

# Mining frequent sequential patterns under regular expressions: a highly adaptative strategy for pushing constraints\*

Hunor Albert-Lorincz<sup>†</sup>

Jean-François Boulicaut<sup>‡</sup>

## Abstract

This paper introduces a new framework for the extraction of frequent sequences satisfying a given regular expression (RE) constraint. Contrary to previous work (SPIRIT algorithms), we represent REs by tree structures and our algorithm can choose dynamically an extraction method according to the local selectivity of the sub-REs. Interestingly, pruning can rely not only on the anti-monotonic minimal frequency constraint but also to the RE constraint that is generally not anti-monotonic. Preliminary experiments on synthetic data have shown that our algorithm takes the shape of the best algorithm from the SPIRIT family and even surpasses it.

## 1 Introduction.

Frequent sequence mining in a database of sequences is an important task [1, 5, 7]. In most of the applications, the lack of user control for specifying the interesting patterns beforehand leads to tedious post-processing phases. Indeed, considering user-defined constraints in conjunction with the minimal frequency is interesting [3, 6]. In this paper, we consider that potentially interesting patterns are specified by a conjunction of the minimal frequency constraint and a regular expression (RE) constraint. In general, RE-constraints are not anti-monotonic and can not be used directly for efficient pruning. We have been studying hierarchical representations of a RE-constraint that can be used to collect information on the properties of the sub RE-constraints. Our algorithm can choose the extraction strategy to favor pruning on minimal frequency or on the RE-constraints. Preliminary experimental results are promising. Due to the lack of space, details about the algorithm, its formal properties but also the experimental results are given in [2].

## 2 The mining task

A sequence is an ordered list (concatenation) of items taken from an alphabet  $\mathcal{A}$ , i.e., a sentence from  $\mathcal{A}^*$ .

$\epsilon$  denotes the empty sequence.  $|S|$  denotes the length of a sequence  $S$ , i.e., the number of items it contains. A database  $\mathbf{d}$  is an unordered collection of sequences.  $S = s_1s_2\dots s_m$  is called a sub-sequence of  $S' = s'_1s'_2\dots s'_n$  if  $\exists k, 0 < k < n$  such that  $s_1s_2\dots s_m = s'_{k+1}s'_{k+2}\dots s'_{k+m}$ . The frequency of a sequence  $S$ , denoted as  $\mathcal{F}(S, \mathbf{d})$ , is the number of the sequences  $S'$  in  $\mathbf{d}$  such that  $S$  is a sub-sequence of  $S'$ .

**Problem statement.** Given a database  $\mathbf{d}$ , a regular expression  $\mathcal{E}$  (associated language  $\mathcal{L}(\mathcal{E}) \subseteq \mathcal{A}^*$ ) and a positive integer  $msup$ , find all the sequences  $S \in \mathcal{A}^*$  which satisfy  $\mathcal{F}(S, \mathbf{d}) \geq msup \wedge S \in \mathcal{L}(\mathcal{E})$ .

**Related work.** Mining frequent sequences has been studied intensively [1, 5, 7, 4]. Various user-defined constraints can be used to focus the mining task on a priori interesting patterns [3, 6]. Constraints can be used to reduce the extraction time and the number of extracted sequences as well. An interesting property of several commonly used constraints (e.g., the minimal frequency) is the anti-monotonicity: it enables efficient pruning. If a non anti-monotonic (nAM) constraint is pushed inside the candidate generation phase, the requirement that the candidates must satisfy both the minimal frequency and the nAM constraint can lack of pruning. Garofalakis et al. have identified this issue in [3] when considering nAM RE-constraints. They have not found a systematic solution and have studied several relaxations of a RE-constraint, giving rise to the four SPIRIT algorithms. Within the SPIRIT framework, the RE  $\mathcal{E}$  is represented by a Finite State Automaton (FSA) which accepts  $\mathcal{L}(\mathcal{E})$ . RE-based pruning is performed by requiring that candidate sequences must correspond to a path fragment of the FSA. The way in which these fragments are chosen defines the different algorithms. For SPIRIT(N), only the symbols which occur in  $\mathcal{E}$  can appear in the candidates. This is basically the GSP framework and the frequent sequences are filtered afterwards against the RE-constraint. For SPIRIT(L), the candidates must match a path fragment in the FSA. It favors frequency-based pruning. For SPIRIT(V), these paths must terminate on a terminal node of the FSA. Frequency-based pruning is less used since some infrequent and non terminal sequences are not generated

\*This research is partially funded by Région Rhône-Alpes (convention Djingle) and the FET arm of the IST programme (European project CInQ IST-2000-26469).

