2016 IEEE 16th International Conference on Data Mining

DESQ: Frequent Sequence Mining with Subsequence Constraints

Kaustubh Beedkar
University of Mannheim, Germany
Email: kbeedkar@uni-mannheim.de

Rainer Gemulla
University of Mannheim, Germany
Email: rgemulla@uni-mannheim.de

***Abstract*—Frequent sequence mining methods often make use of constraints to control which subsequences should be mined; e.g., length, gap, span, regular-expression, and hierarchy constraints. We show that many subsequence constraints—including and beyond those considered in the literature—can be unified in a single framework. In more detail, we propose a set of simple and intuitive "pattern expressions" to describe subsequence constraints and explore algorithms for efficiently mining frequent subsequences under such general constraints. A unified treatment allows researchers to study jointly many types of subsequence constraints (instead of each one individually) and helps to improve usability of pattern mining systems for practitioners.**

I. Introduction

Frequent sequence mining (FSM) is a fundamental task in data mining. Frequent sequences are useful for a wide range of applications, including market-basket analysis [1], web usage mining and session analysis [2], natural language processing [3], information extraction [4], [5], or computational biology [6]. In web usage mining, for example, frequent sequences describe common behavior across users (e.g., the order in which users visit web pages). As another example, frequent textual patterns such as "*PERSON* is married to *PERSON*" are indicative of typed relations between entities and useful for natural-language processing and information extraction tasks [4], [5].

In FSM, we model the available data as a collection of sequences composed of items such as words (text processing), products (market-basket analysis), or actions and events (session analysis). Often items are arranged in an application-specific hierarchy; e.g., *is→be→VERB* (for words), *Canon 5D→DSLR camera→electronics* (for products), or *Rakesh Agrawal→scientist→PERSON* (for entities). The goal of FSM is to discover subsequences or generalized subsequences that occur in sufficiently many input sequences. Since the total number of such subsequences can potentially be very large and not all frequent subsequences may be of interest to a particular application, most FSM methods make use of subsequence constraints to control the set of subsequences to be mined.

A large variety of subsequence constraints has been studied in prior work [1], [7]–[13]. Commonly proposed constraints include *gap or span constraints*, where items in the subsequences need to appear "close" in the input sequence, and *length constraints*, where the number of items in the subsequences is bounded. In n-gram mining [14], for example, the goal is to mine frequent consecutive subsequences of exactly n words. *Hierarchy constraints* allow controlled generalization according to the item hierarchy to find patterns which do not directly occur in the input data. Examples include shopping patterns such as "customers frequently buy some *DSLR camera*, then some *tripod*, then some *flash*" or textual patterns such as "*PERSON be* born in *LOCATION*". *Regular expression (RE) constraints* have also been studied in the context of FSM; here subsequences must match a given RE.

A number of specialized algorithms for various combinations of the above subsequence constraints have been proposed in the literature. In this paper, we show that many subsequence constraints—including and beyond those described above—can be unified in a single framework. A unified treatment allows researchers to study subsequence constraints in general instead of focusing on certain combinations individually. It also helps to improve usability of pattern mining systems for practitioners because it avoids the need to develop customized mining algorithms for the particular subsequence constraint of interest. In this work, we focus on the questions of (1) how to model and express subsequence constraints in a suitable way and (2) how to mine efficiently all frequent sequences that satisfy the given constraints.

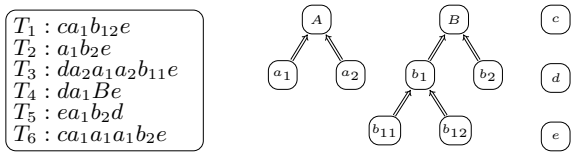
In more detail, we introduce *subsequence predicates* to model subsequence constraints in a general way, and we propose a simple and intuitive *pattern expression language* to concisely express subsequence predicates. Our pattern expressions are based on regular expressions, but—in contrast to prior work on RE-constrained FSM—target input sequences and support capture groups and item hierarchies. Capture groups are the key ingredient for expressing most prior subsequence constraints in a unified way; see Tab. I for examples. Direct support for item hierarchies allows us both to express subsequence constraints concisely and to mine generalized subsequences in a controlled way. Some example pattern expressions as well as anecdotal results are given in Tab. III.

