Sequence Mining Automata: a New Technique for Mining Frequent Sequences Under Regular Expressions

Roberto Trasarti
Pisa KDD Laboratory
ISTI - CNR, Italy

Francesco Bonchi
Yahoo! Research
Barcelona, Spain

Bart Goethals
Math. and CS Department
University of Antwerp, Belgium

Abstract

In this paper we study the problem of mining frequent sequences satisfying a given regular expression. Previous approaches to solve this problem were focusing on its search space, pushing (in some way) the given regular expression to prune unpromising candidate patterns. On the contrary, we focus completely on the given input data and regular expression. We introduce Sequence Mining Automata (SMA), a specialized kind of Petri Net that while reading input sequences, it produces for each sequence all and only the patterns contained in the sequence and that satisfy the given regular expression. Based on this automaton, we develop a family of algorithms. Our thorough experimentation on different datasets and application domains confirms that in many cases our methods outperform the current state of the art of frequent sequence mining algorithms using regular expressions (in some cases of orders of magnitude).

1. Introduction

Extracting frequent subsequences from a database of sequences [7] is an important data mining task with plenty of different application domains, such as web usage mining, bioinformatics, mobility data analysis, etc. Let $\mathcal{D}$ be a database of sequences, where each $T \in \mathcal{D}$ is a finite sequence of symbols from an alphabet Σ: $T = \langle t_1, \ldots, t_{T_n} \rangle$ where $t_i \in \Sigma, \forall i \in \{1, \ldots, T_n\}$. We denote the set of all possible sequences as Σ^*. A sequence $U \in \Sigma^*$ is a subsequence of a sequence $V \in \Sigma^*$, denoted $U \sqsubseteq V$, if U can be obtained by deleting some symbols from V. More formally, $U = \langle u_1, \ldots, u_m \rangle$ is subsequence of $V = \langle v_1, \ldots, v_n \rangle$ if there are m indices $i_1 < \ldots < i_m$ such that $u_1 = v_{i_1}, \ldots, u_m = v_{i_m}$. The support of a sequence $S \in \Sigma^*$ is the number of sequences in $\mathcal{D}$ that are supersequences of S: $sup_{\mathcal{D}}(S) = |\{T \in \mathcal{D} \mid S \sqsubseteq T\}|$. Given a database $\mathcal{D}$ and a minimum support threshold σ, the set of frequent patterns is $\mathcal{F}(\mathcal{D}, \sigma) = \{S \in \Sigma^* \mid sup_{\mathcal{D}}(S) \geq \sigma\}$. *In this paper we study the problem of mining frequent subsequences satisfying a given regular expression constraint $\mathcal{R}$.*

The set of all patterns denoted by a regular expression $\mathcal{R}$ is usually said the language $L(\mathcal{R})$. Thus the problem we address is the following.

Problem 1 *Given a database $\mathcal{D}$ of sequences over an alphabet Σ, a minimum support threshold σ, and a regular expression $\mathcal{R}$ defined over Σ, compute:*

$$\mathcal{F}(\mathcal{D}, \sigma, \mathcal{R}) = \{S \in L(\mathcal{R}) \mid sup_{\mathcal{D}}(S) \geq \sigma\}$$

Related Work. Agrawal and Srikant introduced the problem of sequential pattern mining, an Apriori-like level-wise algorithm, named GSP, and discussed how to handle time constraints, sliding time window, and user-defined taxonomy [7]. Other methods proposed for mining sequential patterns essentially differs from the Apriori horizontal approach, in the data structure used to index the database: vertical approaches such as SPADE [9] or SPAM [2]; and projection-based approaches such as PREFIXSPAN [6].

The first work introducing regular expression (RE) constraints in sequence mining is SPIRIT by Garofalakis *et. al* [5]. SPIRIT is a family of Apriori-like algorithms whose core is similar to the GSP algorithm. Following the SPIRIT work, Albert-Lorincz and Boulicaut introduced an highly adaptive algorithm named RE-Hackle, which represents regular expressions by means of a tree structure and it is able to choose the extraction method dynamically based on the local sensitivity of the sub-RE [1].

