogical equation on Σ^*.*

This property is straightforward since our algorithm reduces the problem of finding a correct solution to the analogical equation $A : B \doteq C : X$ to that of finding a solution to each of the $|A^3|$ equations of letters : $A^3[l] : B^3[l] \doteq C^3[l] : X[l]$ and that we assume that such analogical equations can be correctly solved.

Property 2. *Assuming that we know how to correctly solve analogical equations on alphabets, if for every triple a, b, c of letters of $\Sigma \cup \{\epsilon\}$ there exists a unique correct solution to the analogical equations : $a : b \doteq c : x$ then the algorithm gives a unique correct solution to any analogical equation on Σ^*.*

The demonstration is also straightforward.

[4]Since the structure of the set of all the optimal traces is actually not a list of independent sentences, but a directed acyclic graph, a better algorithm can be produced. This is work in progress.

Property 3. *If the trace between A^1 and B^1 and the trace between A^2 and C^1 are obtained with the degenerated edit distance (see section 2.5), then :*

- *our algorithm computes the same solution as the algorithm of Lepage.*
- *our algorithm computes the "simplest" analogy, according to the definition by Yvon.*

Since they require the introduction of more material, the demonstrations of these properties are given in section 5, with reference to the quoted works.

4 Analogical equations on alphabets and finite groups

In the previous section, we have given an algorithm for solving analogical equations on strings that is based on the edit distance. We have shown that the only remaining problem is to be able to give a correct solution to an analogy equation on a set composed of a finite alphabet Σ plus the empty string ϵ. As a consequence, the aim of this section is to give an algebraic definition of what is analogy between letters and to find a link between this algebraic structure and the distance δ on $\Sigma \cup \{\epsilon\}$ (*cf* section 2.2).

4.1 An algebraic definition of analogy

To find an algebraic definition of analogy on alphabets, we firstly examine what is an analogy in a vector space and reduce it to a set with a minimum of algebraic properties.

In a vector space, the analogy $\overrightarrow{OA} : \overrightarrow{OB} \doteq \overrightarrow{OC} : \overrightarrow{OX}$ is quite naturally solved by choosing X as the fourth summit of the parallelogram built on A, B and C (figure 1). Obviously, this construction verifies all the axioms of a correct solution to the analogical equation.

A parallelogram has two equivalent definitions : either

$$\overrightarrow{AB} = \overrightarrow{CX} \qquad (\text{or : } \overrightarrow{AC} = \overrightarrow{BX})$$

or

$$\overrightarrow{OA} + \overrightarrow{OX} = \overrightarrow{OB} + \overrightarrow{OC}$$

Let us firstly consider the second definition; we will come back to the first one later. If we want to tranfer it into an alphabet Σ, we can define in the same manner an operator $\oplus$ from $\Sigma \cup \{\epsilon\} \times \Sigma \cup \{\epsilon\}$ to a set F such that :

$$a \oplus x = b \oplus c \Leftrightarrow a : b \doteq c : x$$

It is not important what F is for the moment, we will examine this question in the next subsection. We want to define what our operator is and what structure it gives to the set $\Sigma \cup \{\epsilon\}$. The aim is to find how to build an analogical operator $\oplus$ on every alphabet, that is to give the table describing $\oplus$ on $\Sigma \cup \{\epsilon\}$.

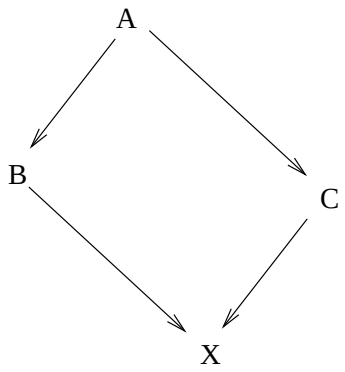

Figure 1: Analogy in a vector space.

4.2 Properties of analogy

We have given in section 1 the axioms of analogy as described by Lepage. Let us rewrite them with the operator $\oplus$. For each axiom, we will exhibit a corresponding algebraic property and this will allow us to determine more precisely what properties the operator $\oplus$ must have.