To mine frequent sequences, we propose to use finite state transducers (FST) as the underlying computational model. To the best of our knowledge, FSTs have not been studied in the context of FSM before. We propose the DESQ system, which includes two efficient mining algorithms termed DESQ-COUNT and DESQ-DFS. Both algorithms translate a given pattern expression to a *compressed* FST, which is subsequently optimized and simulated in a way suitable for frequent sequence mining. DESQ-COUNT is a match-and-count algorithm that aims at highly selective constraints, whereas DESQ-DFS can handle more demanding pattern expressions and is inspired by PrefixSpan [11]. Our experimental study on various real-world datasets suggests that DESQ is an efficient general-purpose FSM framework and competitive to state-of-the-art specialized algorithms.

II. Preliminaries

Sequence database. A *sequence database* is a multiset of sequences, denoted $\mathscr{D} = \{ T_1, T_2, \ldots, T_{|\mathscr{D}|} \}$. Each *sequence* $T = t_1 t_2 \ldots t_{|T|}$ is an ordered list of items from a *vocabulary* $\Sigma = \{ w_1, w_2, \ldots, w_{|\Sigma|} \}$. We denote by ϵ the empty sequence, by $|T|$ the length of sequence T, by Σ^* (Σ^+) the set of all (all non-empty) sequences that can be constructed from items in Σ. Fig. 1(a) shows an example sequence database $\mathscr{D}_{ex}$ consisting of six sequences.

2374-8486/16 $31.00 © 2016 IEEE
DOI 10.1109/ICDM.2016.110



(a) Example database (b) Example hierarchy

Fig. 1: A sequence database and its vocabulary

Item hierarchy. The items in Σ are arranged in an *item hierarchy*, which expresses how items can be generalized (or that they cannot be generalized). Fig. 1(b) shows an example hierarchy in which, for example, item a_1 generalizes to item A. In general, we say that an item u *directly generalizes to* an item v, denoted $u \Rightarrow v$, if u is a child of v. We further denote by $\Rightarrow^*$ the reflexive transitive closure of $\Rightarrow$. For the example of Fig. 1(b), we have $b_{11} \Rightarrow b_1$, $b_1 \Rightarrow B$ and $b_{11} \Rightarrow^* B$. For each item $w \in \Sigma$, we denote by $\mathrm{anc}(w) = \{ w' \mid w \Rightarrow^* w' \}$ the set of *ancestors* of w (including w) and by $\mathrm{desc}(w) = \{ w' \mid w' \Rightarrow^* w \}$ the set of *descendants* of w (again, including w). In our running example, we have $\mathrm{anc}(b_1) = \{ b_1, B \}$ and $\mathrm{desc}(b_1) = \{ b_1, b_{11}, b_{12} \}$.

Subsequence. Let $S = s_1 s_2 \dots s_{|S|}$ and $T = t_1 t_2 \dots t_{|T|}$ be two sequences composed of items from Σ. We say that S is a *generalized subsequence* of T, denoted $S \sqsubseteq T$, if S can be obtained by deleting and/or generalizing items in T. More formally, $S \sqsubseteq T$ iff there exists integers $1 \leq i_1 < i_2 < \dots < i_{|S|} \leq |T|$ such that $t_{i_k} \Rightarrow^* s_k$ for $1 \leq k \leq |S|$. Continuing our example, we have $cBe \sqsubseteq T_1$, $ca_1 \sqsubseteq T_1$ and $a_1c \not\sqsubseteq T_1$.

III. FSM With Subsequence Constraints

Our goal is to provide a general framework to express subsequence constraints, including and beyond previously proposed constraints. Consider the following (admittedly contrived) subsequence constraint as an example of a subsequence constraint.

Example 1. *Consider our example database $\mathscr{D}_{ex}$ and suppose that we are interested in mining sequences of B's and/or descendants of A's. We restrict attention to sequences that occur consecutively in input sequences starting with c or d and ending with e. We also allow to generalize occurrences of descendants of A and B. Then $a_1B \sqsubseteq T_1$ and $AB \sqsubseteq T_1$ satisfy this subsequence constraint, whereas $a_1b_{12} \sqsubseteq T_1$, $a_1b_1 \sqsubseteq T_1$, $a_1B \sqsubseteq T_2$ and $AB \sqsubseteq T_2$ do not.*

The above subsequence constraint cannot be expressed using prior methods. Note that the constraint combines (i) a gap constraint (consecutive), (ii) a hierarchy constraint (descendants of B must be generalized), and (iii) a context constraint (between c or d, and e).

