

The Efficient Computation of Complete and Concise Substring Scales with Suffix Trees

Sébastien Ferré

Irisa/Université de Rennes 1
Campus de Beaulieu, 35042 Rennes cedex, France
Email: ferre@irisa.fr

Abstract Strings are an important part of most real application multi-valued contexts. Their conceptual treatment requires the definition of *substring scales*, i.e., sets of relevant substrings, so as to form informative concepts. However these scales are either defined by hand, or derived in a context-unaware manner (e.g., all words occurring in string values). We present an efficient algorithm based on suffix trees that produces complete and concise substring scales. Completeness ensures that every possible concept is formed, like when considering the scale of all substrings. Conciseness ensures the number of scale attributes (substrings) is less than the cumulated size of all string values. This algorithm is integrated in Camelis, and illustrated on the set of all ICCS paper titles.

1 Introduction

In information systems, one of the most common datatype is the *string*. For instance, in a bibliographic application, most attributes are string-valued (author names, title, journal or conference name). While these strings usually bring a lot of information, they are hardly exploited in conceptual information systems based on Formal Concept Analysis (FCA) [GW99]. They are most often represented as (1) nominal values, which is right for entry types (e.g., “journal”, “inproceedings”) but uninteresting for titles, (2) a set of keywords given by hand [CS00], or (3) a set of keywords derived in a context-unaware manner, e.g., all title words [FR01].

An important objective of conceptual information systems is to ensure a tight combination of querying and navigation [GMA93]. In this respect, the manual or context-unaware production of keywords is unsatisfactory because they are fully part of the navigation structure, and so should be automatically derived from the context, like the concept lattice. We consider in this paper the automatic derivation of *substring scales*, whose values are full strings (like titles), and whose attributes are substrings (corresponding to keywords). For instance, in the case of the bibliographic context of all ICCS papers, one would expect to have substrings like “Formal Concept Analysis”, “Conceptual Graphs”. These substrings play the same role as inequalities and intervals over numeric values (*ordinal* and *interordinal* scales [GW99]), or general terms in taxonomies.

A substring scale should be *complete* in the sense that every possible concept is derived from the scaled context, like when considering all substrings. A substring scale should also be *concise* enough so as not to overwhelm users during navigation, and be computed *efficiently*.

In Section 2, we present a naive conceptual scaling, and show that it does not satisfy conciseness and efficiency. In Section 3, we introduce a new solution, and show with the help of suffix trees that it has good properties w.r.t. completeness, conciseness and efficiency. Section 4 describes an algorithm for computing a complete substring scale from a set of string values. This algorithm is incremental, and so supports context updates, as required in information systems. It has been integrated into CAMELIS, an implementation of Logical Information Systems (LIS) [FR04], and Section 5 shows its application to a bibliographic context of ICCS paper titles, how many domain keywords are clearly identified, and how they naturally form a taxonomy. This paper ends with a discussion about other datatypes (Section 6), and a conclusion (Section 7).

2 Naive Approach

Suppose we have n objects, each object being described by a string over an alphabet Σ . This forms a *string context*.

Definition 1 (string context). A string context is a triple $D = (\mathcal{O}, \Sigma^*, d)$, where \mathcal{O} is a finite set of objects, Σ^* is the domain of strings over a finite alphabet Σ , and d is a mapping from objects to Σ -strings: for every object $o \in \mathcal{O}$, $d(o) \in \Sigma^*$ is the description of the object by a string.

A string context can be seen as a multivalued context with only one attribute, $d(o)$ being the value of this attribute for the object o . All results in this paper also apply to contexts with several attributes, but it is not necessary to consider them explicitly here as each attribute can be treated in isolation.

Example 1 The following table shows a basic string context that serves as an example in the following.

o	$d(o)$
1	abc
2	dab
3	ac
4	dab

The cover of a substring is the set of objects whose description contains it in a string context. This is equivalent to the definition of extent in logical concept analysis [FR04], where formulas would be strings and substrings.

Definition 2 (cover). Let $D = (\mathcal{O}, \Sigma^*, d)$ be a string context. The cover of a string $s \in \Sigma^*$ in D is defined by (where \supseteq denotes the containment relation between strings and substrings)

$$\text{cover}_D(s) = \{o \in \mathcal{O} \mid d(o) \supseteq s\}.$$

For example, in the above string context, the cover of the string "ab" is the set of object {1, 2, 4}.

We want to apply concept analysis on a string context, in order to group objects in a concept when they share common substrings in their description. There is *a priori* no way to prefer some substrings, so that we define the conceptual scale of all substrings of a string context, which accounts for the subsumption relations that exist between strings and substrings.

Definition 3 (scale of all substrings). Let $D = (\mathcal{O}, \Sigma^*, d)$ be a string context. The set of all substrings of the string context D is defined as

$$S(D) = \{s \in \Sigma^* \mid \text{cover}_D(s) \neq \emptyset\},$$

and the related conceptual scale is defined by $\mathbb{S}(D) = (d(\mathcal{O}), S(D), \supseteq)$.