An evolution of SPADE for handling constraints is introduced by Zaki in [8]. The kinds of constraints considered are length and width restrictions, min and max gap, and item constraints. Pei *et al.* extended the projection-based approach of PREFIXSPAN to deal with a very large class of constraints, among which also RE constraints. The efficiency of the resulting algorithm, named prefix-growth, derives from the fact that all this variety of constraints exhibit a *prefix-monotone* property, and thus they result easy to be pushed deeply in the projection-based computation [6].

In the experimental evaluation discussed later, we compare of our methods with prefix-growth and SPIRIT(V) (called simply SPIRIT in the rest of the paper).

Paper Contribution and Organization. In this paper we introduced *Sequence Mining Automata* (SMA), a new mechanism for mining frequent sequences under regular expressions. In the next section we introduce this basic mechanism and prove its correctness. In Section 3 we introduce the first algorithm based on the SMA: SMA-1P. This algorithm simply passes trough the SMA every sequence in the input database, producing all the valid patterns, whose frequency is then counted. In Section 4 we introduce another method which instead uses frequency pruning. The method is named SMA-FC, and it performs a number of scans of the database equals to the number of states of the given regular expression. We further develop SMA-FC by equipping it with a data reduction technique that allows to remove from the database useless sequences as the computation progresses: the resulting algorithm, named SMA-FC*, is introduced in Section 5. The efficiency of our proposal is thoroughly proven empirically in Section 6 on different datasets and application domains.

2. Subsequences Mining Automata

Previous approaches [5, 1, 6] to solve Problem 1 were focusing on its search space, exploiting in different ways the pruning power of the regular expression $\mathcal{R}$ over unpromising patterns. Contrarily, the idea behind our solution is to focus on the input dataset and the given regular expression. In fact, a regular expression may be a very selective kind of syntactical constraint, for which large fraction of an input sequence may result useless w.r.t. counting support for possible valid patterns. The the main idea is to start checking the constraint since the reading of the input database, producing for each sequence in the database, all and only the *valid* (w.r.t. the given regular expression $\mathcal{R}$) patterns contained in the sequence. This is done by means of a *sequence mining automata* (SMA), that we introduce next.

A well-known result from complexity theory states that regular expressions have exactly the same expressive power as *deterministic finite automata* (DFA). Thus, given any regular expression $\mathcal{R}$, we can always build a deterministic finite automaton $\mathcal{A}_\mathcal{R}$ such that $\mathcal{A}_\mathcal{R}$ accepts exactly the language $\mathsf{L}(\mathcal{R})$. However, for our objective we do not need a tool able to recognize the language $\mathsf{L}(\mathcal{R})$, instead we need a tool that given a sequence T it returns all its valid subsequences, i.e., all $\mathsf{S} \in \mathsf{L}(\mathcal{R})$ such that $\mathsf{S} \sqsubseteq \mathsf{T}$. We denote this set as $\mathsf{s}_\mathcal{R}(\mathsf{T}) = \{\mathsf{S} \in \mathsf{L}(\mathcal{R}) | \mathsf{S} \sqsubseteq \mathsf{T}\}$.

For our purposes we adopt a specialization of *Petri Nets*. A Petri Net consists of *places*, *transitions*, and *directed arcs*. Arcs run between places and transitions, and not between places and places or transitions and transitions. The place from which an arc runs to a transition is called the input place of the transition; the place to which arcs run from a transition is called the output place of the transition. Places may contain any number of *tokens*. A distribution of tokens over the places of a net is called a *marking*. Transitions act on input tokens by a process known as *firing*. When a transition fires, it may consume the tokens from its input places, performs some processing task, and places a specified number of tokens into each of its output places. It does this atomically, i.e., in one non-interruptible step. Moreover, transitions can fire concurrently.