<sup>†</sup>INSA Lyon - LIRIS, F69621 Villeurbanne Cedex, France.

<sup>‡</sup>INSA Lyon - LIRIS, F69621 Villeurbanne Cedex, France.

at all. Finally, for SPIRIT(R), candidates must match complete paths in the FSA and thus they satisfy the RE-constraint. There is no frequency-based pruning and all the possible candidates are generated and counted. SPIRIT algorithms perform better or poorer depending of the *selectivity* of the RE-constraint (selectivity is roughly speaking inversely proportional to the number of sequences in  $\mathbf{d}$  that match the RE-constraint). When the number of sequences in  $\mathcal{L}(\mathcal{E})$  is high (low selectivity), SPIRIT(R) and SPIRIT(V) perform poorly due to a lack of frequency-based pruning. SPIRIT(L) outperforms SPIRIT(R) and SPIRIT(V) when the selectivity is high. Thus, the choice of a SPIRIT algorithm must be based on the unknown selectivity of the RE-constraint. We would like a robust algorithm which depends weakly on the selectivity of the RE-constraint, 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 stored in the database. Our idea is that a FSA is not the best representation since the hierarchical structure of the REs is flattened and information about local selectivity are not explicit.

### 3 The RE-Hackle algorithm.

Let us introduce the RE-Hackle algorithm (Regular Expression Highly Adaptative Constrained Local Extractor) for an efficient extraction of the frequent sequences that match a given RE  $\mathcal{E}$ . RE-Hackle manipulates the representations of this expression and its substructures. We assume that a RE-constraint is represented as a RE built over an alphabet of the so-called *atomic sequences* using the following operators: union (denoted  $+$ ),  $k$ -concatenation (denoted by  $\circ_k$ ,  $\circ_o$  being the usual concatenation) and Kleene closure (denoted  $*$ ).

The  $k$ -concatenation of two sequences  $S$  and  $P$  requires that the sequences overlap in  $k$  positions. For instance,  $AB\circ_o CD = ABCD$  and  $ABCD\circ_2 CDEF = ABCDEF$ . When this condition does not hold, the result of the  $k$ -concatenation is  $\epsilon$ . The  $k$ -concatenation of two sets of sequences is done by  $k$ -concatenating each pair of sequences which belongs to their cartesian product. The union of two sequences  $S$  and  $P$  is the set  $\{S, P\}$  and the union of two sets of sequences is the union of these sets. The Kleene closure applies to a set of sequences and represents all the sequences one can build from them using an arbitrary number of concatenations. It includes  $\epsilon$  as well.

We assume that  $k$ -concatenation and union and have a variable arity. As usually, the priority increases from  $+$  to  $\circ_k$  and from  $\circ_k$  to  $*$ . The concatenation can be distributed over the union. When all the possible concatenations have been performed, the resulting

sequence is called an *atomic sequence*. E.g., the RE-constraint representation  $B\circ_o CD\circ_o E\circ_o A\circ_o (H+F)$  can be transformed into  $BCDEDA\circ_o (H+F)$  by 3 concatenations. According to our definition, BCDEA is a newly formed atomic sequence. Note that H and F were already atomic sequences (impossible to concatenate them to their neighbors), but B, CD, E, A were not as they can be packed together to form a longer sequence. The atomic sequences are the smallest elements that we consider during the extraction.

Using a prefix notation to define precisely the semantics of our representations,  $A+BE+CF+D$  can be represented as  $+(A, BE, CF, D)$ ,  $A(B)^*(CF+D)$  by  $\circ_o(A, *(B), +(CF, D))$ . These abstract representations, the so-called *derivation sentences*, are important (see [2] for details about the grammatical underlying structure) and several useful concepts are based on them.