Example 2 The scale of all substrings derived from the string context in Example 1 is given by the following table.

	abc	ab	bc	a	b	c	dab	da	d	ac
abc	x	x	x	x	x	x	x	x	x	x
dab		x		x	x		x	x	x	x
ac				x		x				x

A string context and its derived substring scale can be combined in order to form a scaled formal context, from which the concept lattice can ultimately be built.

Definition 4 (scaled context). Let $D = (\mathcal{O}, \Sigma^*, d)$ be a string context. The scaled context that is derived from D is defined by $K(D) = (\mathcal{O}, S(D), I)$, where

$$(o, s) \in I \iff d(o) \supseteq s.$$

Example 3 The scaled context derived from the above string context and substring scale is given by the following table.

	abc	ab	bc	a	b	c	dab	da	d	ac
1	x	x	x	x	x	x	x	x	x	x
2		x		x	x		x	x	x	x
3				x		x				x
4		x		x	x		x	x	x	x

There are now 3 properties we want to consider in the evaluation of this naive approach:

- completeness: *Is every set of objects sharing a common set of substrings a formal concept of $K(D)$?*
- conciseness: *Is the set of all substrings $S(D)$ concise enough so as to be useful for navigation ?*

– efficiency: *Can the formal context $K(D)$ be computed efficiently ?*

Completeness is here trivially ensured because all substrings occurring in the string context are considered. All other strings label only the bottom concept, and can be ignored without practical consequence. Conciseness can be evaluated as the size of the scale $S(D)$. Given a string context made of n strings of length up to k , the number of substrings is in $O(k^2n)$. Efficiency can be evaluated as the cost of computing the scaled context. Each substring must be produced, and checked against already produced substrings. With the help of a lexical tree, this check can be made in $O(k)$, so that the scaled context can be computed in $O(k^3n)$.

To get an idea of these complexities, suppose we have 1000 strings of length up to 100 characters (e.g., a set of paper titles), the number of substrings, and hence the number of attributes in the scale context, would be up to 10^7 . This is an awful lot, and in most cases many substrings will be redundant: e.g., the substring “ormal Context” generally has the same cover as “Formal Context”.

3 Maximal Substrings and Suffix Trees

Our objective is to reduce the number of attributes in the scaled context, while retaining all the information, i.e., while deriving the same extents and equivalent intents. This is possible because generally many substrings label the same concept (they have the same extent), i.e., they occur in exactly the same strings. The most concise solution consists in retaining one substring label for each meet-irreducible concept, but this entails a loss of information in intents and an arbitrary choice among substring labels.

3.1 Maximal Substrings

The idea is to retain only the more informative substrings, that is the substrings that cannot be extended (by adding characters at the left or at the right) without reducing their cover. This is equivalent to retaining *maximal* substring labels on each concept. We define a new scale of maximal substrings.

Definition 5 (scale of maximal substrings). *Let $D = (\mathcal{O}, \Sigma^*, d)$ be a string context. The set of maximal substrings of the string context D is defined as*

$$S_{max}(D) = \{s \in S(D) \mid \forall t \in S(D) : t \supset s \Rightarrow cover_D(t) \not\supseteq cover_D(s)\},$$

and the related conceptual scale is defined by $\mathbb{S}_{max}(D) = (d(\mathcal{O}), S_{max}(D), \supseteq)$, where \supseteq denotes the containment relation between strings and substrings.

Example 4 *The scale of maximal substrings derived from the string context in Example 1 is given by the following table. Compared to the scale of all substrings given in Example 2, the substring “b” has disappeared because it has the same cover as “ab”. The same happens for the substrings “bc”, “da”, “d” and the empty string “”.*

	<i>abc</i>	<i>ab</i>	<i>a</i>	<i>c</i>	<i>dab</i>	<i>ac</i>
<i>abc</i>	x	x	x	x		
<i>dab</i>		x	x		x	
<i>ac</i>			x	x		x

This scale can be combined with the string context from which it is derived in order to form a scaled context.

Definition 6 (scaled context with maximal substrings).

Let $D = (\mathcal{O}, \Sigma^*, d)$ be a string context. The scaled context that is derived from D is defined by $K_{max}(D) = (\mathcal{O}, S_{max}(D), I)$, where $(o, s) \in I \iff d(o) \supseteq s$.

Example 5 The scaled context derived from the above string context and substring scale is given by the following table.

	<i>abc</i>	<i>ab</i>	<i>a</i>	<i>c</i>	<i>dab</i>	<i>ac</i>
1	x	x	x	x		
2		x	x		x	
3			x	x		x
4		x	x		x	

This scaled context is just a projection of the scale context in Section 2 over the set of substrings $S_{max}(D)$.

3.2 Completeness

An important question is: *Did we loose something by retaining maximal substrings only?* More precisely, we have to show that the concept lattice has the same extents, and equivalent intents. We first prove that every substring has a maximal substring extension with the same cover.