Subsequence predicates. We propose subsequence predicates as a general, natural model for subsequence constraints. A *subsequence predicate* P is a predicate on pairs (S, T), where $T \in \Sigma^+$ is any input sequence and $S \sqsubseteq T$ is a subsequence. Subsequence $S \sqsubseteq T$ satisfies the constraint when $P(S, T)$ holds. Note that P is not a predicate on (only) subsequence S; it also involves input sequence T. We denote by $G_P(T) = \{ S \sqsubseteq T \mid P(S, T) \}$ the set of *P-subsequences* in T. For each $S \in G_P(T)$, we say that S is *P-generated* by T. For example, let P_{ex} be the subsequence predicate that expresses subsequence constraint of Ex. 1, then $G_{P_{ex}}(T_1) = \{a_1B, AB\}$ and $G_{P_{ex}}(T_2) = \emptyset$.

Subsequence predicates can encode different application needs, including but not limited to the various subsequence constraints discussed before. A subsequence predicate can act as a filter on the set of all subsequences of T (only A's and B's), but may also consider the context in which these subsequences occur (consecutively between c or d and e). In practice, we may construct subsequence predicates that generate all n-grams, all adjective-noun pairs, all relational phrases between named entities, all electronic products, or, in log mining, sequences of items that occur before and/or after an error item. We propose a suitable way to express subsequence predicates in Sec. IV.

FSM and subsequence predicates. Let P be a subsequence predicate. The *P-support* $\mathrm{Sup}_P(S, \mathscr{D}) = \{ T \in \mathscr{D} \mid S \in G_P(T) \}$ of sequence $S \in \Sigma^+$ in database $\mathscr{D}$ is the *multiset* of all sequences in $\mathscr{D}$ that P-generate S.

The *P-frequency* of S in $\mathscr{D}$ is given by $f_P(S, \mathscr{D}) = |\mathrm{Sup}_P(S, \mathscr{D})|$. In our example database, we have $\mathrm{Sup}_{P_{ex}}(Aa_1AB, \mathscr{D}_{ex}) = \{ T_3, T_6 \}$ and thus $f_{P_{ex}}(Aa_1AB, \mathscr{D}_{ex}) = 2$.

Given a *support threshold* $\sigma > 0$, we say that a sequence S is *P-frequent* if $f_P(S, \mathscr{D}) \geq \sigma$. Our goal is to find all P-frequent sequences $S \in \Sigma^+$ along with their frequencies. The set of all P_{ex}-frequent sequences for $\sigma = 2$ in our example database is given by $\{AAAB{:}2, AB{:}2, Aa_1AB{:}2, a_1B{:}2\}$, where we also give P-frequencies.

IV. Pattern Expressions

We propose a pattern language for expressing subsequence predicates in a simple and intuitive way. Our language is based on regular expressions, but adds features that allows us to unify many prior subsequence constraints. We subsequently suggest a computational model based on FSTs, and describe the formal semantics of our language.

A. Pattern Language

Our language consists of the following set of *pattern expressions*, defined inductively: (1) For each item $w \in \Sigma$, the expressions w, $w_{=}$, $w^{\uparrow}$, and $w^{\uparrow}_{=}$ are pattern expressions. (2) . and $.^{\uparrow}$ are pattern expressions. (3) If E is a pattern expression, so are (E), $[E]$, $[E]^*$, $[E]^+$, $[E]?$, and for all $n, m \in \mathbb{N}$ with $n \leq m$, $[E]\{n\}$, $[E]\{n,\}$, and $[E]\{n,m\}$. (4) If E_1 and E_2 are pattern expressions, so are $[E_1E_2]$ and $[E_1|E_2]$.

Pattern expressions are based on regular expressions, but additionally include capture groups (in parentheses), hierarchies (by omitting $_{=}$), and generalizations (using $^{\uparrow}$ and $^{\uparrow}_{=}$). We make use of the usual precedence of rules for regular expressions to suppress square brackets (but not parentheses); operators that appear earlier in the above definition have higher precedence. We refer to expressions of form (1) or (2) as *item expressions*. We write $G_E(T)$ to refer to the set of subsequences "generated" by expression E on input T (see Sec. IV-B for a formal definition).

