Proceedings 2nd Int. Workshop on Knowledge Discovery in Inductive Databases KDID'03
co-located with ECML-PKDD 2003, J-F. Boulicaut and S. Dzeroski (Eds.),
Cavtat-Dubrovnik (Croatia), September 22, 2003. pp. 2-16. ISBN 953-6690-34-9.

A framework for frequent sequence mining under generalized regular expression constraints

Hunor Albert-Lorincz and Jean-François Boulicaut

INSA Lyon LIRIS CNRS FRE 2672 - Bâtiment Blaise Pascal
F-69621 Villeurbanne Cedex, France
{Hunor.Albert-Lorincz@insalien.org, Jean-Francois.Boulicaut@ insa-lyon.fr}

Abstract. This paper provides a framework for the extraction of frequent sequences satisfying a given regular expression (RE) constraint. We take advantage of the information contained in the hierarchical representation of an RE by abstract syntax trees (AST). Interestingly, pruning can be based on the anti-monotonicity of the minimal frequency constraint, but also on the RE constraint, even though this latter is generally not anti-monotonic. The AST representation enables to examine the decomposition the RE and to choose dynamically an adequate extraction method according to the local selectivity of the sub REs. Our algorithm, RE-Hackle, explores only the candidate space spanned over the regular expression, and prunes it at each level. Due to the dynamic choice of the exploration method, this algorithm surpasses its predecessors. We provide an experimental validation on both synthetic data and a real genomic sequence database. Furthermore, we show how this framework can be extended to regular expressions with variables providing context-sensitive specification of the desired sequences.

1. Introduction

Frequent sequential pattern mining in a database of sequences is an important data mining technique [1,11]. Its application domains span from the analysis of biological data to the discovery of WWW navigation paths or alarm analysis. In most of these applications, the lack of user control in specifying the interesting patterns beforehand leads to intractable extraction or costly post-processing phases. Considering other user-defined constraints, in conjunction with the minimal frequency constraint, has been proved useful [10,3,12,9]. Not only it limits the number of returned patterns but also it can reduce extraction cost to an acceptable extent. In this paper, we consider constraints expressed by regular expressions (RE). Thanks to REs, the user can specify the language of desired patterns in a rather flexible way. We consider that a *sequence* (or string) is an ordered list of items taken from a finite alphabet $\mathcal{A}$. ||S|| denotes the length of S, i.e., the number of items it contains. A database of sequences (D) is an unordered collection of sequences. $S' = s'_1 s'_2 ... s'_m$ is called a sub-sequence of $S = s_1 s_2 ... s_n$ $(m<n)$ if $\exists$ k s.t. $s'_1 s'_2 ... s'_m = s_{k+1} s_{k+2} ... s_{k+m}$. The frequency of a sequence S Freq(S,D) is the number of the sequences S' in D s.t. S is a sub-sequence of S'.
Problem statement: Given a database D, a regular expression E and a positive integer *minsup*, find all the sequences S which satisfy the following properties:

- $\mathrm{Freq}(S,D) \geq$ *minsup*
- $S \in \mathcal{L}(E)$, $\mathcal{L}(E)$ being the language specified by the regular expression E.

Constraint-based mining is difficult. Efficient algorithms are often ad-hoc, i.e., they are optimized for specific constraints and/or a specific kind of data. Several algorithms exploit efficiently the minimal frequency constraint thanks to its anti-monotonicity [1,6,4,8,11]. Pushing constraints that involve other user-defined constraints has been studied as well, e.g., in [3,11,9,7]. A RE-constraint is generally neither anti-monotonic nor monotonic and thus can not be used directly for pruning the search space. Furthermore, if a non anti-monotonic (nAM) constraint is "pushed" inside an extraction algorithm, the requirement that the sequences must satisfy both the minimal frequency and the nAM constraint can lack of pruning [3]. To tackle RE-constraints, the authors of the SPIRIT algorithms have proposed in [3] several relaxations of RE constraints. These relaxations have led to four ad-hoc algorithms that use different pruning strategies and perform unequally depending on the selectivity[1] of the constraint. In other terms, the choice of a given SPIRIT algorithm must be based on the selectivity of the RE constraint which is a priori unknown. We would like a robust algorithm which depends weakly of the selectivity of the constraint and the characteristics of the data, i.e., an algorithm which would consider the individual selectivity of the sub-expressions and choose the best pruning strategy during the extraction according to the sequences in the database. This would be a major step towards efficient sequence mining for many application domains.

We proposed in [2] the RE-Hackle (Regular Expression-Highly Adaptive Local Extraction) framework. It is based on a hierarchical abstract syntax tree (AST) of the RE that can collect much more information on the local properties of the constraint than the previous methods based on Finite State Automata [3]. The collected information is used to examine each sub-expression of the initial RE and the RE-Hackle algorithm chooses the extraction strategy to favor pruning on the minimal frequency constraint or on the RE-constraint. For that purpose, candidate sequences are assembled during a bottom-up processing of the AST. Each node of this tree, the Hackle-tree, encodes a part of the RE and, when the algorithm reaches a node, it has information about the frequent sequences generated by its sub-tree. The number of these sequences, the structure of the tree in the neighborhood of the node and global information such as the cost per candidate evaluation and per database scan can be used to determine a candidate generation strategy and to optimize the execution time.