Lemma 1. *Let D be a string context. For all substring $s \in S(D)$, there exists a maximal substring $m \in S_{max}(D)$ such that m is an extension of s , $m \supseteq s$, and has the same cover, $cover_D(m) = cover_D(s)$.*

Proof: We prove by recurrence that the lemma is true for every length of s (denoted by $|s|$), starting with the longest (string length is bounded by k), and decreasing it.

1. Base case: $|s| = k$.
 $\exists t \in S(D) : t \supset s \implies s \in S_{max}(D)$.
2. General case: the lemma is assumed true for every substring longer than s .
 - either $\forall t \in S(D) : t \supset s \implies cover_D(t) \neq cover_D(s)$
 $\implies s \in S_{max}(D)$,
 - or $\exists t \in S(D) : t \supset s \wedge cover_D(t) = cover_D(s)$
 $\implies |t| > |s|$
 $\implies \exists m \in S_{max}(D) : m \supseteq t \wedge cover_D(m) = cover_D(t)$
 $\implies \exists m \in S_{max}(D) : m \supseteq s \wedge cover_D(m) = cover_D(s)$. ■

It follows immediately that every non-maximal substring attribute can be replaced by a maximal substring that contains it, hence no loss of information, and that has the same cover, hence discriminating the same extents.

Theorem 1. *The concept lattices of $K(D)$ and $K_{max}(D)$ are equivalent for conceptual navigation. They have the same extents, and intents are equivalent in the sense that missing substrings in the latter case are redundant, i.e., included in some maximal substring from the same intent.*

For instance, the substring “ormal Contex” is replaced by “Formal Context”, but “al Context” is maximal if it can be extended by either “Formal Context” or “Logical Context” in different strings.

3.3 Conciseness and Efficiency with Suffix Trees

In order to evaluate the improvement of using maximal substrings, we have to bound their number, and compare it with the set of all substrings. We also have to compare the computation complexity for building the scaled context, because a smaller context does not entails necessarily a more efficient computation.

It seems difficult to estimate precise complexities, given the definition of $S_{max}(D)$. However there exists a very interesting data structure for reasoning and computing with sets of strings, namely *suffix trees* [Ukk95,Gus97]. The suffix tree of a string is the compact lexical tree of all suffixes of this string, where *compact* means that branches may be labelled by several characters, and nodes have several children. Figure 1 displays the suffix tree of the string “googol\$”, where \$ is a special final character that is necessary in the computation of the suffix tree. Each leaf represents a suffix, whose position in the string labels the leaf (starting with position 0). Each node represents a repeated substring, whose occurring positions are the leaf labels below this node. For instance, we can read in Figure 1 that the word “go” occurs at positions 0 and 3 in the string “googol\$”. In addition suffix links point from a node n (representing a substring s) to another node n' (representing the first suffix of s , i.e., s minus its first character). For instance, the suffix link in Figure 1 (dashed arrow) goes from the node “go” to the node “o”.

However we are here interested in finding maximal substrings over several strings, and so in building the suffix tree of a set of strings. This is an easy generalisation of suffix trees [Gus97]. It suffices to end each string with a different special character, and to concatenate them. A difference is that each leaf must be labelled by a string in addition to a position so as to determine to which string a suffix belongs to. Figure 2 displays the generalized suffix tree of the set of strings in Example 1. In leaf labels, the first number identifies the string, and the second number gives the suffix position in this string. The sets labelling nodes are the cover of the substrings they represent; and black nodes correspond to maximal substrings. The computation of these 2 informations is explained in Section 4.2.

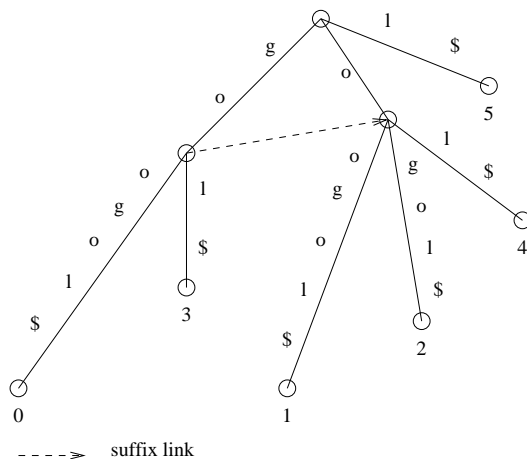


Figure1. The suffix tree of the string “googol\$”.

A known result states that every *maximal repeat* is represented by a node in the suffix tree (Lemma 7.12.1 in [Gus97]). These maximal repeats differ from our maximal substrings in 2 ways:

1. a maximal repeat may occur in a single string at different positions, and in this case cover a single object: these repeats are not maximal because they can be replaced by the full string that contains them (same cover),
2. full strings (from the string context) may not be repeated, but are obviously maximal substrings.

Proposition 1 (maximal substrings in suffix trees). *The set of maximal substrings forms a subset of the suffix tree nodes plus full strings.*

A fundamental result of suffix trees is that the number of nodes (hence the number of maximal repeats) and the computation time are both in $O(\sum_{o \in \mathcal{O}} |d(o)|)$, i.e., the cumulated size of all strings. This can be approximated as $O(kn)$, if k is the maximum length of strings, and n the number of strings.