We say, that a regular expression representation is in a *canonical form* if it contains only atomic sequences. In the following, we assume that all the representations are in the canonical form. A *sub-constraint* is a term of the derivation sentence of a given constraint. Candidate generation is based on the sub-constraints from  $\mathcal{E}$ . These sub-constraints are extracted according to operator priorities. E.g.,  $B+C$  cannot be extracted from  $A\circ_o B+C$  as the priority of  $\circ_o$  prevails over  $+$ . Furthermore, a sub-constraint must contain as many terms as the arity of the operator in the initial constraint  $\mathcal{E}$ . A *maximal sub-constraint* is a sub-constraint, which is not contained in any other sub-constraint except  $\mathcal{E}$ . E.g.,  $A\circ_o B + C$  has two maximal sub-constraints:  $A\circ_o B$  and  $C$ . The maximal sub-constraint defines partitions over the initial RE and can be processed in a distributed parallel environment. An *active operator* connects the maximal sub-constraints of a given constraint. E.g., the active operator for  $A\circ_o B+C$  is  $+$ . A sequence is said *legal* w.r.t. a given RE if one of the sub-constraints of the initial RE matches it exactly. E.g., BD and ABDE are legal w.r.t.  $(A+C)\circ_o BD \circ_o (E+D)$  but ABD is not as the arity of this concatenation is 3, and ABD contains only 2 terms instead of 3.

**Hackle-tree.** A Hackle-tree is an Abstract Syntax Tree 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 length. The tree reflects the way in which these atomic sequences are assembled by the concatenations, unions and Kleene closures to form the initial RE-constraint. Figure 1 provides such a tree for the RE-constraint  $C((C(A+BC)D)+(A+B+C)^*)C$ .

**Cardinalities.** We define the *theoretical cardinality* of a RE-constraint as the number of sequences it can

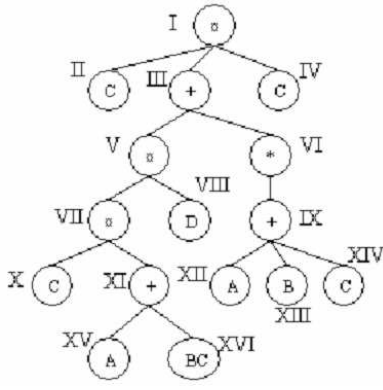


Figure 1: Hackle-tree for  $C((C(A+BC)D)+(A+B+C)^*)C$ .

generate after the expansion of all the operators. The *experimental cardinality* is an estimation of the number of sequences eventually matched by the RE-constraint. While the theoretical cardinality refers only to the RE-constraint, the experimental cardinality takes into account the database, i.e., the results of the counting phases. The theoretical (resp. experimental) cardinality estimates the upper (resp. lower) bound for the number of possible sequences. They converge towards the exact number of sequences represented by the nodes. Let us now specify cardinality computation.

When  $N.explored$  is true,  $\xi_{th}(N) = |N.items|$  and  $\xi_{exp}(N) = |N.items|$ . When  $N.explored$  is false, the cardinalities are computed as follows:

- If  $N.type = \perp$  then
  - $\xi_{th}(N) = 1$
  - $\xi_{exp}(N) = \mathcal{F}(N.items, \mathbf{d}) > msup$
- If  $N.type = \circ_k$  then
  - $\xi_{th}(N) = \prod_{Q \in N.siblings} \xi_{th}(Q)$
  - $\xi_{exp}(N) = \prod_{Q \in N.siblings} \xi_{exp}(Q)$
- If  $N.type = +$  then
  - $\xi_{th}(N) = \sum_{Q \in N.siblings} \xi_{th}(Q)$
  - $\xi_{exp}(N) = \sum_{Q \in N.siblings} \xi_{exp}(Q)$
- If  $N.type = *$  then
  - $\xi_{th}(N) = (N.siblings.\xi_{th})^{max(N.age, 1)}$
  - $\xi_{exp}(N) = (N.siblings.\xi_{exp}) \times |N.items|$

Let us compute  $\xi_{th}$  for the Hackle-tree given in Figure 1.