Captured and uncaptured expressions. Pattern expressions specify which subsequences to output (captured) as well as the context in which these subsequences should occur (uncaptured). We make use of parentheses to distinguish these two cases; the semantics is similar to the use of capture groups in regular expressions. Given an expression E, only subexpressions that are enclosed in or contain a capture group will produce non-empty output; all other subexpressions serve to describe context information. For example, the pattern expression $E_{ex} = [c|d]([A^{\uparrow} \mid B^{\uparrow}_{=}]^+)e$ describes precisely the subsequence constraint of Ex. 1. Here subexpressions $[c|d]$ and e describe context and $([A^{\uparrow} \mid B^{\uparrow}_{=}]^+)$ output.

Item expressions. Item expressions are the elementary form of pattern expressions and apply to one input item. If the item expression "matches" the input item, it can "produce" an output item; see Tab. II for an overview. Fix some $w \in \Sigma$. The most basic item expression is $w_{=}$: it matches only item w and

TABLE I: Pattern expr. for prior subsequence constraints

Subsequence constraint	*Example*	Pattern expression
All subsequences [1], [11], [15]		$[.^*(.)]^+$
Bounded length [13]	*length 3–5*	$[.^*(.)]\{3,5\}$
n-grams [10], [14]	3-, 4- *and* 5-*grams*	$(.)\{3,5\}$
Bounded gap [10], [13]	*each gap at most 3*	$(.)[.\{0,3\}(.)]+$
Serial episodes [16]	*length 3, total gap* ≤ 2	$(.)[.?.?(.) \mid .?(.).? \mid (.).?.?](.)$
Hierarchy [1], [8]	*generalized 5-grams*	$(.^\uparrow)\{5\}$
Regular expression [9], [12], [17], [18]		
	subsequences matching $[a\|b]\,c^*d$	$(a\|b)[.^*(c)]^*.^*(d)$
	contiguous subsequences matching $[a\|b]\,c^*d$	$([a\|b]\,c^*d)$

produces either ϵ (if uncaptured) or w (if captured). Using our example hierarchy of Fig. 2(a), we have $G_{A_=}(A) = \emptyset$ (note that we ignore output ϵ), $G_{(A_=)}(A) = \{\, A \,\}$, and $G_{(A_=)}(a_1) = \emptyset$. Sometimes we do not want to only match the specified item but also all of its descendants in the item hierarchy (e.g., we want to match all nouns in text mining). Item expression w serves this purpose: it matches any item $w' \in \mathrm{desc}(w)$ (which includes w) and, when captured, produces the item that has been matched. For example, we have $G_{(A)}(A) = \{\, A \,\}$, $G_{(A)}(a_1) = \{\, a_1 \,\}$, and $G_{(A)}(b_1) = \emptyset$. Our language also provides *wild card* symbol "." to match any item; again, the matched item is produced when the wild card is captured. For example, $G_{(.)}(A) = \{\, A \,\}$, and $G_{(.)}(a_1) = \{\, a_1 \,\}$.

To support mining with controlled generalizations (e.g., to mine patterns such as "*PERSON* lives in *CITY*"), we use the *generalization operator* $^\uparrow$, which generalizes items along the hierarchy. Item expressions that use the generalization operator must be captured. More specifically, item expression $w^\uparrow$ matches any item $w' \in \mathrm{desc}(w)$—as expression w does—, and it produces either the matched input item or any of its ancestors that is also a descendant of w. For example, $G_{(B^\uparrow)}(b_{12}) = \{\, b_{12}, b_1, B \,\}$ and $G_{(b_1^\uparrow)}(b_{12}) = \{\, b_{12}, b_1 \,\}$. We also allow the use of a wild card with generalization operator: expression "$.^\uparrow$" matches any item and produces each of its generalizations. For example, $G_{(.^\uparrow)}(b_1) = \{\, b_1, B \,\}$. Our final item expression is used to enforce a generalization: $w^\uparrow_=$ matches any descendant of w and produces w, independently of which descendant has been matched. For example $G_{(B^\uparrow_=)}(b_{12}) = \{\, B \,\}$.

Composite expressions. Item expressions can be arbitrarily combined using operators ? (optionality), $*$ (Kleene star), $+$ (Kleene plus), $\{n, m\}$ (bounded repetition), $|$ (union), and concatenation to match (sequences of) more than one input item. The semantics of these compositions is as in regular expressions.

Examples. Our pattern expressions allow us to express many existing subsequence constraints in a unified way; see Tab. I for some examples. Note that the use of capture groups enables many of these pattern expressions. Pattern expressions can additionally express many customized subsequence constraints that cannot be handled by existing FSM frameworks; see Tab. III for some examples.