Proposition 2 (number of maximal substrings). *The number of maximal substrings is in $O(kn)$, i.e., a k -fold improvement compared to the naive scaling.*

In the example of paper titles, this is about 2 orders of magnitude better. The same factor is obtained when computing the concept lattice, or even squared for algorithms that are quadratic in the number of attributes. About the complexity of computing the scaled context with maximal substrings, we need to take into account the selection of maximal substrings among suffix tree nodes, and the computation of covers. We describe an algorithm in next section, and show its complexity is in $O(kn \cdot \ln(kn))$, thus adding only an logarithmic factor to the computation of the suffix tree.

4 An Efficient and Incremental Algorithm based on Suffix Trees

As said in previous section, the maximal substrings of a string context is made of full strings and a subset of the nodes of the suffix tree of all these full strings. The first step is then to compute this suffix tree. We sketch in Section 4.1 the well known Ukkonen’s algorithm, which can build such a suffix tree in an incremental manner, and in linear time. In Section 4.2 we refine this algorithm in order to determine which nodes of the suffix tree represent maximal substrings. This requires the computation of substring covers, which is a useful information *per se* in information systems (e.g., for computing answers to queries). The complexity of this refined algorithm is given, as well as practical details on its implementation and integration with the existing logical information system, CAMELIS.

4.1 Ukkonen’s Algorithm for Computing Suffix Trees

The rough principle of Ukkonen’s algorithm is to have 3 gliding positions on a string s (having length n): the position j of the suffix $s[j, n]$ being added¹, the position $i \geq j$ of the next character to be read, and the position $j \leq p \leq i$ that matches the last node on the path $s[j, i]$ in the suffix tree. Initially i and j are set to 0, and p is the root. While the character $s[i]$ can be read down the tree, position i is incremented and p is updated accordingly. Otherwise a leaf labelled by j is added to the node nd at the end of the path $s[j, i - 1]$ (nd is created if this path ends in the middle of a branch), position j is incremented, a suffix link is followed from p in order to reach the path $s[j + 1, i - 1]$, and if nd has been created, a new suffix link is created from nd to the end of the new path. This is repeated until all suffixes have been added. Because position i is always greater or equal than position j , and at least one of these positions is incremented at each step, the suffix tree can be computed in linear time with respect to the length of s . This impressive result is achieved with the help of additional tricks, which are given in detail in the literature [Gus97,Ukk95].

4.2 Computing Covers and Maximality

We first give a few definitions and results to help navigating in suffix trees. In these definitions we talk equivalently of nodes and substrings. For instance, we can talk of the cover of a node, or we can talk about the node “ab” in Figure 2.

Lemma 2 (cover of a node). *The cover of a suffix tree node is the set of string identifiers that label the leaves below this node.*

Definition 7 (right extensions of a node). *The right extensions of a node nd are the children nodes of nd . The substring nd is a proper prefix of every right extension.*

¹ The notation $s[a, b]$ denotes the substring of s from position a to position b .

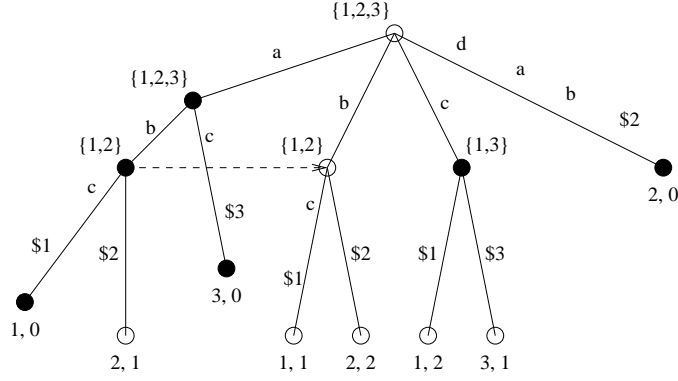


Figure2. The generalized suffix tree of the set of strings {“abc”, “dab”, “ac”}.

Definition 8 (left extensions of a node). *The left extensions of a node nd are the antecedent nodes of suffix links ending at nd . The substring nd is a proper suffix of every left extension.*

Our objective is to compute the set of maximal substrings, along with their cover. In Section 3.3 it was observed that a maximal substring is either a full string, or a repeated substring such that every other substring that contains it has a smaller cover. Therefore maximal leaves are those labelled with 0 as suffix position (full strings); and maximal nodes are those labelled by a cover strictly greater than any (left or right) extension.

Suppose we have the suffix tree for the $i - 1$ first strings of a string context, enriched with covers and maximality, and we want to update the enriched suffix tree with the string i . An important result can be summarized as “once maximal, always maximal”.