4.2.1 Symmetry.

The symmetry is expressed as follows :

$$a : b \doteq c : d \Rightarrow c : d \doteq a : b$$

that is

$$a \oplus d = b \oplus c \Rightarrow c \oplus b = d \oplus a$$

The symmetry alone does not help us to refine the definition of $\oplus$ for the moment.

4.2.2 Exchange of the means.

Exchange of the means is expressed as follows :

$$a : b \doteq c : d \Rightarrow a : c \doteq b : d$$

and can be transcripted as :

$$a \oplus d = b \oplus c \Rightarrow a \oplus d = c \oplus b$$

From this assumption, we can deduce that our operator $\oplus$ must be commutative, as $c \oplus b = b \oplus c$ if $a : b \doteq c : d$. More over, we notice that the equation concerning symmetry of analogy is always true because of the commutativity of the operators $\oplus$ and $=$.

4.2.3 Determinism.

Determinism (which is a facultative property, but expresses a common sense solution for elementary analogical equations) is expressed as follows :

$$a : a \doteq c : x \Rightarrow x = c$$

$$a : b \doteq a : x \Rightarrow x = b$$

It can be written with $\oplus$ as :

$$a \oplus x = a \oplus c \Rightarrow x = c$$

$$a \oplus x = b \oplus a \Rightarrow x = b$$

The first equation expresses the algebraic property of *left regularity*. Because of the commutativity, we can say that $\oplus$ must also be *right regular*, thus simply *regular*.

4.3 Construction of an operator and a distance

4.3.1 The alphabet as a finite group.

We already know that our operator $\oplus$ must be commutative and regular. In addition to these properties, we want to solve some cases in analogy that Lepage cannot handle .

One of these cases is to find a solution to the analogical equation : $a : \epsilon \doteq \epsilon : x$, which can be expressed as : $a \oplus x = \epsilon \oplus \epsilon$. If we consider that $\oplus$ is an internal composition operator (*i.e.* $F = \Sigma \cup \{\epsilon\}$) and ϵ is the null element of $\Sigma \cup \{\epsilon\}$ for $\oplus$, then we transform the above expression into : $a \oplus x = \epsilon \oplus \epsilon = \epsilon = x \oplus a$.

If we consider that every element in $\Sigma \cup \{\epsilon\}$ has a symmetric, then every equation of this form has a solution which is the symmetric of a. Assuming that $(\Sigma \cup \{\epsilon\}, \oplus)$ is a *group*[5] is sufficient to get these properties. Moreover, we know that this group is *abelian* since $\oplus$ is commutative.

4.3.2 An example : the additive cyclic group.

We take as an example the cyclic finite abelian group. The table that describes $\oplus$ in this case is given at Figure 2. It enables us to solve every analogical equation on letters.

For example, this table brings the unique solution $x = b$ to the analogical equation :

$$a : d \doteq c : x$$

[5] $(G, \oplus)$ is a group if :

- $\oplus$ is an internal composition operator (stability of G with $\oplus$)
- $\oplus$ is associative : $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
- there exists a null element, called ϵ
- every element of G has a symmetric : $\forall x \in G, \exists! \bar{x} / x \oplus \bar{x} = \bar{x} \oplus x = \epsilon$

since

$$a \oplus d = c \oplus x \Rightarrow x = b$$

The case quoted at the beginning of section 4.3.1 :

$$a : \epsilon \doteq \epsilon : x$$

is now solved as $x = f$, since :

$$a \oplus f = \epsilon \oplus \epsilon = \epsilon$$

$\oplus$	ϵ	a	b	c	d	e	f
ϵ	ϵ	a	b	c	d	e	f
a	a	b	c	d	e	f	ϵ
b	b	c	d	e	f	ϵ	a
c	c	d	e	f	ϵ	a	b
d	d	e	f	ϵ	a	b	c
e	e	f	ϵ	a	b	c	d
f	f	ϵ	a	b	c	d	e

Figure 2: A table for an analogical operator on an alphabet of 6 elements plus ϵ, seen as the additive cyclic group $\mathcal{G}_7$.

4.3.3 A distance on the additive cyclic group.

We also must be able to build a distance on the alphabet, since it is necessary to compute the edit distance between strings of letters when using the Wagner and Fisher algorithm (section 2.4).