The contribution of this paper is threefold. We provide an introduction to the RE-Hackle algorithm defined in [2] and we introduce a new optimization technique based on Hackle-trees. Also, we provide experimental results on both synthetic data and real biological data that were missing from [2]. It shows that our algorithm adapts dynamically its pruning strategy and tends to take the shape of the best SPIRIT algorithm without any prior knowledge of constraint selectivity. Finally, we show that the framework can be extended to regular expressions with variables. Adding variables clearly increases the expressive power of the exploration language and reaches beyond the scope of context free grammars.

[1] Selectivity is an intuitive characterization of REs which is roughly speaking inversely proportional to the number of sequences in the database that match the initial constraint.

The paper is organized as follows. Section 2 provides the needed definitions. The RE-Hackle basic algorithm is described in Section 3. An original optimization technique is described in Section 4. Section 5 overviews the experimental results and it compares RE-Hackle to our fair implementation of two SPIRIT algorithms. Section 6 introduces an extension of the framework to regular expressions with variables. Section 7 concludes.

2. Definitions

A *RE constraint* in a kind of a regular expression built over an alphabet of sequences using the following operators: union (denoted +), concatenation (denoted ¤_k and sometimes "." when k is 0) and Kleene closure (denoted *). The empty set and the empty sequence are noted Ø and ε.

The *k-telescoped concatenation* of two sequences $S=s_1\, s_2 \ldots s_{||s||}$ and $P= p_1\, p_2 \ldots p_{||P||}$ is a new sequence. This operator requires that the sequences overlap in k positions. When k is zero, we get the usual concatenation. It is used for candidate generation.

$\text{Op¤}_k(S, P)= S\ \text{¤}_k\ P = \{ s_1\, s_2 \ldots s_{||s||-k}\, p_1\, p_2 \ldots p_{||p||}\}$ if for all $0<j\leq k$ we have $p_j=s_{||s||-k+j}$ and $||S|| > k$ and $||P|| > k$. If $\exists\ 0<j\leq k$ s.t. $p_j<>s_{||s||-k+j}$ $\text{Op¤}_k(S, P)=\varepsilon$**.**

Concatenating two sets of sequences $\{S_1, S_2, \ldots, S_n\}$ and $\{P_1, P_2, \ldots, P_m\}$ gives a new set of sequences that contains all the sequences resulting from the Cartesian product of the sequences from the two sets.

$$\{S_1, \ldots, S_n\}\ \text{¤}_k\ \{P_1,\ldots, P_m\} = \{ S_i\ \text{¤}_k\ P_j \mid 0<i \leq n \text{ et } 0<j \leq m\}$$

The k-telescoped concatenation of *n* sequences $S_1, S_2, \ldots, S_n$ is:

$$\text{Op¤}_k(S_1, S_2,\ldots, S_n) = \text{Op¤}_k(S_1, \text{Op¤}_k(S_2,\ldots, S_n))$$

E.g., $\text{Op¤}_3(\text{ACDE},\text{CDEF},\text{DEFD}) = \text{Op¤}_3(\text{ACDE},\text{Op¤}_3(\text{CDEF}, \text{DEFD})) = \text{Op¤}_3(\text{ABCD},\text{CDEFD}) = \text{ACDEFD}$.

The union of *n* sequences $S_1, S_2, \ldots S_n$ is the set of sequences:

$$\text{Op+}(S_1, S_2,\ldots, S_n) = \{S_1, S_2, \ldots, S_n)\}$$

The union of two sets of sequences is the union of these sets:

$$\{S_1,\ldots, S_n\} + \{P_1,\ldots, P_m\} = \{S_1,\ldots, S_n, P_1,\ldots, P_m\}$$

The *Kleene closure* applies to a set of sequences and denotes all the sequences one can build from them using concatenations. It includes the empty set **Ø**.

$$\text{Op*}\{S_1,\ldots, S_n\} = \{ \textbf{Ø}, \{S_1, \ldots, S_n\}\ \text{¤}_k\ \{S_1, \ldots, S_n\}, \\ \text{Op*}(\{S_1,\ldots, S_n\}\ \text{¤}_k\ \{S_1,\ldots, S_n\}, S_1,\ldots, S_n)\}$$

Moreover, the function *frequent* applied to a set of sequences $\{S_1,\ldots, S_n\}$ scans the database and returns a set that contains the frequent sequences.

The operators have a variable arity. The priority increases from + to ¤_k and from ¤_k to *. The concatenation can be distributed over the union. The union and the concatenation are associative. When all the possible concatenations have taken place, the resulting sequence is called an *atomic sequence*. Consider the RE-constraint B.CD.E.A(H+F) which can be transformed into BCDEA(H+F) by 3 concatenations. According to our definition, BCDEA is a newly formed atomic sequence. H and F were already atomic sequences as they cannot be concatenated to their neighbors, but B, CD, E and A are not as they can be packed together to form a longer sequence. The building bricks of our RE-constraints are the atomic sequences, i.e., the smallest elements considered during the extraction phase.