Lemma 3. *Once a node has been identified as maximal upon the addition of a string, it will remain maximal upon the addition of any other string.*

Proof: We prove the contraposition of this lemma. Suppose a node nd is not maximal after inserting the i -th string, and already existed before this insertion. This implies there is an extension nd' of nd that has the same cover. Suppose nd' was created during the insertion of string i . This implies that the substring represented by nd' was not a repeated substring, while nd was. This means the 2 nodes had different covers, which contradicts the fact they have the same cover after insertion of i . So nd and nd' did exist, had the same cover, and nd' was an extension of nd . Therefore nd was not maximal before the insertion of the string i . ■

Therefore a node becomes maximal when adding string i if it is not yet maximal, covers i , and has no extension covering i . From Lemma 2 the addition of string i in a cover comes from the creation of a leaf labelled by i below this node.

Modification 1. Ukkonen’s algorithm is modified so that each time a leaf is added on node nd , then the string i is added to the cover of nd as well as all its ancestors till the root. There are 2 cases about maximality:

1. if i already belongs to the cover of nd (due to the creation of a previous leaf), then nd has a right extension that covers i , and so it is not maximal;
2. otherwise i does not belong to any extension of nd . First, nd is marked as i -maximal, meaning that this node became maximal upon the insertion of string i . Then any ancestor of nd that was marked i -maximal is unmarked (this does not contradict Lemma 3 because unmarking occurs at the same stage as marking).

Modification 2. A second modification of Ukkonen’s algorithm is necessary. When a suffix link is created from some node nd' to the node nd , then a leaf has necessary been added to node nd' (Ukkonen’s algorithm), and so i belongs to the cover of nd' . As nd' is a left extension of nd , it follows that nd must be unmarked but *only if* it has been marked i -maximal.

The complexity of computing the enriched suffix tree is modified compared to Ukkonen’s algorithm. This is due to the traversal of the ancestors of a node in the first modification. Given the size of the suffix tree is in $O(kn)$, its height is in $O(\ln(kn))$. This traversal is applied for each creation of a leaf, i.e. $O(kn)$ times. Hence the time complexity for computing the enriched suffix tree is $O(kn.\ln(kn))$.

4.3 Practical Aspects

The above algorithm, based on Ukkonen’s algorithm, has been implemented and integrated in CAMELIS², a logical information system [FR04]. CAMELIS makes use of a toolbox of logic components, *logic functors* [FR02], amongst which the functor **String** handles representation and reasoning on strings and substrings. This functor has been extended with suffix tree algorithms so as to compute the maximal substrings in an incremental way. The cumulated complexity of a non-incremental algorithm would not be linear, but quadratic because of repeated computation of maximal substrings upon insertion of new strings. The logic functor also makes it possible to compute the extension of a substring so as to support navigation in CAMELIS, to remove strings from the string context, and to manually hide maximal substrings when judged irrelevant by users (customization). A concrete application example is given in the next section.

5 Example and Application-Specific Improvement

As an illustrative example of our approach, we consider the string context made of the titles of all papers published at ICCS from 1993 to 2005, as they were

² <http://www.irisa.fr/lande/ferre/camelis/>.

found on DBLP pages³. This string context contains $n = 374$ strings, whose length is bounded by $k = 140$. For string contexts of this size, the worst case number of substring attributes in the scaled context are the following.

all substrings (nk^2)	7,330,400
maximal substrings (nk)	52,360

In the case of maximal substrings, the worst case number of substrings is more sharply defined as the cumulated size of all strings, which is 21,412 in the ICCS string context.

We applied our algorithm for computing all maximal substrings of the ICCS string context. The computation time is a few seconds on a standard machine, and the number of maximal substrings is only 3,816. This is to be compared with 569,676 substrings found with the naive approach. This low figure, compared to the worst case, can partly be explained by the homogeneity of the string context, where titles share many common keywords. The size of the scaled context, i.e., the number of crosses, is 44,056; equivalently, objects have on average 117 attributes. This means that each title, whose average length is 57, contains on average 117 maximal substrings. Even if this is small in comparison to $k^2 = 19,600$, this still seems a lot.

Figure 3 shows a navigation tree in CAMELIS made of these maximal substrings, along with their object count. This tree contains several informative substrings, and the tree structure reflects their containment relations⁴: e.g., “Concept”, “Conceptual”, “Concept Analysis”, “Formal Concept”, “Negation in Concept”. But at the same time there are irrelevant substrings, e.g., “c”, “nce”, and redundant substrings, e.g., “al Concept” and “n Concept” w.r.t. “Concept”. The problem here is that the algorithm makes no assumption on the contents of strings, and makes no difference between letters and spaces. However, word boundaries are important for the readability and relevance of substrings for users.

We adapted the traversal of the navigation tree so as to allow applications to define a filtering of this tree. The application must determine for each substring which part is relevant. For instance, in the ICCS string context, this part goes from the beginning of the first capitalized word to the end of the last capitalized word, thus neglecting grammatical words at both ends: e.g., the relevant part of “al Concept” is “Concept”. Then a substring is filtered out from the navigation tree when its relevant part is equal to the relevant part of its parent node in the tree: e.g., “al Concept” is filtered because it has the same relevant part as “Concept”. This filtering entails no change at all in the suffix tree, and all displayed substrings *are* maximal substrings. The consequence is just that some substrings are skipped in the navigation tree, but the containment ordering is kept.

³ <http://www.informatik.uni-trier.de/~ley/db/conf/iccs/index.html>.