$$\begin{aligned} \xi_{th}(I) &= \xi_{th}(II) \times \xi_{th}(III) \times \xi_{th}(IV) = 1 * 5 * 1 = 5 \\ \xi_{th}(III) &= \xi_{th}(V) + \xi_{th}(VI) = 2 + 3 = 5 \\ \xi_{th}(V) &= \xi_{th}(VII) + \xi_{th}(VIII) = 2 * 1 = 2 \\ \xi_{th}(VI) &= \xi_{th}(IX) = 3 \\ \xi_{th}(VII) &= \xi_{th}(X) * \xi_{th}(XI) = 1 * 2 = 2 \\ \xi_{th}(IX) &= \xi_{th}(XII) + \xi_{th}(XIII) + \xi_{th}(XIV) \\ &= 1 + 1 + 1 = 3 \\ \xi_{th}(XI) &= \xi_{th}(XV) + \xi_{th}(XVI) = 1 + 1 = 2 \end{aligned}$$

Attribute	Semantics
Type of the node	$\perp$ leaf $\circ_k$ concatenation $+$ union $*$ Kleene closure
Siblings	List of the siblings (null for a leaf)
Parent	Parent (null for root)
$\xi_{th}$	Node theoretical cardinality
$\xi_{exp}$	Node experimental cardinality
Items	Frequent legal sequences found by the node.
State	<i>Unknown</i> if node exploration has not yet begun. <i>Satisfied</i> if some frequent legal sequences have been found. <i>Violated</i> if no frequent sequence has been found.
Explored	True if the node exploration is completed
age	Only for Kleene closure nodes: number of visits on the node
k	Only for k-concatenations
Seq	Only for the leaves: encoded atomic sequence

Table 1: Attributes of the Hackle-tree nodes

It means that using this RE-constraint, it is possible to generate up to  $\xi_{th}(I) = 5$  sequences.

**Extraction phrase.** The list of the nodes of a Hackle-tree one must examine at a given step is called the *extraction phrase*  $\psi$ . When starting the extraction,  $\psi$  contains all the leaves of the tree. This list is updated after each database scan by replacing the explored nodes with their parents.

**Extraction functions.** The *extraction functions*  $C()$  are applied to the nodes of the Hackle-tree, and return the candidate sequences that have to be counted. Let  $N$  denote a node, we define these functions depending of  $N.type$ :

- $\perp$ :  $C(N) = N.seq$
- $\circ_k$ :  $C(N) = \circ_k(M.items), \forall M \in N.siblings$
- $+$ :  $C(N) = +(M.items), \forall M \in N.siblings$   
If  $\exists M \in N.siblings$  with  $M.state \neq Satisfied$  then  $C(N) = \emptyset$ .
- $*$ :  $C(N) = (N.siblings.items) o_{age} (N.items)$

RE-Hackle extracts all the frequent sequences and their frequencies when they match a given RE  $\mathcal{E}$ . At every generation phase, the extraction functions are applied to the nodes in the extraction phrase to generate the candidates. Candidates are counted and the frequent ones are used for the next generation.

A new extraction phrase containing the parent nodes of the examined nodes is built and the Hackle-tree is transformed after each generation. E.g., the branches which can no longer generate new candidates are cut: if any sub-constraint is violated, then the whole node is erased together with its siblings and this information is propagated to its parent.

**An example.** Let us illustrate the execution of our algorithm on the following example database.

Id	Sequences
1	CCADCABC
2	ECBDACC
3	ACCBACFBAC
4	CCBAC

Assume  $msup = 2$  and again  $\mathcal{E} = C((C(A + BC)D) + (A+B+C)*C)$  (Cf. Figure 1). The extraction needs 7 generations and 6 database scans.

• 1st Generation.

$\psi_1 = \text{II, X, XV, XVI, VIII, XII, XIII, XIV, IV}$

Candidates: A, B, C, D, BC

Frequent sequences: A, B, C, D

BC is not frequent and Node XVI is pruned.

• 2nd Generation.

$\psi_2 = \text{VI, VII}$

Candidates: AA, AB, AC, BA, BB, BC, CA, CB, CC

Frequent sequences: AB, AC, BA, CB, CC

$\psi_2' = \text{XI, IX}$  would have been the application of the defined principle. An optimization enables to avoid it [2]. No frequent sequence has been found at Node VII. It can be pruned together with its parent (Node V).

• 3rd Generation ( $age = 1$ ).