Canonical form We say that a regular expression is in a *canonical form* if it contains only atomic sequences. In the following, we assume that all the REs are in the canonical form.

Examples of RE-constraints and their associated derivation phrases:
A+BE+CF+D = Op+(A, BE, CF, D)
A(B)*(CF+D) = Op¤$_0$(A, Op*(B), Op+(CF,D))

Sub-constraints are taken from the initial RE-constraint: they correspond to terms of the derivation phrase. Extraction must not break the priorities of the operators. E.g., B+C can not be extracted from A¤$_0$B+C as the priority of the concatenation prevails over the union. Besides, a sub-constraint must contain as many terms as the arity of the operator in the initial constraint. A *maximal sub-constraint* is a sub-constraint, which is not contained in any other sub-constraint except the initial constraint. E.g., A¤$_0$B+C has two maximal sub-constraints: A¤$_0$B and C. The maximal sub-constraints naturally define partitions over the initial RE. The *active operator* connects the maximal sub-constraints of a given constraint. E.g., the active operator for A¤$_0$B+C is the union.

Hackle-tree. A *Hackle-tree* is an AST which encodes the structure of the canonical form of a RE-constraint. Every inner node of this tree corresponds to an operator, and the leaves contain atomic sequences of (possibly) unequal lengths. The tree reflects the way in which these atomic sequences are assembled by the operators to form the initial RE-constraint. Figure 1 provides such a tree for the RE-constraint C((C(A+BC)D)+(A+B+C)*)C. Nodes are marked with roman numbers to support the discussion. Attributes associated to each node have been described in [2] and are given in the following table:

Attribute	Semantics
type	Type of the node: ⊥ leaf, ¤ concatenation, + union, * Kleene closure.
siblings	List of the siblings. NULL for the leaves.
parent	Parent of the node. NULL for the root.
ξ_{th} ,ξ_{exp}	Theoretical and experimental cardinality of the node.
items	Frequent legal sequences found by the node.
state	*Unknown* – The exploration of the node has not yet begun. *Satisfied* – Exploration has found frequent legal sequences. *Violated* – The node did not generated a frequent sequence.
explored	Coupled to the attribute State. True if the exploration is completed.
K	Parameter for the k-telescoped concatenation*:* ¤$_k$
age	Only for Kleene closures: counts the times the node has been visited.
seq	Only for the leaves: encoded atomic sequence or symbol.

The construction of the tree is not trivial. The RE has to be transformed in its canonical form and arities have to be computed from the number of the maximal sub-constraints. Then a node is created for the active operator as well as a sibling for each maximal sub-constraint, which is expanded recursively. This method minimizes the height of the tree and can be encoded efficiently.

The intuitive notion of selectivity can be formalized using cardinalities [2]. The *theoretical cardinality* of a constraint is the number of sequences it can generate after the expansion of all the operators if every possible sequence is in the database. The *experimental cardinality* of a constraint is the number of sequences extracted from the database. While the theoretical cardinality refers only to the constraint, the experimental cardinality takes into account the database instance, i.e., the results of the counting phases.

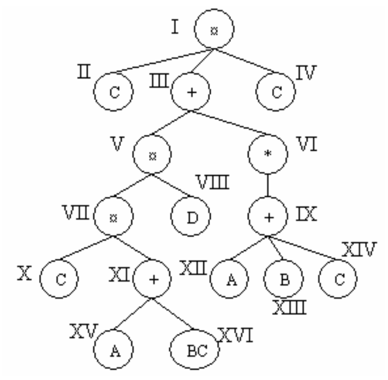

Figure 1 *Hackle-tree encoding constraint* C((C(A+BC)D)+(A+B+C)*)C.

The *extraction phrase* Ψ is the list of the nodes one must examine at a given step. The whole extraction process is controlled by this phrase: its modification defines a new generation and thus a new database scan. When the extraction starts, Ψ contains all the leaves of the tree collected from left to right. It is updated after each database scan by replacing the explored nodes with their parents.

The *extractor functions* (denoted C) are applied to the nodes of the Hackle-tree, and return the candidates that have to be counted. They are defined as follows:

- Leaves: C(N) = N.seq
- Concatenation: all siblings of N must be explored,

  C(N) = $¤_k$ M.items, for all M∈N.siblings , k comes from the node N
- Union: all siblings of N must be explored,

  C(N) = ∪ M.items, for all M∈N.siblings
- Kleene closure: the sub-tree of N must be explored,

$$\mathrm{C(N)} = \bigcup_{age>0} \mathrm{C(N, age)}$$

with C(N,age+1)=$¤_{\text{age-1}}$frequent{C(N,age)}and C(N,1)=N.siblings.items

3. The RE-Hackle Algorithm

RE-Hackle extracts all the frequent sequences which match a given regular expression, i.e., which are valid w.r.t. the root of its Hackle-tree. Details about the basic algorithm are given in [2]. Here, we just provide informal comments that enable to introduce our new optimization and the extension to RE-constraints with variables. The algorithmic schema is as follows (T is a Hackle-tree, E a RE-constraint, Ψ the extraction phrase, and C the set of candidates):

```
T ← BuildTree (E)
loop    Ψ ← BuildExtractionPhrase (T)
        C ← GenerateCandidates (Ψ)
        CountCandidates (C)
        T ← TransformTree (T)
while Ψ<>Ø and C<>Ø
return T.root.items
```