B. Computational Model

We translate patterns expressions into FSTs, which are a natural computational model for pattern expressions. An FST is a type of finite state machine for string-to-string translation [19]. FSTs are similar to finite state automata but additionally label transitions with output strings. Conceptually, an FST reads an input string and translates it to an output string in a nondeterministic fashion. We will use FSTs to specify subsequence predicate $P(S, T)$: the predicate holds if the FST can output subsequence S when reading input T.

Finite state transducers. More formally, we consider a restricted form of FSTs defined as follows. An FST $\mathcal{A}$ is a 5-tuple $(Q, q_S, Q_F, \Sigma, \Delta)$, where Q is a set of states, $q_S \in Q$ is the initial state, $Q_F \subseteq Q$ is the set of final states, Σ is an input and output alphabet, and $\Delta \subseteq Q \times (\Sigma \cup \{\, \epsilon \,\}) \times (\Sigma \cup \{\, \epsilon \,\}) \times Q$ is a transition relation. For every transition $(q_{from}, in, out, q_{to}) \in \Delta$, we require that $out \in \mathrm{anc}(in) \cup \{\, \epsilon \,\}$ and that whenever $in = \epsilon$ then $out = \epsilon$. Our notion of FSTs differs from traditional FSTs in that we use a common input and output alphabet and in that we restrict output labels. The latter restriction ensures that our FSTs output generalized subsequences of their input (Lemma 1). Fig. 2(a) shows an example FST, where $q_S = q_0$, $Q_F = \{\, q_{11} \,\}$, and each transition is marked with in:out labels. We refer to transitions with $in = \epsilon$ (and thus $out = \epsilon$) as ϵ-*transitions*; these transitions are marked with ϵ in the figure.

Runs and outputs. Let $T = t_1 t_2 \dots t_n$ be an input sequence. A *run* for T is a sequence $p = p_1 p_2 \dots p_m$ of transitions, where for $1 \leq i \leq m$: $p_i = (q_i, w_i, w'_i, q'_i) \in \Delta$, $q_1 = q_S$, $q_{i+1} = q'_i$, and $w_1 w_2 \dots w_m = T$ (recall that $w_i \in \Sigma \cup \{\, \epsilon \,\}$ so that $m \geq n$). Intuitively, the FST starts in state q_S and repeatedly selects transitions that are consistent with the next input item. If $q_m \in Q_F$, we refer to p as an *accepting run*. The output $O(p)$ of run p is the sequence $S = w'_1 \dots w'_m$ of output labels, where we omit all w'_i with $w'_i = \epsilon$ and set $S = \epsilon$ if all $w'_i = \epsilon$. The set of sequences generated by FST $\mathcal{A}$ is given by

$$G_{\mathcal{A}}(T) = \{\, O(p) \neq \epsilon \mid p \text{ is an accepting run of } \mathcal{A} \text{ for } T \,\}.$$

EXAMPLE 2. *Consider the FST $\mathcal{A}_{F2(a)}$ of Fig. 2(a). $\mathcal{A}_{F2(a)}$ has two accepting runs for sequence $T_1 = ca_1b_{12}e$, which are given by $p_1 = q_0 \xrightarrow{\epsilon} q_1 \xrightarrow{c:\epsilon} q_3 \xrightarrow{\epsilon} q_5 \xrightarrow{\epsilon} q_6 \xrightarrow{a_1:a_1} q_8 \xrightarrow{\epsilon} q_{10} \xrightarrow{\epsilon} q_5 \xrightarrow{\epsilon} q_7 \xrightarrow{b_{12}:B} q_9 \xrightarrow{\epsilon} q_{10} \xrightarrow{e:\epsilon} q_{11}$ with output $O(p_1) = a_1B$, and p_2 (as p_1 but using $q_6 \xrightarrow{a_1:A} q_8$) with output $O(p_2) = AB$. Thus, $G_{\mathcal{A}_{F2(a)}}(T_1) = \{\, a_1B, AB \,\}$, as desired. There is no accepting run for T_2 so that $G_{\mathcal{A}_{F2(a)}}(T_2) = \emptyset$. Observe that $\mathcal{A}_{F2(a)}$ generates precisely the P-sequences of Ex. 1.*

The following lemma states that our FSTs generate generalized subsequences of their inputs and thus specify subsequence predicates. Note that the lemma holds for any run, whether or not accepting.