In the case of the additive cyclic group, we have a table of analogy (figure 2). From this table, it is possible to deduce some properties of the distance by considering the following equation :

$$(a : b \doteq c : d) \Leftrightarrow (a \oplus d = b \oplus c) \Rightarrow \delta(a, b) = \delta(c, d)$$

This equation is coherent with the first characterization of a parallelogram, as given in section 4.1. On the example on figure 2, we can see that any result of $x \oplus y$ appears three times in the up-right triangle of the table (bottom-left triangle is the symmetric). As a consequence of the above equation, we deduce three equations for each value of the table.

This leads to a system of constraints on the values of the distance table, which is easy to solve. There are several solutions. The distance related to the analogy defined by table of figure 2 is necessarily of the form represented on figure 3. The table has only $\lfloor \frac{n}{2} \rfloor$ different values and is a symmetrical Toeplitz matrix.

δ	ϵ	a	b	c	d	e	f
ϵ	0	α	β	γ	γ	β	α
a	α	0	α	β	γ	γ	β
b	β	α	0	α	β	γ	γ
c	γ	β	α	0	α	β	γ
d	γ	γ	β	α	0	α	β
e	β	γ	γ	β	α	0	α
f	α	β	γ	γ	β	α	0

Figure 3: The discrete distance δ on the finite group $\mathcal{G}_7$, with $\alpha > 0$, $\beta > 0$ and $\gamma > 0$.

It is important to note that we have no proof yet that this table is a distance table. Table values of the variables have to verify the triangular inequality (positivity, symmetry and identity are verified if the values are positive). This work is in progress.

But there is a way to get a distance table : by using a geometrical representation in $\mathbb{R}^2$, in which the letters are regularly placed on a circle (see figure 4) and by defining the distance between letters as the euclidian distance in $\mathbb{R}^2$. This distance table is coherent with figure 3.

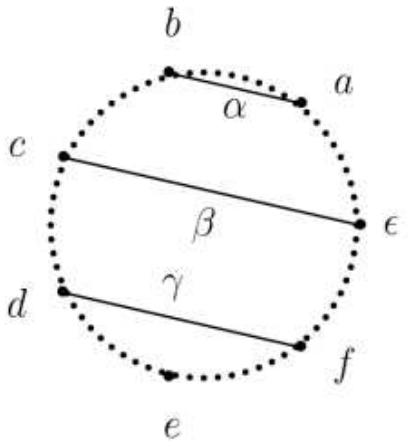

Figure 4: Representing $\mathcal{G}_7$, the additive cyclic finite group with 7 elements, and defining a distance on $\mathcal{G}_7$.

5 Related work

Solving analogical equations between strings has only drawn little attention in the past. Most relevant to our discussion are the works of Yves Lepage, especially [5] and [6], presented in full details in [7] and the very recent work of Yvon.

5.1 The algebraic frame of Yvon

In [13], Yvon considers that comparing sequences for solving analogical equations must be based only on insertions and deletions of letters, and must satisfy the axioms of Lepage.

He first recalls the notion of *shuffle*, as introduced eg. in [10] or [4] as follows.

Definition 6 (Shuffle). *If v and w are two words in $\Sigma^\star$, their shuffle is the language defined as:*

$$v \bullet w = \{v_1 w_1 v_2 w_2 \dots v_n w_n, \text{ with } v_i, w_i \in \Sigma^\star, v = v_1 \dots v_n, w = w_1 \dots w_n\}$$

The shuffle of two words u and v contains all the words w which can be composed using all the symbols in u and v, with the constraint that if a precedes b in u (or in v), then it necessary precedes b in w.

Taking for instance $u = abc$ and $v = def$, the following words: $abcdef$, $abdefc$, $adbecf$, ... all belong to $u \bullet v$; this is not the case with $abefcd$, in which d occurs after, instead of before, e.