Extraction relies on the extraction phrase. It starts at the leaves of the Hackle-tree and lasts several generations. At every generation, the extraction functions are applied to the nodes in the extraction phrase; the algorithm counts the candidates and uses the frequent ones to feed the next generation (an upper level) which will be assembled by the new nodes of the updated extraction phrase. It is a levelwise algorithm in the structured space of the language associated to the RE-constraint. The number of the levels is limited by the height of the Hackle-tree plus the number of times Kleene nodes are evaluated. As candidates are built up from atomic sequences, it takes usually less database scans than GSP-like algorithms [1]. The Hackle-tree is transformed after each generation, e.g., for pruning branches which can no longer generate new candidates.

Let us comment an execution of the algorithm given the following database, a minimal frequency of 2, and the RE-constraint from Figure 1.

ID	Sequences
1	CCADCABC
2	ECBDACC
3	ACCBACFBAC
4	CCBAC

The extraction needs for 7 generations and 6 database scans.

1st Generation

Ψ_1 = II,X,XV,XVI,VIII,XII,XIII,XIV,IV
Candidates: A, B, C, D, BC
Frequent sequences: A, B, C, D

BC is not frequent and the algorithm prunes Node XVI.

2nd Generation

Ψ_2 = VI,VII
Candidates: AA, AB, AC, BA, BB, BC, CA, CB, CC
Frequent sequences: AB, AC, BA, CB, CC

Ψ_2'= XI,IX would have been the application of the defined principle: the substitution of Nodes XV and XVI by XI and the substitution of Nodes XII, XIII and XIV by IX. In fact, as unions never need to access the database, the algorithm replaces them with their nearest concatenation or Kleene closure parent during an intermediate generation. It is not necessary to compute explicitly Ψ_2'. While processing Ψ_2, no frequent sequence has been found at node VII, so it is erased from the tree. Its parent, the node V, can be pruned too thanks to the anti-monotonicity of the minimal frequency constraint.

3rd Generation

Ψ_3 = VI *(age=1)*
Candidates: ABA, ACB, ACC, BAB, BAC, CBA, CCB, CCC
Frequent sequence: ACC, BAC, CBA, CCB

A Kleene Closure node remains in the extraction phrase while it continues to generate frequent sequences.

4th Generation

Ψ_4 = VI *(age=2)*
Candidates: ACCB, BACC, CBAC, CCBA
Frequent sequences: CBAC, CCBA

ACC ¤$_2$ CCB = ACB. Parameter k of the concatenation is given by the age of the Kleene node.

5th Generation

Ψ_5 = VI *(age=3)*
Candidates: CCBAC Frequent sequences: CCBAC

6th Generation

Ψ_6 = VI *(age=4)*
Candidates: - Frequent sequences: -

No candidate to count since CCBAC ¤$_4$ CCBAC = ε.

7th Generation

Ψ_7 = I
Candidates: CC, CAC, CBC, CCC, CABC, CACC, CBAC, CCAC, CCBC, CCCC, CACCC, CBACC, CCBAC, CCCBAC, CCBACC, CCCBACC
Frequent sequences: CC, CBAC, CCBAC

The Kleene closure returns every frequent combinations of A and B, plus the empty sequence. The root assembles them to C and generates the result, i.e., the frequent items associated to the root. Notice that the candidates of the k^{th} generation are not necessarily of length k.

Let us now discuss the processing of the Kleene closure nodes thanks to *second-level alphabets*. For each atomic sequence of a constraint, we are defining a new symbol which replaces it in an equivalent second-level RE-constraint. For example, the RE-constraint A(AB|FE)DCA becomes A(α|β)χ given the new symbols α=AB, β=FE and γ=DCA. Every initial symbol such as A is added automatically to the second-level alphabet. The RE-Hackle algorithm works with second-level alphabets.

The nodes corresponding to the Kleene closures collect the frequent sequences extracted by their descendents. These sequences of unequal length must be combined at later ages to compute the closures.