Our subsequences mining automata SMA is a specialized kind of Petri Net, which can be constructed from a DFA by transforming each edge of the DFA in a transition with its two arcs from its input place and to its output place. Moreover it has the following peculiarities:

- *Transitions do not consume tokens*.
- *External input*: all the transitions are activated by a global external signal, corresponding to one symbol of the input sequence. For each input sequence we got as many external signals as the symbols in the sequence, and in the proper order. A transition fires only when the actual signal is the one for which the transition is devised. When a transition fires it appends the actual signal to all the tokens in its input place, producing new tokens in its output place, with the constraint that no duplicate tokens are allowed in the same place.
- *Parallel execution*: all the transition activated by the same signal are executed in parallel.

The initial marking consists of only the token representing the empty sequence ε in the starting transition. When all the symbols of the input sequence T have been used as global `signal`, the sequence has been processed and in the acceptance place of the SMA we will find all the valid subsequences of T, i.e., $\mathsf{s}_\mathcal{R}(\mathsf{T})$. The whole process is clarified in the following example.

Example 1 *Given* $\mathcal{R} \equiv \mathsf{A}^*\mathsf{B}(\mathsf{B}|\mathsf{C})\mathsf{D}^*\mathsf{E}$, *we show how the sequence* $\mathsf{T} \equiv \mathsf{ACDBFAEBCFDE}$ *is processed by the* SMA *computing* $\mathsf{s}_\mathcal{R}$. *The whole process is graphically described in Figure 1. We start (initial marking) with only an empty string token* $\langle\varepsilon\rangle$ *in the initial state. The input sequence is processed from left to right. The first symbol of the sequence* A *is sent as signal to the net. There is only one transition aimed at processing* A*: it fires and by appending* A *to the unique token* $\langle\varepsilon\rangle$ *it produces the token* $\langle\mathsf{A}\rangle$ *in its output place. When the second signal* C *is sent to the net there is no transition able to fire, because the unique transition aimed at processing the signal* C *has no token in its input place. So nothing change. Similarly happens for the third symbol* D. *When the fourth symbol, a* B*, is sent to the net, two new tokens* $\langle\mathsf{B}\rangle$ *and* $\langle\mathsf{AB}\rangle$ *are produced by appending* B *to* $\langle\varepsilon\rangle$ *and* $\langle\mathsf{A}\rangle$ *respectively. With the fifth symbol nothing happens. The sixth symbol is again an* A. *The first*