LEMMA 1. *Let $T \in \Sigma^*$ be an input sequence and $\mathcal{A}$ be an FST. For any run p of $\mathcal{A}$ for T, it holds $O(p) \sqsubseteq T$.*

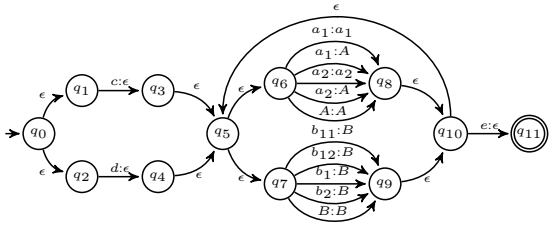
Note that not all subsequence predicates can be expressed with FSTs; e.g., there is no FST for predicate "all subsequences of form a^*b^* with an equal number of a's and b's". FST are a good trade-off between expressiveness and computational complexity, however: they can express many subsequence constraints that occur in practice and they lend themselves to efficient mining (see Sec. V).

Translating pattern expression. We now describe how to translate a pattern expression E into an FST $\mathcal{A}(E)$. The FST formally defines the semantics of pattern expressions: we set $G_E(T) \stackrel{\text{def}}{=} G_{\mathcal{A}(E)}(T)$. Each item expression is translated into a two-state FST with $Q = \{\, q_S, q_F \,\}$, where q_S is the initial and q_F the final state. The transitions of the FST depend on the item expression and are summarized in Tab. II, column "FST". The translation rules for composite expressions mirror the Thompson construction [20] for translating regular expressions to finite state automata. For example, expression E_{ex} translates to the FST of Fig. 2(a).

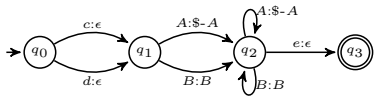
Compressed FST. The translation rules above can produce very large FSTs, especially when the vocabulary is large. For example, if the hierarchy has n items and average depth d, the

TABLE II: Translation rules for item expressions (where $w, w', w'' \in \Sigma$)

Expr.	Matches	Transl. type	Produces	FST	Compressed FST
$w_=$	w	Uncaptured	ϵ	$\{q_S \xrightarrow{w:\epsilon} q_F\}$	$\{q_S \xrightarrow{w_=:\epsilon} q_F\}$
		Captured	w	$\{q_S \xrightarrow{w:w} q_F\}$	$\{q_S \xrightarrow{w_=:w} q_F\}$
w	$w' \in \text{desc}(w)$	Uncaptured	ϵ	$\{q_S \xrightarrow{w':\epsilon} q_F \mid w' \in \text{desc}(w)\}$	$\{q_S \xrightarrow{w:\epsilon} q_F\}$
		Captured	w'	$\{q_S \xrightarrow{w':w'} q_F \mid w' \in \text{desc}(w)\}$	$\{q_S \xrightarrow{w:\$} q_F\}$
.	$w \in \Sigma$	Uncaptured	ϵ	$\{q_S \xrightarrow{w:\epsilon} q_F \mid w \in \Sigma\}$	$\{q_S \xrightarrow{.:\epsilon} q_F\}$
		Captured	w	$\{q_S \xrightarrow{w:w} q_F \mid w \in \Sigma\}$	$\{q_S \xrightarrow{.:\$} q_F\}$
$w^\uparrow$	$w' \in \text{desc}(w)$	Captured	$\text{anc}(w') \cap \text{desc}(w)$	$\{q_S \xrightarrow{w':w''} q_F \mid w' \in \text{desc}(w), w'' \in \text{anc}(w') \cap \text{desc}(w)\}$	$\{q_S \xrightarrow{w:\$\text{-}w} q_F\}$
$.^\uparrow$	$w \in \Sigma$	Captured	$\text{anc}(w)$	$\{q_S \xrightarrow{w:w'} q_F \mid w \in \Sigma, w' \in \text{anc}(w)\}$	$\{q_S \xrightarrow{.:\$\text{-}\top} q_F\}$
$w_=^\uparrow$	$w' \in \text{desc}(w)$	Captured	w	$\{q_S \xrightarrow{w':w} q_F \mid w' \in \text{desc}(w)\}$	$\{q_S \xrightarrow{w:w} q_F\}$



(a) FST



(b) Compressed FST

Fig. 2: FST (a) and cFST (b) for $[c|d]([A^\uparrow \mid B_=^\uparrow]^+)e$.