Definition 7 (Complementary set). *Let I_y be the set of indices of the word y. If x is a subword of w, the complementary set of x with respect to w is $w\backslash x = \{y, y \in w, x = w(\{0 \dots |w|\}\backslash I_y)\}$. If x is not a subword of w, $w\backslash x$ is empty.*

The complementary subwords of x with respect to w is simply what "remains" of w when the symbols in x are discarded. For instance, the complementary subword of *price* in *priceless* is the set $\{less, elss\}$; the complementary subwords of *ive* in *derivative* is the set: $\{deriv, dervati, derivat\}$. This operation can be turned into a (symetric) binary relationship as follows:

Definition 8 (Complementary relationship). *$w \in \Sigma^\star$ define a binary relationship denoted $w\backslash$ and defined as : $uw\backslash v$ if and only if $u \in w\backslash v$.*

Then, Yvon gives the following definition[6] of the set of solutions to an analogical equation on sequences :

Definition 9 (The set of solutions to an analogical equation).

$$x \text{ is a solution of } u : v = w : x \Leftrightarrow t \in b \bullet c\backslash a$$

[6] This is not given in [13] as a definition, but as a property. In the scope of this paper, and with the agreement of the author, we use it as a definition, actually equivalent to the one which is given in the paper.

Yvon constructs a finite-state transducer to compute the set of a solutions to any analogical equation on strings. He produces also a refinement of this finite state machine able to rank these analogies so as to recover an intuitive preference towards simple analogies. The "simplicity" of an analogy is related to the number of entanglements produced by the shuffle operator. This corresponds to the intuition that good analogies should preserve large chunks of the original objects; the ideal analogy involving identical objects.

For example, the solution $x = reception$ to the analogical equation $defective : defection \doteq receptive : x$ has a simplicity degree of three, since the words cannot be cut in less than three chunks corresponding to the pattern : $abc : aec \doteq gbc : gec$

When the edit distance is calculated only through insertion and deletions, all of same cost, our algorithm finds the simplest analogy, or all the simplest analogies if there are more than one.

In the framework of Yvon, our approach is equivalent to computing the longest common subsequence (and not all the common subsequences). Thus we can consider that we propose an extension that uses one supplementary editing operator : the substitution of a letter by a different letter. Moreover, Yvon says himself in [13] that his work is very similar to Lepage's work :

> "(The Lepage algorithm is) quite similar to ours, and entails a conception of analogies between words which is fully in agreement with our definition".

5.2 An algorithm by Lepage

Lepage ([5]) details the implementation of a analogy solver for words, based on a generalisation of the algorithm computing the longest common subsequence between strings (see section 2.5). Given the equation $u : v \doteq w : x$, this algorithm proceeds as follows: it alternatively scans v and w for symbols in u; when one is found, say in w, it outputs in x the prefix of w which did not match, while extending as much as possible the current match; then exchange the roles of v and w. If u has been entirely matched, then output if necessary in x the remaining suffixes of v and w; otherwise the analogical equation does not have any solution.

To understand this algorithm, consider the following analogy with the domains of integers: suppose we have to find an integer x such that $\frac{u}{v} = \frac{w}{x}$. Assuming that u, v and w are given under the form of their decomposition into a product of primes, the solution obains by discarding in the product $v \times w$ all the primes in u. This is basically what this solver does; the non-commutativity of string concatenation imposing to scan v and w in a specific order.

This algorithm is quite obviously a particular instance of ours : If we restrict our algorithm to the case where the edit distance is calculated only through insertion and deletions, all of same cost, and we assume that there is only one solution to the analogical equation, our algorithm is equivalent to that of Lepage.

Conclusion and Discussion

In this paper, we propose a new algorithm for solving analogy between sequences. This algorithm is built in two steps. The first step consists in building an alignment between the sequences A and B, and A and C at the same time so as the result is three sequences of the same length. The second step is then to solve analogies between symbols of these resulting sequences.

We have given an algebraic structure so as to get a unique solution of all possible combinations of analogical equations with symbols chosen in an alphabet. We have shown that a finite additive cyclic group is a sufficient structure and we give an example of the corresponding operator $\oplus$ on a given set $\Sigma \cup \{\epsilon\}$. We also have shown how to build a distance on this group, needed for alignments at the first step of the algorithm. In the next paragraphs, we propose a discussion about the algebraic structure of analogy and the complexity of our algorithm.