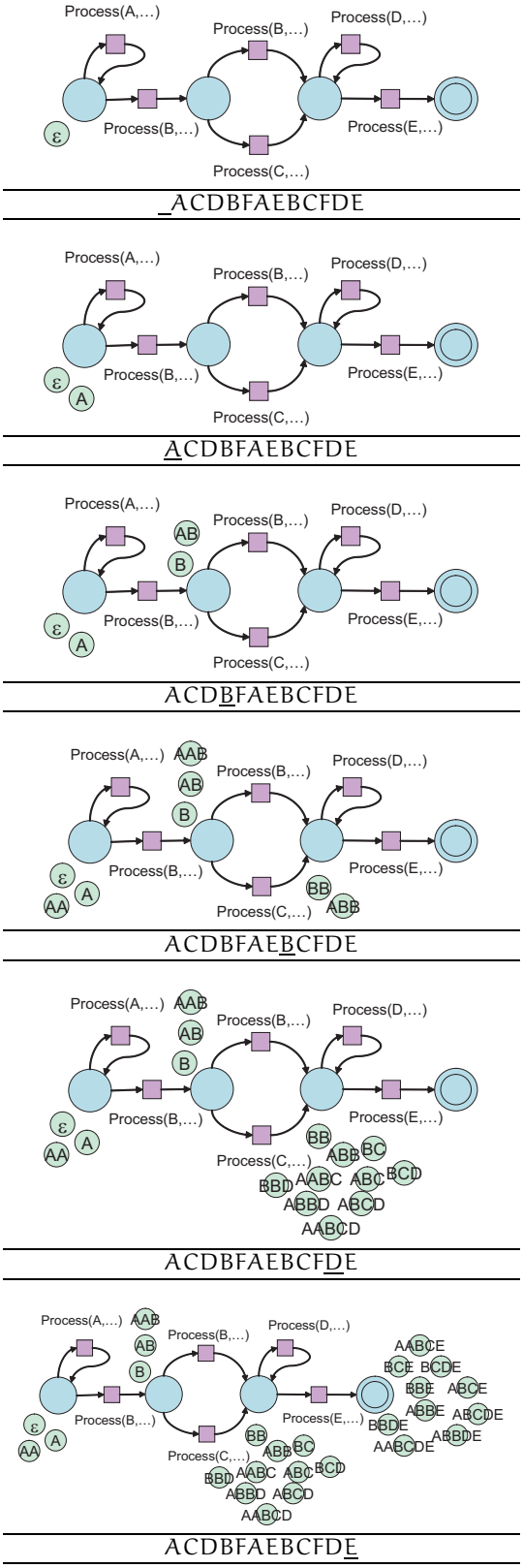


Figure 1. Processing example of the sequence ACDBFAEBCFDE **by** SMA **computing** $s_{\mathcal{R}}$ **where** $\mathcal{R} \equiv A^*B(B|C)D^*E$**.**

transition fires but it produces only a new token $\langle AA \rangle$*: the token* $\langle A \rangle$ *produced by appending* A *to the token* $\langle \varepsilon \rangle$ *is not produced because already present in the output place (that for this transition corresponds to the input place). The process continues this way until the whole sequence has been processed. At the end, the tokens that are in the final place are* $s_{\mathcal{R}}(T)$*. [Proof of correctness omitted due to space constraints.]*

We next discuss three simple extensions to the basic SMA, that allow to deal with patterns that are *strings*, and regular expressions containing *wildcards* and *variables*.

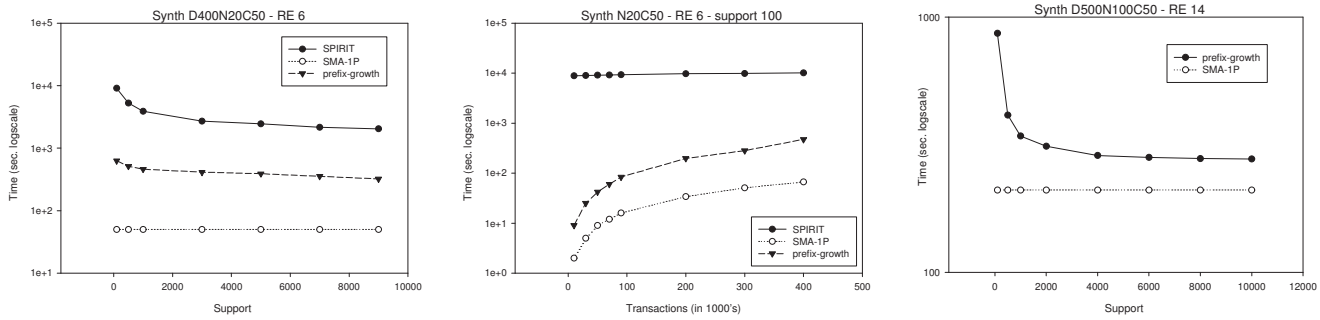
Mining Substrings. In many application domains what is really needed is not subsequences, but substrings, i.e., subsequences made of consecutive symbols. More formally $U = \langle u_1, \ldots, u_m \rangle$ is a substring of $V = \langle v_1, \ldots, v_n \rangle$ if it exist and index i such that $u_1 = v_i, u_2 = v_{i+1}, \ldots, u_m = v_{i+m-1}$. We can easily adapt our methods to produce patterns that are substring by a simple modification to the SMA processing: at each new signal, all tokens except those ones produced by the current signal are deleted.

Allowing Wildcards. It is typical in the biological or chemical domains, to have interesting patterns that contain *holes*, i.e., positions where any symbol can be placed. To handle these kind of patterns we must allow *wildcards* in the regular expression. A wildcard in a regular expression is associated in the SMA to a transition without a proper label: in other terms, a transition that matches any signal, and thus it fires at every iteration.