⁴ In fact, it is a directed acyclic graph as a substring may be subsumed by several substrings, but it is displayed as a tree in the graphical interface. This implies a substring may occur at different places in the navigation tree.

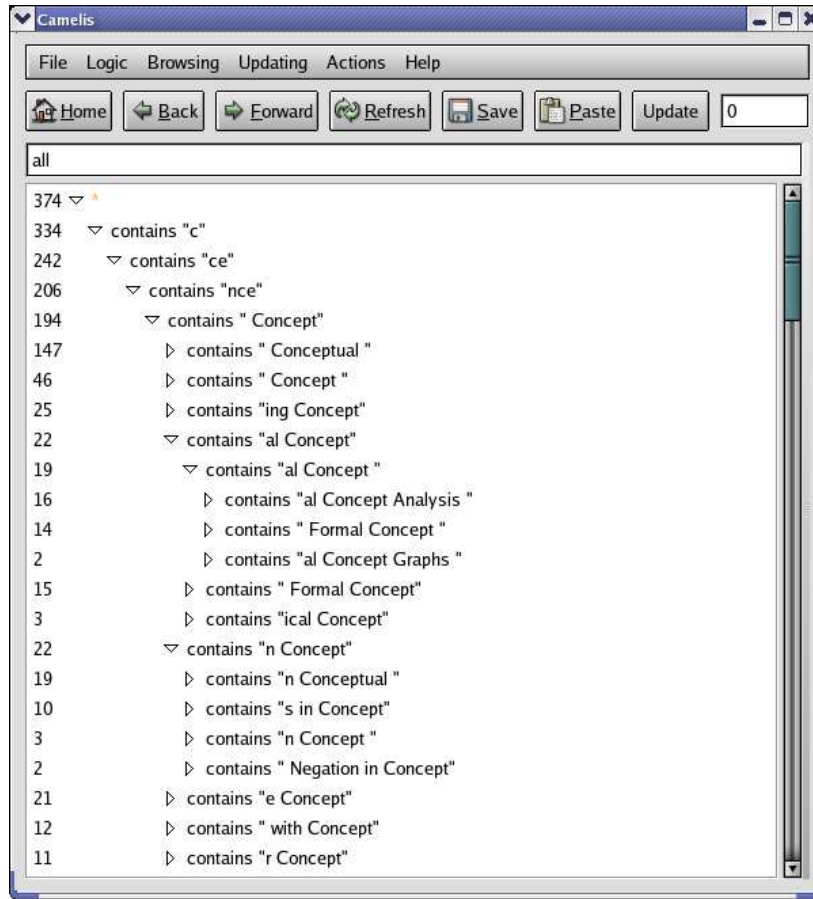


Figure3. Navigation tree of maximal substrings in ICCS string context.

A filtered navigation tree for the ICCS string context is displayed in Figure 4. This time we get a much more readable and informative navigation tree. Note for instance how “Concept” is refined by “Concept Analysis”, “Concept Graphs”, etc., and how “Concept Analysis” is refined by “Formal Concept Analysis” and “Temporal Concept Analysis” (skipping “al Concept Analysis”). Note also that full strings (prefixed by the keyword *is*) appear as maximal strings, so that full titles can be accessed. After filtering, the number of substrings is only 928 (vs 3,816). If only proper substrings are considered, i.e., if full strings are excluded, this number is a mere 554 substrings. This is the same order of magnitude as the number of objects. The following table summarizes the decreasing of the number of substrings in successive approaches.