$\psi_3 = \text{VI}$

Candidates: ABA, ACB, ACC, BAB, BAC, CBA, CCB, CCC

Frequent sequences: ACC, BAC, CBA, CCB

A Kleene closure node remains in the extraction phrase while it continues to generate frequent sequences. Its  $age$ , here 1, is used for candidate generation.

• 4th Generation ( $age = 2$ ).

$\psi_4 = \text{VI}$

Candidates: ACCB, BACC, CBAC, CCBA

Frequent sequences: CBAC, CCBA

E.g., candidate  $\text{BACC} = \text{BAC} \circ_2 \text{ACC}$ .

• 5th Generation ( $age = 3$ ).

$\psi_5 = \text{VI}$  Candidates: CCBAC

Frequent sequence: CCBAC

• 6th Generation ( $age = 4$ ).

$\psi_6 = \text{VI}$

No candidate since  $\text{CCBAC} \circ_4 \text{CCBAC} = \epsilon$ .

• 7th Generation.

$\psi_7 = \text{I}$

Candidates: CC, CAC, CBC, CCC, CABC, CACC, CBAC, CCAC, CCBC, CCCC, CACCC, CBACC,

CCBAC, CCCBC, CCCBAC, CCBACC, CCCBACC  
Frequent sequences: CC, CBAC, CCBAC

The Kleene closure returns every frequent combination of A, B and C, plus  $\epsilon$ . The root assembles them to C and the frequent items associated to Node I can be returned.

**The algorithm.** The algorithm is given by the following pseudo code:

```

Expr ← CanonicalForm (E);
T ← BuildExpTree(Expr);
K ← 1;
ψ1 ← initPhrase ();
C ← GenCand (ψ1);
While (C ≠ ∅ and ψK ≠ ∅) do
  For all c ∈ C s.t. F(c, d) ≥ msup do
    Node(c).items ← Node(c).items ∪ c;
  UpdateNodes(T, ψK);
  Compute ξexp ∀ N ∈ ψK;
  ψK+1 ← Rebuild(ψK);
  Compute ξth ∀ N ∈ ψK+1;
  Transform(T) given ξth and ξexp ∀ N ∈ ψK+1;
  C ← GenCand(ψK+1);
  PruneCandidates(C);
  K ← K+1;
Return T.root.items;

```

The principal procedures are now introduced (See [2] for details). **CanonicalForm** transforms a RE into its canonical form. **BuildExpTree** builds the Hackle-tree for **Expr**. It looks for its maximal sub-constraints, its active operator, and it creates a new node for it. Then, it recursively proceeds on each maximal sub-constraint that is added as a sibling for the operator node. When **Expr** is an atomic sequence, it creates a leaf. **InitPhrase** computes the initial extraction phrase from the leaves of the Hackle-tree. **GenCand** generates the candidates by applying the extraction functions on the nodes of the extraction phrase. **Node(c)** denotes the node which has generated candidate **c**. **UpdateNodes** updates the nodes after candidate counting. The nodes which have not generated any sequence are marked *Violated* and erased from the Hackle-tree. When a node is violated, its parents are violated in cascade until a node whose type is not a concatenation is reached. The extraction phrase is rebuilt at every generation using **Rebuild**. A node, whose exploration is completed, is replaced by its parent if all its siblings are explored and if it is not in a deleted branch. Only Kleene closures are kept several generations in the extraction phrase as long as they produce frequent sequences. **Transform** rearranges the shape of the tree for balancing frequency-based and RE-based pruning (Cf. next subsection). **PruneCandidates** performs duplicate elimination (the same sequence can be generated from different branches

of the Hackle-tree) and prunes candidates whose subsequences are legal but not frequent.

**Adaptive extraction methods.** Some nodes can generate a large number of candidates without frequency-based pruning. E.g., assume that the siblings A, B, C and D of the concatenation node N return many frequent sequences (High  $\xi_{th}$ ), the extraction method can be adapted to avoid a combinatorial explosion. Indeed, it is possible to group the nodes in larger overlapping buckets to benefit of more frequency-based pruning. E.g., N can be replaced by a new node N2 and a new level (nodes Y, Z and W) to enable pruning:  $N2 = \circ_1(Y, Z, W)$ ,  $Y = \circ_o(A, B)$ ,  $Z = \circ_o(B, C)$ , and  $W = \circ_o(C, D)$ . After the evaluation of the additional level, the number of the candidates should globally decrease, e.g., N2 contains only 3 children and generates less candidates than N. We can introduce more levels and nodes. This mechanism enables a tradeoff between the number of the candidates and the number of database scans: it depends on the size of the database, the counting cost per candidate and the experimental cardinalities of the siblings for the considered node [2].