Allowing Variables. Variables allow to define very expressive regular expressions. They differ from wildcards as once a variable has been bounded to a value, all its other appearances within the same pattern must be bounded to the same value. In the following example variables are represented in lowercase. Consider the regular expression AxBx: the patterns ABBB and ACBC are valid (with $x = B$ and $x = C$ respectively). Allowing variables in our method is achieved by maintaining for each token the list of variables instantiated that it contains. A transition in the SMA associated with a variable, will fire with any signal, but it will produce new tokens starting from input token in which the variable itself has not yet been instantiated or it has been previously instantiated with the same symbol of the current signal.

3. One-pass Solution

In this section we introduce a first very simple, yet very efficient way to exploit the sequence mining automata. The proposed solution has two interesting features: (1) it performs a unique pass over the input database, (2) it is minimum support threshold independent. Obviously point 2, is not only a nice feature, it is also a strong limitation as it can be read also as "it does not perform any frequency-based pruning". This issue will be discussed later. The method,



(a) (b) (c)

Figure 2. Run-time comparison between $SMA-1P$, SPIRIT, and prefix-growth at different minimum support thresholds, datasets and regular expressions.

named SMA-1P (SMA *one pass*) is described in Algorithm 1. SMA-1P just processes by means of the SMA all the input sequences T one by one, and enters all resulting valid patterns $s_{\mathcal{R}}(T)$ in a hash table HT for support counting. After the whole input database $\mathcal{D}$ has been processed, the hash table is visited and frequent patterns are outputted.

Algorithm 1 SMA-1P

Input: $\mathcal{D}$, σ, $\mathcal{R}$
Output: $\{S \in L(\mathcal{R}) \mid sup_{\mathcal{D}}(S) \geq \sigma\}$

```
1: for all T ∈ D do
2:    compute s_R(T) using the SMA corresponding to R
3:    for all S ∈ s_R(T) do
4:       if S in HT then
5:          HT[S].count++
6:          else insert S in HT with HT[S].count = 1
7: for all I ∈ HT do
8:    if HT[I].count ≥ σ then
9:       output I
```

In Figure 2 we report run-time comparison between SPIRIT, prefix-growth and SMA-1P (details on experiments settings are provided later in Section 6). Our method, albeit so simple is very efficient, in some cases outperforming of two orders of magnitude SPIRIT, and one order of magnitude prefix-growth as evident in Figure 2(a). The gap between the two previous methods and SMA-1P increases as the minimum support threshold shrinks. SMA-1P is support-independent and thus its run-time stays constant as the support changes, while run-time of the two other methods explodes for small supports. This consideration makes SMA-1P a very good algorithm for mining at low support levels. Figure 2(b) shows that the run-time of SMA-1P increases linearly with the size of the dataset. A deeper analysis on the behavior of SMA-1P and prefix-growth is reported in Figure 3(a),(b), and (c). The three plots confirm that, as predictable, the performance of our simple method degrades when the selectivity of the regular expression constraint shrinks. In particular, for a given RE, if we reduce the number of symbols in the alphabet (called "items" in the plots), then larger part of the input sequences will match the RE, i.e, the constraint is less selective. The same holds also for prefix-growth. However we can observe that, as the RE grows in size (and thus become less selective) the difference between the two methods shrinks: i.e., SMA-1P is much faster than prefix-growth on RE6, the difference is reduced for R10, and for R14 we have that in the worst selectivity case (an alphabet of 20 symbols) prefix-growth outperforms SMA-1P. In conclusion, when the RE is not selective enough, our method's performance degrades. In these cases we must rely on frequency-based pruning, as presented in the next section.