By definition, the *age* of a Kleene closure is the number of times it has been visited (it begins with 1). Assume that A, ADC and BAD have been found frequent by the siblings of a Kleene node, it means that the closure of {A,ADC,BAD} must be computed. The candidates should be {AA, AADC, ABAD, ADCA, ADCADC, ADCBAD, BADA, BADADC, BADBAD}. Assume now that all of them are frequent. At the third age, the node must concatenate only the sequences, which share a common sub-sequence of the first generation, i.e., A, ADC or BAD. For example, even though ABAD and ADCA share a common sequence of length 2, they should not be concatenated because ABADC does not belong to {A,ADC,BAD}*. With a representation that keeps no information about the composition of the second (nth) age sequences, the overlapped concatenation of these sequences is practically impossible. It has motivated the introduction of second-level alphabets. Dealing with {A,β,γ} given that β=ADC and χ=BAD is an elegant solution for candidate generation. The second level candidate Aβ will be converted to its first level representation, i.e., AADC for counting purposes and handled in its second level representation when computing overlapped concatenations. Notice that in our hierarchical representation, the overlapping parts of the sequences are easily identified. For instance, the sequence Aβχ = (A)(ADC)(BAD) contains only three symbols in its second level representation, which is a shortcut for a flattened sequence of length 7. Candidates are generated in the same way as with GSP [1] where the age of the Kleene node encodes the number of the overlapping second-generation sequences (parameter k for the k-telescoped concatenation $¤_k$). The use of a second level alphabet can boost the extraction of the Kleene closure, as the candidates are assembled from sequences rather than from simple symbols. Consequently, their length grows faster then with any previous method and the number of database scans can drop drastically.

4. Optimization

Let us introduce an original optimization for RE-Hackle, the so-called *ascending flux of candidates*. It can be easily shown that the exploration of each Kleene node delays by one generation the evaluation of its parent, as parents can not be put in the extraction phrases if the extraction of their siblings is not completed. Although generally it does not penalize the extraction (the algorithm can continue to work with some other branches of the tree), it would be better to have a guarantee that the embedded structures will never lack of efficiency. Therefore, we decided to forward the frequent sequences found by a Kleene node immediately to its parent, i.e., without waiting for the completion of the exploration. Indeed, the extraction against constraint A*B can be inefficient if the database contains long sequences of symbols A but does not contain AB. We can forward the extracted sequences immediately after having found them. Thus, the optimized algorithm can work on several levels of the Hackle-tree. The extraction phrase is now allowed to contain nodes which are in direct son-parent relations.

The reconstruction of the extraction phase becomes more complex, so does the propagation of the violations and the candidate generation. Technically, it is achieved by 3 possible values for attribute *explored*:

- Waiting: the exploration of the node has not yet begun
- Inprogress: the exploration has begun, but the node will generate some more candidates during the next generation
- Finished: the exploration is finished and the node has been taken out from the extraction phrase.

If the parent of a closure is a concatenation which generates no frequent sequence at a given iteration, then its sub-trees can be pruned immediately and the exploration of the closure can be stopped. The candidates from the nth age can be counted together with the frequent sequences of the (n-1)th age concatenated to the other siblings of the concatenation. In this way, when the nth age becomes empty, the concatenations of the (n-1)th age with the other siblings are available and no iteration is wasted. In Table 1, we compare the basic and the optimized algorithm when AAAACBB is mined given the RE A*B. The optimized version recognizes that AB does not occur in the database and it does not generate useless sequences of A. Here, the optimization saves four database scans.

Scan	RE-Hackle (basic)	RE-Hackle (optimized)
1	A, B	A, B
2	AA,B	AA, B, ~~AB~~
3	AAA	STOP
4	AAAA	
5	~~AAAAA~~	
6	~~AB~~, ~~AAB~~, ~~AAAB~~,~~AAAAB~~	
7	STOP	

Table 1 *Ascending flux of candidates. Infrequent candidates are stroked.*

In some cases, the basic algorithm can give rise to a large number of candidates without any pruning. For instance, assume a concatenation with a dozen of siblings each of them producing 3 to 4 frequent sequences. The extractor functions would return between 531.441 to 16.777.216 candidates and this is clearly unacceptable.

Fortunately, the *adaptation of the extraction method* can avoid this combinatorial explosion. It is always possible to group the nodes in larger overlapping buckets, and to benefit of more frequency-based pruning. Figure 2 illustrates this principle. Assume that the siblings A, B, C and D of the node X return many frequent sequences (*Here, A,B,C, and D denote the nodes of the tree and are not elements of the alphabet*), and that we do not want the concatenation to produce a large number of candidates. So, to protect X, we replace it by a new node X^2 which introduces a new level (nodes Y, Z and W). The initial nodes are grouped two by two and associated to Y, Z and W to enable pruning. As the suffixes of the sequences in Y are the same as the prefixes of Z, the candidates of X^2 will be constituted by a *1-telescoped concatenation*.

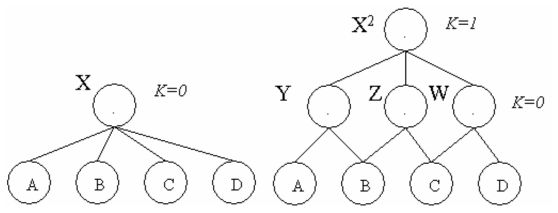

Figure 2 *– Adaptation of the extraction method.*

After the evaluation of the additional level the number of the candidates should globally decrease. So, X^2 contains only three sons and it is supposed to generate fewer candidates than X. If this number is still too large, the algorithm can decide to introduce one more additional level between Y, Z, W and X^2. The number of the inserted levels is determined by the algorithm during the extraction by the use of the theoretical cardinality. If the theoretical cardinality of a concatenation node exceeds a level, this optimization technique introduces a new level between the node and its siblings. Each new level requires another whole database scan, but the number of the candidates is expected to decrease. This mechanism enables a tradeoff between the number of the candidates and the number of database scans. One should notice, that the length of the candidates increases with each level, so the RE-Hackle will never take more passes than GSP or SPIRIT(L and V). The control of the tradeoff is however complex: it depends on the size of the database, the cost per candidate for counting and the cardinalities of the siblings of the node we are considering.