FST for "$.^\uparrow$" has $\Theta(nd)$ transitions. To avoid this explosion of FST size and support efficient mining, we make use of a compressed FST (cFST) representation for this purpose; see column "compressed FST" of Tab. II. The cFST of an item expression has exactly one transition, but input and output labels are taken from an alphabet larger than Σ. Each transition in the cFST describes a set of transitions in the corresponding FST in a concise way. More specifically, cFSTs use as input labels ., w, and $w_=$ for all $w \in \Sigma$. Here "." matches all input items, w matches all items in $\text{desc}(w)$, and $w_=$ matches only item w. cFSTs use as output labels ϵ, w, \$, \$-w, and \$-$\top$ for $w \in \Sigma$. Each transition encodes the set of output labels in the corresponding FST: ϵ and w are as before, \$ encodes the matched input item, \$-w the matched input item and all its ancestors up to w, and \$-$\top$ the matched item and all its ancestors. The cFST translations for composite expressions remain unmodified. Fig. 2(b) shows a cFST $\mathcal{A}_{ex}$ for E_{ex}. Note that the cFST has fewer transitions than its uncompressed counterpart of Fig. 2(a).

Simulating cFSTs. To simulate a cFST, we start with the initial state q_S and repeatedly select a transition in which the input label matches (see column "Matches" in Tab. II) the next input item. If there are multiple such transitions, we try them one by one via backtracking. As we move from state to state, we keep track of the outputs in a buffer (column "Produces" in Tab. II). If we reach a final state after consuming all input items, we add the buffered output to the set $G_\mathcal{A}(T)$. See [21] for more details and discussion.

V. Pattern Mining

We now outline three methods for mining P-frequent sequences: Naïve, DESQ-COUNT, and DESQ-DFS; more information can be found in [21]. We assume that subsequence predicate P is described by a cFST $\mathcal{A}$.

Naïve approach. The naïve "generate-and-count" approach is to compute $G_\mathcal{A}(T)$ for each input sequence $T \in \mathscr{D}$ via cFST simulation and count how often each sequence has been generated. The naïve approach is generally inefficient because it considers many globally infrequent sequences. For example, we obtain $G_{A_{ex}}(T_3) = \{AAAB, AAa_2B, Aa_1AB, Aa_1a_2B, a_2AAB, a_2Aa_2B, a_2a_1AB, a_2a_1a_2B\}$ for input sequence T_3, but only $AAAB$ and Aa_1AB are P-frequent.

DESQ-COUNT. DESQ-COUNT reduces the number of sequences that are generated and counted by making use of an *f-list* F, which contains all items along with their frequencies and can be precomputed. For our example database, we obtain f-list $F_{ex} = \{A{:}6, e{:}6, B{:}6, a_1{:}6, d{:}3, b_2{:}3, b_1{:}2, c{:}2, b_{12}{:}1, b_{11}{:}1, a_2{:}1\}$. In DESQ-COUNT, we make use of the f-list to reduce the size of $G_\mathcal{A}(T)$ by generating sequences composed of frequent items only. For example, for T_3, we have $G_{\mathcal{A}_{ex}}(T_3) = \{AAAB, Aa_1AB\}$, which is much smaller than the full set given above. We compute the reduced $G_\mathcal{A}(T)$ by adapting cFST simulation to work with the f-list. To do so, we stop exploring a run as soon as an infrequent item is produced.

The pruning performed by DESQ-COUNT can substantially reduce the number of candidate sequences. DESQ-COUNT is inefficient (and sometimes infeasible), however, if pruning is not sufficiently effective and the sets $G_\mathcal{A}(T)$ are very large. The DESQ-DFS algorithm, which we present next, addresses such cases.

DESQ-DFS. In DESQ-DFS, we adapt the pattern-growth framework of PrefixSpan [11] to FSTs. Pattern growth approaches arrange the output sequences in a tree, in which each node corresponds to a sequence S and associated with a *projected database*, which stores the set of input sequences in which S occurs. Starting with an empty sequence database and the full sequence database, the tree is build recursively by performing a series of *expansions*. In each expansion, a frequent sequence S (l items) is expanded to generate sequences with prefix S (of $l+1$ items), their projected databases as well as their supports.

We adapt this pattern-growth approach to efficiently generate P-frequent sequences as follows. For a sequence S, we store in its projected database the state of simulations of $\mathcal{A}$ on all input sequences that generate S as a partial output. We refer to a state of simulation as a *snapshot*, which is a triple $T[pos@q]$ where T is the input sequence, pos is the position of the next input item, and q is the current state in $\mathcal{A}$. Thus, our projected database for a sequence S contains all snapshots that generate S as a partial output. When expanding a sequence S, for all snapshots in its projected database, we resume the simulation of the FST for T at item t_{pos} in state q until an output item is produced or the entire input is consumed. In the former case, we add a new snapshot to respective child node.