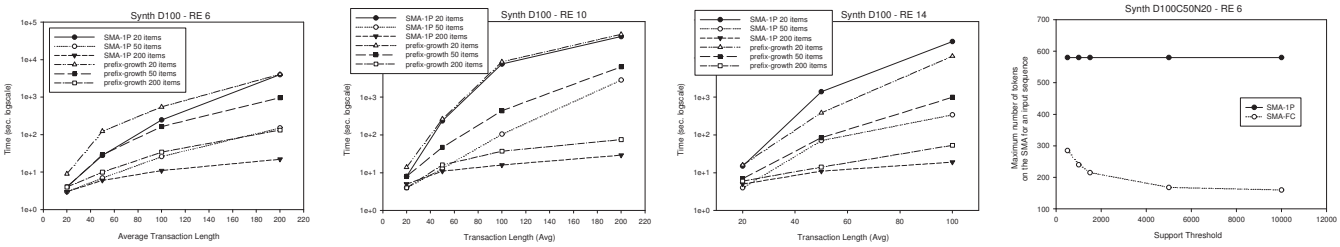
4. Pushing Frequency

The well known *anti-monotonicity* property of frequency is usually exploited by almost all pattern mining algorithms by pruning from the search space super-patterns of patterns found infrequent. One simple way of using the same property with our SMA is to break the computation in parts by introducing a set of cuts in the SMA, and check global frequency of the various tokens before proceeding in the computation. Given a SMA a *valid set of cuts* is a partition $p_1, \ldots, p_n$ of the places of the SMA such as does not exist a path from a place in p_j to a place in p_i if $j > i$.

Intuitively, by giving a set of cuts $p_1, \ldots, p_n$ we give a series of n SMAs, where the SMA corresponding to p_i is the one containing all places $p_1 \cup p_2 \cup \ldots \cup p_i$. In turn, these SMAs correspond to n regular expressions $\mathcal{R}_1, \ldots, \mathcal{R}_n$ for which the following properties trivially hold.

Property 1 *Given a regular expression $\mathcal{R}$, its corresponding* SMA *and a set of cuts $p_1, \ldots, p_n$, it holds that $\forall 1 \leq i \leq j \leq n : \forall S \in L(\mathcal{R}_i), \exists S' \in L(\mathcal{R}_j) : S \sqsubseteq S'$; and $\forall S' \in \mathcal{F}(\mathcal{D}, \sigma, \mathcal{R}_j), \exists S \in \mathcal{F}(\mathcal{D}, \sigma, \mathcal{R}_i) : S \sqsubseteq S'$; and $\mathcal{R}_n = \mathcal{R}$.*

Following the properties above, the process starts with the reduced SMA containing only the places in p_1. The SMA adopted during the second scan is the one given by



(a) (b) (c) (d)

Figure 3. (a),(b),(c): run-time of SMA-1P and prefix-growth run-time for different problem settings, and regular expressions. The minimum support used is $\sigma = 500$. (d): maximum number of tokens produced on the SMA for an input sequence.

$p_1 \cup p_2$, during the third scan $p_1 \cup p_2 \cup p_3$ and so on. At the end of each scan, similarly to what done for the one-pass solution, the tokens contained in the final place are memorized in the hash table HT for support counting. This way, at the end of the $i_t h$ scan we obtain an intermediate information about frequent patterns, i.e., $\mathcal{F}(\mathcal{D}, \sigma, \mathcal{R}_j)$, that can be used in subsequent scans by removing the infrequent tokens. At the end we have a number of scans equal to the number of partitions n. Practically, this is obtained by changing the operator that is performed by the transitions of the SMA in such a way that only tokens that have not been previously find infrequent are generated.

The method described above is general and does not specify how the cuts are provided. An interesting line of research is that of developing a method that, given the regular expression, is able to decide how to cut the corresponding SMA in such a way to optimize the trade-off between number of databases scans and frequency-based pruning. We do not develop such a method in this paper, and we leave it for future investigation. In this paper we adopt a simple solution. We assume to have a cut after every place of the SMA. This means that at the first scan of the input database we use only the first place of the SMA, at the second iteration only the first two places, and so on. This way we collect information about infrequent patterns that can be exploited by the subsequent scans allowing frequency based pruning of unpromising tokens. We call this approach SAM-FC (SMA *Full Check*). Figure 3(d) shows the benefits of the frequency-based pruning in terms of the maximum number of tokens produced over the SMA for an input sequence.

5. Adding data-reduction

Data reduction techniques have been used successfully in the context of constrained frequent set mining [3]. In our context we can use push data reduction techniques, whenever multiple scans of the database are performed, i.e., for SMA-FC. In particular the following property holds.