5. Experimental Results

We have used a semi-optimized implementation of the RE-hackle algorithm for our experiments (no ascending flux of candidates). Furthermore, we have put the data in main memory for both our SPIRIT and RE-Hackle implementations. We used our fair implementations of SPIRIT(L) and SPIRIT(V) [3]. First, we have generated synthetic datasets following a zipfian distribution. First, our synthetic dataset contains 100k transactions of length 20 over an alphabet of 100 symbols. We have then decreased the number of the symbols in the alphabet and created conditions for the emergence of long patterns. The execution time of our algorithm (see Figure 2) does not increase exponentially and take the properties of the better algorithm for this case, i.e., SPIRIT(V).

We have been considering different RE-constraints with different granularities (ratio between the number of symbols composing the RE and its number of atomic sequences). E.g., the granularity of AB(CDE|RY)CF is 9/4=2.25, for AC|T(G|BB)E we have 7/5=1.4, and for (A|B|C)*D(T|Z) we get 6/6=1. The execution time of RE-Hackle decreases as the granularity increases (see Figure 4). Notice, that the SPIRIT algorithms are practically insensible to this factor.

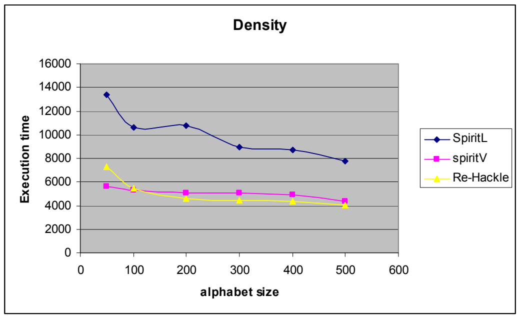

Figure 3 *Influence of the data density*

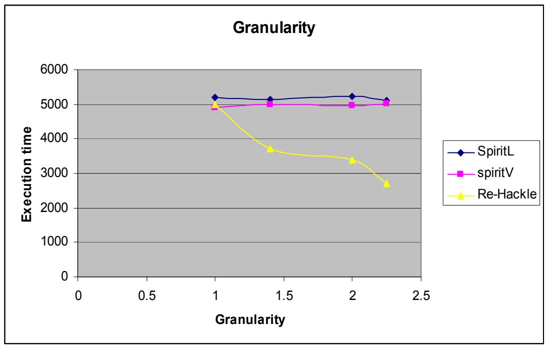

Figure 4 *Influence of RE granularity*

The power of RE-Hackle pruning strategy is somewhat between SPIRIT(L) and SPIRIT(R)[2]. As SPIRIT(L) relies on frequency-based pruning and SPIRIT(R) on RE-based pruning, RE-Hackle will not always beat both of them, but it is quite often the best, as it uses both pruning criteria. We provide in Figure 5 a comparison between RE-Hackle, SPIRIT(L) and SPIRIT(V) on real biological data (5000 sequences of length 3000 to 6000 items over an alphabet of four symbols given by Dr. O. Gandrillon from University Lyon 1). Various RE-constraints (with an increasing experimental cardinality) have been used. As the pruning strategy is adjusted dynamically according to the cardinality of the constraint and the content of the database, RE-Hackle approaches the performances of the best SPIRIT implementation without prior knowledge of the selectivity of the constraint. This behavior was mainly obtained by the use of the adaptive extraction method.

[2] [3] introduces 4 SPIRIT algorithms denoted N, L, V and R. Following this order, the algorithms do more and more RE-based pruning (almost no RE-pruning with SPIRIT(N), no frequency-based pruning for SPIRIT(R)).

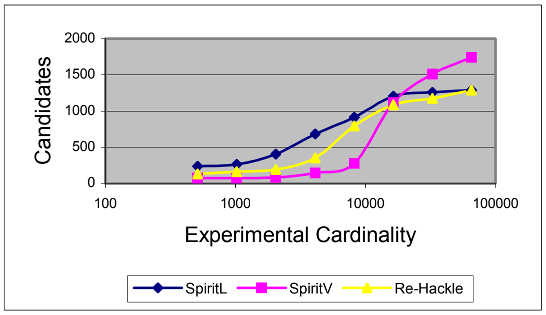

***Figure 5** Comparison to SPIRIT variants*

This is an important qualitative result. It is not surprising that both SPIRIT algorithms have their operating areas and RE-Hackle is situated between the two curves. As the experimental cardinality rises, RE-Hackle begins to use more and more frequency-based pruning and avoids the combinatorial explosion which makes SPIRIT(V) inadequate to deal with high cardinality constraints. At a given complexity, RE-Hackle decides to use only frequency-based pruning and has the same behavior than SPIRIT(L). Indeed, we have an adaptive pruning method.