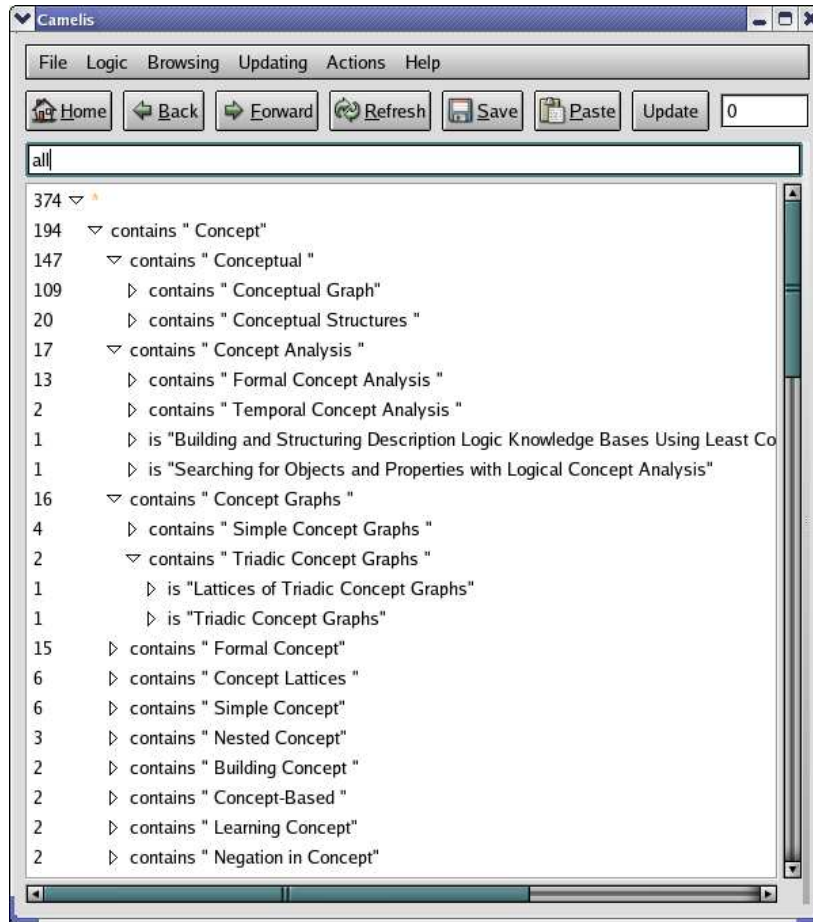


Figure4. Filtered navigation tree of maximal substrings in ICCS string context.

all substrings	569,676
maximal substrings	3,816
filtered maximal substrings	928
filtered maximal proper substrings	554

Navigation trees in above figures are dynamic. In conceptual navigation (a.k.a. browsing), they play the same role as attribute lists, and shrink to a subset of relevant substrings each time a substring is selected. Figure 5 displays the shrunk navigation tree after the substring “Concept Graphs” has been selected. Grey-colored substrings are those covering all selected strings: they make up the intent of the current concept. Of course, it is possible to consult the extent of the current concept as a list of titles (or full bibliographical references). Fi-

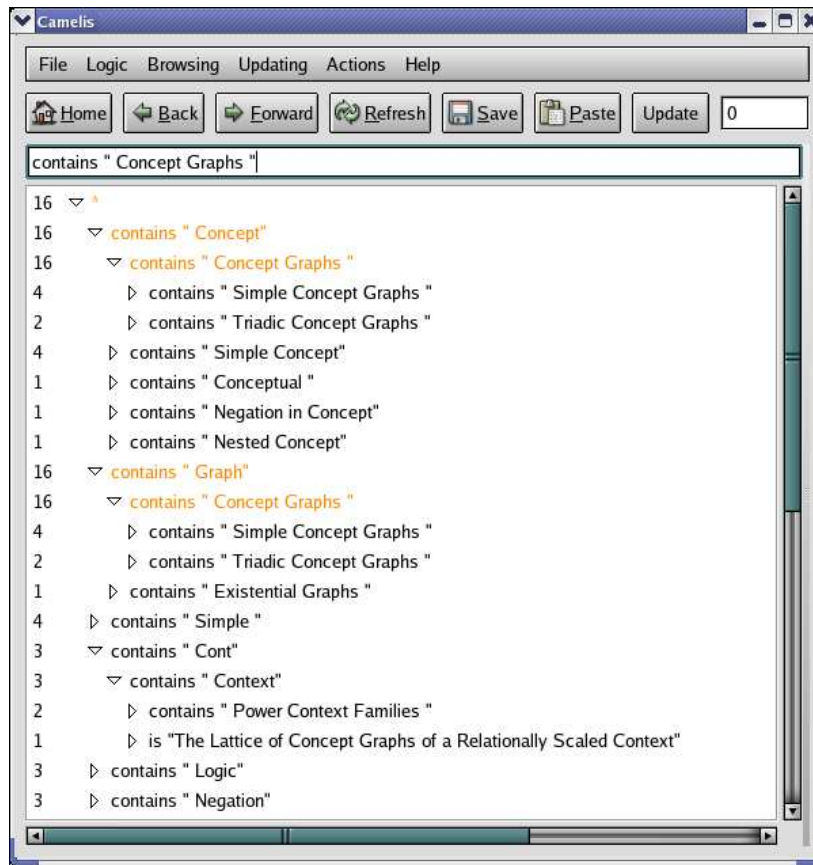


Figure5. Filtered navigation tree after selecting a substring in ICCS string context.

nally, thanks to the logical nature of CAMELIS, it is also possible to use arbitrary substrings in queries, even if they are not maximal.

6 Discussion

Our approach has no pretention w.r.t. Natural Language Processing (NLP). It proves successful on the analysis of a set of paper titles, but it is difficult to say how it would behave on more free text. Our purpose was to provide a solution for analysing strings that is simple and generic (no *a priori* knowledge needed, unlike in NLP), exhaustive (no arbitrary choices), and efficient (for actual use in information systems). However, if linguistic knowledge is available, it can still be used as a preprocessing stage before applying our algorithm: e.g., removal of plurals, replacing words by their root or a dictionnay entry.