About the algebraic definition of analogy

In section 4, we have defined an operator and given an algebraic structure to represent analogy between symbols. We have said that the choice of a finite cyclic group composed of the alphabet plus ϵ is sufficient to get the desired properties of analogy on symbols. The fact that it is cyclic gives a method to build an analogical operator (and its table) on an alphabet for every cardinal of the alphabet.

Then we have shown on an example a method to build a distance table inspired from the analogical operator by solving equations. These equations are based of the property of triangular inequality, necessary to get a distance, and on the analogies defined by the table got for analogical operator $\oplus$. The idea here is to say that the choice of the cyclic finite group imposes constraints on the construction of the distance table. We also have shown that a particular geometric construction is an example of representation of this analogy and of its associated distance in a two dimensionnal euclidian space.

An extension to this work could focus on several points. The first point is to know if we could examine a more general algebraic structure than a cyclic finite group. We have shown in this paper that it is sufficient, but is there a less restrictive algebraic form for respecting the desired properties of an analogy ? Are there other types of group that respect the analogical properties ? The next step is then to study more precisely the influence of the analogical operator on the distance table. For example, is there an algorithm to build a general solution for building a distance for every alphabet given a analogical table ? Conversely, we know that a uniform distribution of the symbols of the alphabet on a circle in a two dimensionnal euclidian space gives us a distance between symbols, but are we sure that there exists at least one analogical table to define an operator corresponding to this distance ? If we dispose of an *a priori* distance and its table, is there an algorithm to transform this distance into an analogical distance or to built a anolgical distance taht is sufficiently close to the given distance ?

About operational problems of solving analogical equations in sequences

Finally, we discuss the complexity of our algorithm. The first step is to build alignments between sequences in two couples (A^1 and B^1, and A^2 and C^1). This step is twice the complexity of the Wagner and Fisher algorithm, that is $\theta(n.\max(m,p))$ if n, m and p are the lengths of A, B and C respectively. Then the algorithm computes the alignement between the three sequences. This operation is linear in function of the length of the sequences. The second step is to compute the solution of every elementary analogical equation on letters and it is also linear in function of the length of the sequences after the second alignment. Finally the complexity of our algorithm is $\theta(n.\max(m,p))$.

Acknowledgements. The authors thank F. Yvon for his enlighting discussions and the LaTeX code chunks from his papers, and P. Struillou, our algebraic adviser.

References

[1] A. Aho and J. Ullman. *The Theory of Parsing, Translation and Compiling, Vol 1 : Parsing.* Prentice-Hall, 1972.

[2] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithmique du texte.* Vuibert, 2001.

[3] Douglas Hofstadter and the Fluid Analogies Research Group. *Fluid Concepts and Creative Analogies.* Basic Books, New York, 1994.

[4] D. Kozen. *Automata and Computability.* Springer, 1996.

[5] Yves Lepage. Solving analogies on words: an algorithm. In *Proceedings of COLING-ACL'98, Vol 1*, pages 728–735, Montréal, August 1998.

[6] Yves Lepage. Analogy and formal languages. In *Proceedings of FG/MOL 2001*, pages 373–378, Helsinki, August 2001.

[7] Yves Lepage. *De l'analogie rendant compte de la commutation en linguistique.* Grenoble, 2003. Habilitation à diriger les recherches.

[8] Yves Lepage and Shin-Ichi Ando. Saussurian analogy: a theoretical account and its application. In *Proceedings of COLING-96*, pages 717–722, København, August 1996.

[9] Vito Pirrelli and François Yvon. Analogy in the lexicon: a probe into analogy-based machine learning of language. In *Proceedings of the 6th International Symposium on Human Communication*, Santiago de Cuba, Cuba, 1999.

[10] J. Sakarovitch. *Eléments de théorie des automates.* (to be published).

[11] D. Sankoff and J. Kruskal, editors. *Time Warps, String Edits and Macromolecules : the Theoru and Practice of Sequence Comparison.* Addidon-Wesley, 1983.

[12] R.A. Wagner and M.J. Fisher. The string-to-string correction problem. *JACM*, 1974.

[13] F. Yvon. Analogy-based NLP : Implementation issues. Technical report, Ecole Nationale Supérieure des Télécommunications, 2003.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399