#### 4 Experimental Results.

Due to the lack of space, we just report here our comparison of RE-Hackle with our implementations of the SPIRIT algorithms. We have used synthetic zipfian distribution data sets (following Zipf's law). The database contains 100k transactions of length 20 over an alphabet of 100 symbols. Granularity is the average length of atomic sequences in RE-constraints. The strength of the pruning strategy of RE-Hackle is somewhat between the one of SPIRIT(L) and SPIRIT(R). SPIRIT(L) uses frequency-based pruning and SPIRIT(R) uses RE-based pruning. In Figure 2, we compare RE-Hackle to SPIRIT(L), SPIRIT(V) and the mean of these two SPIRIT algorithms. Our first experimental results reflect that RE-Hackle takes the shape of the best SPIRIT algorithm: it chooses the best pruning strategy that has been computed according to the cardinality of the constraint and the content of the database.

#### 5 Conclusion.

We have introduced a new characterization of RE-constraints and a new algorithm for sequence mining. Our approach computes a tradeoff between frequency-based pruning and RE-based pruning. It opens a framework for sequential pattern mining under constraints. Not only we suspect that a larger family of constraints can benefit from Hackle-trees but we already identified further optimizations (e.g., dynamic transformation of Hackle-trees) that can boost the performances of the

High $\xi_{th}$ ( $\xi_{th} > 10^6$ , granularity = 1.8)			
	Execution	Scans	Candidates
RE-Hackle	5546	5	24528
SPIRIT(V)	11346	8	25663
SPIRIT(L)	8652	8	8654
Mean	9999	8	-

High $\xi_{th}$ ( $\xi_{th} > 10^6$ , granularity = 1)			
	Execution	Scans	Candidates
RE-Hackle	9152	8	10243
SPIRIT(V)	12135	8	26350
SPIRIT(L)	8604	8	8736
Mean	10369	8	-

Low $\xi_{th}$ ( $100 < \xi_{th} < 1000$ )			
	Execution	Scans	Candidates
RE-Hackle	1450	3	69
SPIRIT(V)	2170	5	129
SPIRIT(L)	2400	5	206
Mean	2285	5	-

Figure 2: Comparison RE-Hackle vs. SPIRIT

basic framework [2]. We are currently studying these issues on real data sets.

#### References

- [1] R. Agrawal and R. Srikant, *Mining sequential patterns*, in Proceedings ICDE'95, IEEE (1995), pp. 3-14.
- [2] H. Albert-Lorincz and J-F. Boulicaut, *Adaptive strategies for pushing regular expression constraints: the RE-Hackle framework*, Research Report INSA Lyon/LIRIS, F-69621 Villeurbanne, 2002.
- [3] M. Garofalakis and R. Rastogi and K. Shim, *SPIRIT: Sequential Pattern Mining with Regular Expression Constraints*, in Proceedings VLDB'99, Morgan Kaufmann (1999), pp. 223-234.
- [4] J. Han and J. Pei and B. Mortazavi-Asl and Q. Chen and U. Dayal and M.-C. Hsu, *FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining*, in Proceedings SIGKDD'00, ACM (2000), pp. 355-359.
- [5] H. Mannila and H. Toivonen, and A. I. Verkamo, *Discovery of frequent episodes in event sequences*, Data Mining and Knowledge Discovery journal, 1(3), 1997, pp. 259-289.
- [6] M. J. Zaki, *Sequence Mining in Categorical Domains: Incorporating Constraints*, in Proceedings CIKM'00, ACM (2000), pp 422-429.
- [7] M. J. Zaki. *SPADE: An Efficient Algorithm for Mining Frequent Sequences*, Machine Learning journal, 42(1/2), 2001, pp 31-60.