Property 2 *Given a regular expression $\mathcal{R}$, its corresponding SMA and a set of cuts $p_1, \ldots, p_n$, it holds that: $s_{\mathcal{R}i}(T) = \emptyset \Rightarrow s_{\mathcal{R}j}(T) = \emptyset, \forall j \geq i$.*

Exploiting this property, we enhance our method by checking, for each sequence in the input database, if it generates at least one token in the border of the current cut, otherwise the sequence can be pruned from the input database.

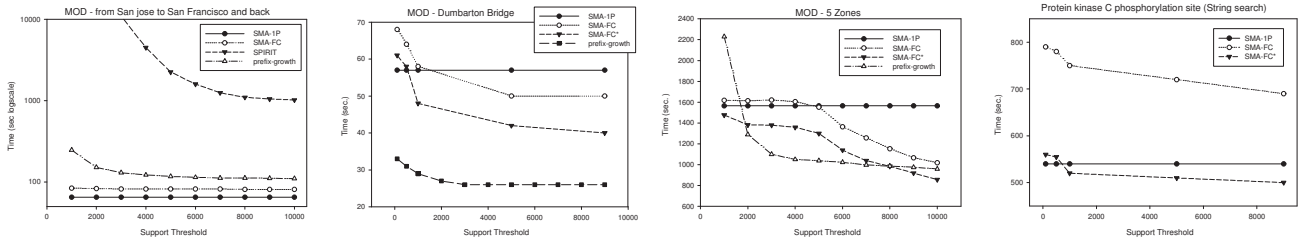
With this data reduction the total number of sequences decreases as the computation progresses from a cut to the following. Note that this data reduction is strengthened by the frequency constraint: as less tokens survive in the process, as more transactions are pruned. Moreover also the opposite holds: as more transactions are pruned as less tokens are found frequent, and thus the data reduction strengthen the frequency-based pruning. We denote our method equipped with data reduction as SMA-FC*.

6. Experimental evaluation

We implemented our methods, prefix-growth and SPIRIT in C++ with Visual Studio 2003 and using STL. All the experiments have been run on a Windows XP machine equipped with AMD Athlon XP processor, 2.09 GHz, 2 Gb RAM. All the software developed together with all the datasets used in the experimentation can be downloaded from `http://www-kdd.isti.cnr.it/SMA/`. Next we describe the datasets and the RE we experimented with.

Synthetic Data. Several synthetic datasets have been generated using the IBM dataset generator [7]. Each dataset is named by its number of sequences (in thousands), size of alphabet, and average size of sequences: for instance the dataset D400N20C50 contains 400000 sequences over an alphabet of 20 symbols with an average length of 50 symbols per sequence. The regular expressions experimented over these synthetic datasets are: RE6 $\equiv$ A*B(B|C)D*E, RE10 $\equiv$ A*B(B|C)D*EF*(G|H)I*, and RE14 $\equiv$ A*(M|(N*B(B|C))D*E(N|F)*(G|H)I*L.

Moving Objects Data. We generated a large database containing 250000 trajectories using Brinkhoff's network-based synthetic generator of moving objects [4], over the San Francisco Bay Area map. In order to obtain sequences from the trajectories, the map has been discretized using a 17*17 regular grid, obtaining an alphabet Σ of 289 symbols. Regular expressions we use in our experiments are:



(a) (b) (c) (d)

Figure 4. Run time comparisons on different datasets and regular expressions.

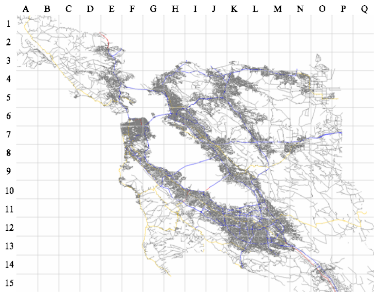


Figure 5. The San Francisco Bay Area map.

From San Jose to San Francisco and back – via CA-101 (west-bound of the bay), i.e., passing through San Mateo (cell H9 of our map); or via I-880 (est-bound of the bay), i.e., passing through Hayward (cell J8 of our map)[1].