The string datatype can be seen as a logic, where formulas are sets of strings and substrings, the deduction relation \sqsubseteq is based on the string containment \supseteq , and disjunction \sqcup computes the maximal substrings shared by 2 strings. This kind of logic can be used in the framework of Logical Concept Analysis (LCA) [FR00,FR04], where intents are precisely computed by application of disjunction: $int(O) = \bigsqcup_{o \in O} d(o)$. The set of maximal substrings can thus be computed by applying this disjunction on all subsets of objects in a logical context. A similar approach based on *pattern structures* [GK01] has been applied on graphs and subgraphs for analysing molecules [Kuz99]. These results could in principle be applied on strings, as strings can be represented with graphs. However, the complexity of this approach is totally different because the disjunction operation must be applied in the best case once for each concept. This results in an exponential complexity to be compared with the polynomial complexity of the algorithm presented in Section 4. The equivalent of our approach to graphs (if possible) would be to compute in a polynomial way the set of maximal subgraphs, i.e., all elements of *closed subgraph sets* [GK01], without computing the concepts.

A logic, defined as a language of formulas (representation) partially ordered by a subsumption relation (reasoning), is definitely valuable for describing objects in a natural way, querying a context in an expressive manner, and organizing navigation features, as demonstrated by our numerous experiments with LIS. However the use of logical disjunction for computing maximal features (e.g., substrings, subgraphs) and concepts is untractable in general. Given the importance of actually computing the concept lattice for many people, this may explain why LCA has not been more widely accepted, while there is an obvious and shared interest in exploiting various and richer datatypes. We hope the results presented in this paper for the string logic will improve its acceptability by the FCA community.

A long-term objective is to design a genuine logical *and* conceptual navigation for all sorts of datatypes. The results presented in this paper are a first step in this direction, and a significant one if we consider that strings cover a large part of many applications. An important way to reduce the problem is the shift from the direct production of the logical concept lattice to the production of maximal features that determine the same lattice. Indeed, producing the full lattice *a priori* is not necessary, as experienced in LIS applications, and it is not manageable in many real applications given its size (computation cost and visualization). It seems sufficient to show neighbour concepts of the current concept when browsing a context. Moreover the availability of maximal features makes it possible to compute the logical concept lattice with regular algorithms.

In LIS applications, logics are formed by the composition of *logic functors* [FR02] corresponding to various datatypes (e.g., strings, integers) and datatype constructors (e.g., sum, tuples, sets). It seems promising to extend logic functors so as to integrate the incremental computation of maximal features. This decomposition allows for highly specialized data structures and algorithms (e.g., suffix trees), and has been applied here for the string datatype.

7 Conclusion

We have defined *scales of maximal substrings* and their computation from multi-valued contexts with string-valued attributes. For each string-valued attribute such a scale is computed from a set of strings with the help of a suffix tree. Scales of maximal substrings are proved complete w.r.t. the formation of concepts. The use of suffix trees enables to bound their size by the cumulated size of strings (kn), and to efficiently compute them ($O(kn \cdot \ln(kn))$).

An algorithm is given as an adaptation of Ukkonen's algorithm for computing suffix trees. It has been integrated in logical information systems so as to support logical and conceptual navigation. This is illustrated on the navigation among ICCS papers through the automatic extraction of keywords from titles. We plan to extend these results to other datatypes, like strings with gaps in patterns, XML trees, and graphs, where the challenge is to compute maximal features without having to build the concept lattice, and possibly in a polynomial way like for substrings.

References

- [CS00] R. Cole and G. Stumme. CEM - a conceptual email manager. In G. Mineau and B. Ganter, editors, *Int. Conf. Conceptual Structures*, LNCS 1867, pages 438–452. Springer, 2000.
- [FR00] S. Ferré and O. Ridoux. A logical generalization of formal concept analysis. In G. Mineau and B. Ganter, editors, *Int. Conf. Conceptual Structures*, LNCS 1867, pages 371–384. Springer, 2000.
- [FR01] S. Ferré and O. Ridoux. Searching for objects and properties with logical concept analysis. In H. S. Delugach and G. Stumme, editors, *Int. Conf. Conceptual Structures*, LNCS 2120, pages 187–201. Springer, 2001.
- [FR02] S. Ferré and O. Ridoux. A framework for developing embeddable customized logics. In A. Pettorossi, editor, *Int. Work. Logic-based Program Synthesis and Transformation*, LNCS 2372, pages 191–215. Springer, 2002.
- [FR04] S. Ferré and O. Ridoux. An introduction to logical information systems. *Information Processing & Management*, 40(3):383–419, 2004.
- [GK01] B. Ganter and S. Kuznetsov. Pattern structures and their projections. In H. S. Delugach and G. Stumme, editors, *Int. Conf. Conceptual Structures*, LNCS 2120, pages 129–142. Springer, 2001.
- [GMA93] R. Godin, R. Missaoui, and A. April. Experimental comparison of navigation in a Galois lattice with conventional information retrieval methods. *International Journal of Man-Machine Studies*, 38(5):747–767, 1993.
- [Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
- [GW99] B. Ganter and R. Wille. *Formal Concept Analysis — Mathematical Foundations*. Springer, 1999.
- [Kuz99] S. Kuznetsov. Learning of simple conceptual graphs from positive and negative examples. In J. M. Żytkow and J. Rauch, editors, *Principles of Data Mining and Knowledge Discovery*, LNAI 1704, pages 384–391. Springer, 1999.
- [Ukk95] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.