6. Extension to RE-constraints with variables

Our partners, the biologists, have shown interest in being able to specify variables in the RE constraints. For instance, it is interesting to look for frequent sequences that match a generalized RE like X(A|B)C*XCB (where X denotes a variable which take values in $\mathcal{A}$*). Hackle-trees provide an elegant way for tackling variables which specify that recurring sub-patterns are desired.

A *variable* X[3] is a shortcut for any sequence built over $\mathcal{A}$. Starting from now, we use upper case letters for variables and sequences while lower case letters are used for symbols. An *extended RE-constraint* is a RE-constraint that can contain variables. For instance, constraint (A|C)XBX requires that every sequence begins with an A or a C and contains two occurrences of the same sub-sequence X separated by the symbol B. Sequences ACDBCD or CADBAD satisfy it. Sub-sequences CD and AD are two *instances* for the variable X.

An *augmented second-level alphabet* is a second-level alphabet that contains a new symbol for every distinct variable. An extended RE-constraint is built over an augmented second-level alphabet, i.e., an alphabet that contains some variables. Mining under extended RE-constraints is now straightforward using an X-tree, i.e., a Hackle-tree on the augmented second-level alphabet associated to the extended RE-constraint.

For practical reasons, we impose now that Kleene closures can not return the empty sequence. If the result of a closure is empty, the node is immediately violated and pruned out of the X-tree. As a variable is a shortcut for all possible frequent sequences, every distinct variable can be modeled by the most general Kleene closure, i.e., a closure applied to the initial alphabet A. A variable which appears only once can be

[3] We reserve letters X and Y to denote variables.

replaced immediately by a Kleene closure. We thus assume that a variable appears at least twice in the RE-constraint. It is important to instantiate the variables with values that have been generated in the same generation. In other terms, when the expressions with variables are flowing up in the tree, the so-called X-Hackle algorithm has to know from which generation they come from. Therefore, they can be indexed by the number of the active generation of the Kleene node. This count starts when they begin to flow up in the X-tree. As variables are implemented via Kleene closures, the corresponding nodes have the following fields.

Attributes	Semantics
type	* Kleene closure
siblings	All the symbols of the initial alphabet.
parent	Parent of the node. NULL for the root.
ξ_{th}, ξ_{exp}	Same as in a Hackle-tree.
state, explored	Same as in a Hackle-tree.
Age	Counts the times the node has been visited.
items[1..age]	Frequent legal sequences found by the node in generation

Figure 6 provides the X-tree representation of the generalized RE-constraint Y(AX*)BXY. It is not needed to represent explicitly Nodes 2 and 6: their parent points directly to Node 11. Also, Nodes 5 and 9 are shortcuts for Node 10.

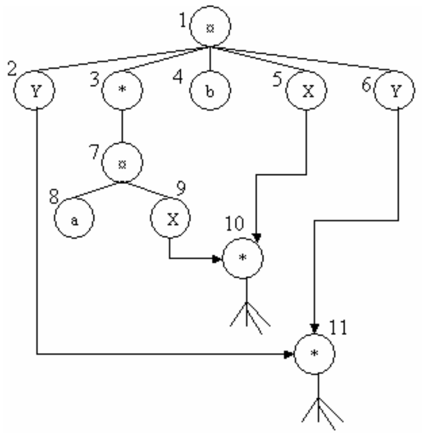

Figure 6 *X-Tree of the generalized RE-constraint* Y(AX*)BXY

Variables X and Y are shortcuts for two general Kleene closure nodes that will extract any possible frequent sequence. This setup combined with the ascending flux of candidates can extract the sequences described by generalized RE- expressions. Conceptually, the ascending flux of candidates can be viewed as the creation of a new X-tree at every generation even though, in practice, we can avoid the duplication of the tree. At the kth evaluation of the variables, the algorithm creates a clone of the X-tree in which the kth values of the variables are trapped. This tree is then explored by the RE-Hackle algorithm. At the next generation, X-Hackle creates a (k+1)th tree again and pass it to the RE-Hackle algorithm. Notice again, that in an implementation the duplication of the tree can be avoided by clever indexing. We now sketch the X-Hackle algorithm that uses the following definitions.

Let us denote *First*$_k$*(*$N_1, \ldots, N_m$*)* the function that, given a list of nodes, returns the set of the k-sequences (sequences of length k) which are prefixes of the concatenation of the nodes. Let us denote *Last*$_k$*(*$N_1, \ldots, N_m$*)* the function that, given a list of nodes, returns the set of the k-sequences which are suffixes of the concatenation of the nodes.