Dumbarton Bridge – Find patterns that goes from Palo Alto (I10) to San Francisco (F7). We want also to see those patterns that still goes from Palo Alto to San Francisco but that starts in San Jose (L12). Considering that the zone between I10 and H9 is congested by traffic, we want to know if someone decides to cross the Dumbarton Bridge to avoid the congested area[2].

Five zones – Find patterns passing trough 5 different zone, from north to south[3].

Protein Sequences. Protein is an interesting application domains, for the small size of the alphabet, and the length of the sequences. Moreover in this domain, regular expressions may be used to express meaningful and interesting patterns. The dataset is downloaded from the Entrez database at NCBI/NIH[4]. It contains 103120 sequences over an alphabet of 24 symbols, and average sequence length equals to 482. As RE we used one representing *Protein kinase C phosphorylation site*[5]: i.e., $\mathcal{R}_2 \equiv (S|T) \cdot (R|K)$ (where $\cdot$ represent the wildcard).

[1] $\mathcal{R} \equiv$ (K11|L11|M11|K12|L12|M12|K13|L13|M13)(H9|J8)(F6|F7|F8|G7)(K11|L11|M11|K12|L12|M12|K13|L13|M13).

[2] $\mathcal{R} \equiv$ L12*I10(I10|J9)H9*F7.

[3] $\mathcal{R} \equiv$ (D4|E4|F4|G4|H4|I4|D5|E5|F5|G5|H5|I5)(E6|F6|G6|H6|I6|J6|E7|F7|G7|H7|I7|J7)(F8|G8|H8|I8|J8|K8|F9|G9|H9|I9|J9|K9)(G10|H10|I10|J10|K10|L10|G11|H11|I11|J11|K11|L11)(M11|I12|J12|K12|L12|M12|N12|J13|K13|L13|M13|N13).

[4] http://www.ncbi.nlm.nih.gov/sites/entrez

[5] http://www.expasy.org/prosite/PDOC00005

Run-time comparison. Figure 4 reports run-time comparison among the various methods: results change a lot with the different regular expressions. In Figure 4(a) we report the run-time of SPIRIT (note the logscale), while in the other plots we avoid it, as SPIRIT is always much slower than all the other methods. Surprisingly in Figure 4(a), the direct approach of SMA-1P is the fastest one. In Figure 4(b) instead prefix-growth outperforms all the other methods. In this plot we can also appreciate, as the minimum support threshold grows, the benefits of the frequency-based pruning in SMA-FC, and the additional benefits of the data reduction technique in SMA-FC*. These benefits are also evident in Figure 4(c). Interestingly, in this plot prefix-growth is the fastest method for some support thresholds, but it performs very poorly for small supports, while for large supports it is outperformed by SMA-FC*.

Figure 4(d), reports run-time on the protein dataset in the case of the string search showing that the direct approach of SMA-1P is very effective. In fact at every new step only the newly created tokens are kept on the SMA: this reduces a lot the advantage of frequency-based pruning, making it not worth the price of multiple database scans. This explains the poor performance of SMA-FC. However, the combination of frequency-based pruning and data reduction technique of SMA-FC* performs very well on strings.

References

[1] H. Albert-Lorincz and J.-F. Boulicaut. Mining frequent sequential patterns under regular expressions: A highly adaptive strategy for pushing contraints. In *Proc. of SDM'03*.

[2] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *KDD'02*.

[3] F. Bonchi and B. Goethals. FP-Bonsai: the art of growing and pruning small fp-trees. In *Proceedings of PAKDD'04*.

[4] T. Brinkhoff. Generating traffic data. *IEEE Data Eng. Bull.*, 26(2):19–25, 2003.

[5] M. N. Garofalakis, R. Rastogi, and K. Shim. Spirit: Sequential pattern mining with regular expression constraints. In *Proceedings of VLDB'99*.

[6] J. Pei, J. Han, and W. Wang. Mining sequential patterns with constraints in large databases. In *Proc. of CIKM'02*.

[7] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *EDBT'96*.

[8] M. J. Zaki. Sequence mining in categorical domains: Incorporating constraints. In *Proceedings of CIKM'00*.

[9] M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1/2):31–60, 2001.