The X-Hackle algorithm can now be sketched. A longer version of this paper contains more details and an example of execution.

```
T ← BuildExpTree(E);
For all X∈Variables, $V_1$[X]=Ø
   For all A∈$\mathcal{A}$
     If Freq(A,D) > minsup
        Then $V_1$[X]←$V_1$[X] U {A}
gen ← 1
While (Promising()) do
   $T_{gen}$ ← T
   $V_{gen}$[.]← ComputeValues($V_{gen-1}$[.])
   For k = 1 to gen
      Iterate RE-Hackle($T_k$, $V_k$[.], k-gen+1)
   gen ← gen+1
Return $T_1$.root.items U … U $T_{gen}$.root.items
```

V_{gen}[X] contains all the possible values for the X variable in T_{gen}. These are (gen)-sequences. ComputeValues(V_{gen-1}[.]) computes the (gen-1)-concatenation of the (gen-1)-sequences for every variable X and returns the frequent (gen)-sequences. Iterate RE-Hackle(T_k, V_k[.], c) performs an RE-Hackle iteration on the T_k tree using the values from V_k[.] for the instanciation of the variables. If T_k is already explored it stops. The third parameter c corresponds to RE-Hackle generation for the k^{th} tree. Promising() is a boolean function which decides wether X-Hackle should continue or not. Just to give a hint, it takes all the concatenations containing variables and computes all the possible First and Last functions. E.g., during the evaluation of (XA)*B, if CAB is not frequent then it is useless to compute the other elements of the closure that finish by CA. The idea is very similar to the one used in the ascending flux of candidates optimisation technique. Formalisation will come in a future publication. If all the First and Last functions from a given node are violated (e.g., they don't generate any frequent sequence) the node is pruned. The algorithm finishes when the X-tree gets empty.

7. Conclusion

We have proposed a framework that characterizes and exploits the local properties of RE-constraints to benefit of both RE-based and frequency-based pruning. Our solution computes dynamically a balance between the two pruning strategies and restricts the search space for arbitrary RE-constraints. It enables to take the shape of the best SPIRIT ad-hoc algorithm without any prior knowledge of the selectivity of the constraint. Thanks to the two-level alphabets, complex expressions are handled efficiently. The cardinalities of a sub-expression appear useful for choosing the extraction method. It enables to introduce global information such as database access costs in the reorganization of the extraction structures. This adaptative strategy associated to powerful optimization techniques such as the transformation of Hackle-trees introduced in [2] or the ascending flux of candidates introduced here enable to tackle RE-constraints even though they are neither anti-monotonic nor monotonic constraints on the search space of the initial symbols. The RE-Hackle approach opens a new framework for the extraction of frequent sequences that satisfy rich syntactic constraints. We did not yet identify all the possible uses of this hierarchical constraint. However, we suspect that a larger (w.r.t. the class of RE-constraints) type of constraint can be handled within this framework. We started to consider RE-constraints with variables. Good properties of the framework are preserved and the potential for applications is clearly enlarged since RE-constraints with variables enable to express context-sensitive restrictions. Our future work concerns further optimizations for the RE-Hackle algorithm. Furthermore, we have to implement X-Hackle and look for a relaxation of the pruning strategy, as the computation of the Last/First sets is expensive. We would like to find a faster termination condition as well.

Acknowledgements. This research is partially funded by Région Rhône-Alpes (programme d'appui à la société Djingle) and the Future and Emerging Technologies arm of the IST programme (project cInQ IST-2000-26469).

References

[1] R. Agrawal, R. Srikant. Mining sequential patterns. Proceedings *ICDE'95, Tapei (Taiwan), 1995. pp. 3-14.*

[2] H. Albert-Lorincz, J-F. Boulicaut. Mining frequent sequential patterns under regular expressions: a highly adaptive strategy for pushing constraints (poster paper). Proceedings *SIAM DM'03, San Francisco (USA), May 1-3, 2003. pp. 316-320.*

[3] M. Garofalakis, R. Rastogi, K. Shim. SPIRIT: Sequential Pattern Mining with Regular Expression Constraints. Proceedings *VLDB'99, Edinburgh (UK), 1999. pp. 223-234.*

[4] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, M.-C. Hsu. FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining. Proceedings *SIGKDD'00, Boston (USA), 2000. pp. 355-359.*

[5] H. Mannila, H. Toivonen. Levelwise search and borders of theories for knowledge discovery. *Data Mining and Knowledge Discovery journal,* Vol. 1(3),1997, pp. 241-258.

[6] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery journal,* Vol. 1(3),1997, pp. 259-289.

[7] M. Leleu, C. Rigotti, J-F. Boulicaut, G. Euvrard. Constraint-based sequential pattern mining over datasets with consecutive repetitions. Proceedings *PKDD'03, Catvat-Dubrovnik (Croatia), 2003.* To appear.

[8] J. Pei et al. PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth. *Proceedings ICDE'01, Heidelberg (D), 2001. pp. 215-224.*

[9] J. Pei, J. Han, and W. Wang. Mining sequential patterns with constraints in large databases. *Proceedings CIKM'02, McLean (USA), 2002. pp. 18-25.*

[10] R. Srikant, R. Agrawal - Mining Sequential Patterns: Generalizations and perfor-mance Improvements. *Proceedings EDBT'96, Avignon (F), 1996. pp. 3-17.*

[11] M. J. Zaki. SPADE: An Efficient Algorithm for Mining Frequent Sequences. *Machine Learning Journal, Vol. 42(1/2*), 2001, pp *31-60.*

[12] M. J. Zaki. Sequence Mining in Categorical Domains: Incorporating Constraints. *Proceedings CIKM'00, Washington (USA), 2000, pp. 422-429.*