In the latter case, we add T to the support of S if we end up in a final state. During each expansion, we also keep track of number of input sequences that can generate S, which allows us to prune partial sequences as soon as it becomes clear that they cannot be expanded to a P-frequent sequences.

VI. Experimental Evaluation

We conducted an experimental study on two publicly available real-world datasets. Our goal was to investigate whether pattern expressions are sufficiently powerful to express prior and new subsequence constraints, whether DESQ's algorithms are efficient, and how they perform relative to each other and to prior algorithms. We summarize the key results of our experimental study here. Additional experiments and a more in-depth discussion can be found in [21].

Datasets. Our first dataset, NYT, contains over 21M sentences from articles published in the the New York Times corpus [22]. We generated an item hierarchy using annotations from the Stanford CoreNLP tools. The NYT hierarchy consists of named entities, which generalize to their type (PERSON, ORGANIZATION, LOCATION, MISC) and then to ENTITY, and of words, which generalize to their lemma and then to their part-of-speech tag. Our second dataset, PRT, is a dataset of over 100K protein sequences obtained from [23] and is composed of 25 amino acid codes (items). The hierarchy is flat, i.e., there are no generalizations.

Implementation and setup. We implemented DESQ in Java (JDK 1.8; http://dws.informatik.uni-mannheim.de/en/resources/software/desq/). For length and gap constraints, we additionally used (1) C++ implementation of cSPADE [13] from the authors, (2) our implementation of SPADE in Java which adds hierarchy constraints, (3) our implementation of prefix-growth [12] in Java. For RE constraints, we used prefix-growth and (4) a C++ executable of SMA [18] obtained from the authors.

Experiments on the NYT dataset were performed on a machine with two Intel(R) Xeon(R) CPU E5-2640 v2 processors and 128GB of RAM running CentOS Linux 7.1. Experiments on the PRT dataset were performed on a machine equipped with Intel Core i7-4712HQ and 16GB RAM running Windows 10. We used a different setup for the PRT dataset as the SMA implementation is provided as a Windows binary only.

Traditional constraints. We first investigated the overhead of DESQ compared to specialized miners for prior subsequence constraints. In particular we considered length and gap constraints as well as item hierarchies (T_1–T_3 of Tab. III). We used the NYT dataset; the results are shown in Fig. 3(a) using log-scale. We observed that DESQ-DFS was up to two orders of magnitude faster than cSPADE and had negligible overhead (less than 2.5%) compared to prefix-growth. For all other experiments, DESQ-DFS was competitive and had an overhead of up to 13%.

RE constraints. In this set of experiments, we evaluated the efficiency of DESQ for mining frequent subsequences (all or contiguous) that match a RE (P_1–P_4 in Tab. III, PRT dataset, constraints from [24]). We compared DESQ's performance against state-of-the-art RE-constraint FSM methods SMA and prefix-growth. The results are shown in log-scale in Fig. 3(b). We observed that DESQ was up to 2.5x slower than SMA for P_1 and up to 1.3x slower than SMA on P_2. We do not give SMA results for P_3 and P_4 because the implementation produced incorrect results (acknowledged by the original authors). We did not investigate this further as the SMA source code is not available. DESQ was roughly on par with prefix-growth for P_1–P_4 (up to 1.3x) slower.

Customized constraints. We considered pattern expressions that express constraints in information extraction (IE) and natural language processing (NLP) applications (N_1–N_5 in Tab. III, constraints inspired from [4], [5], [25], [26]). We evaluated the performance of Naïve, DESQ-COUNT and DESQ-DFS. The runtime results are shown in Fig. 3(c) in log-scale. For expressions N_1–N_3, DESQ-COUNT and DESQ-DFS had similar performance. For N_4–N_5, however, runtimes were higher and DESQ-DFS was significantly faster than DESQ-COUNT (up to 14x). To gain insight into these results, we computed the average number μ of P-sequences per input sequence (shown above each bar). For small values of μ, DESQ-COUNT and DESQ-DFS had similar performance, whereas for larger values of μ, DESQ-DFS was much more efficient. When μ is small, the simple counting method of DESQ-COUNT is expected to work well because few sequences are generated. When μ is large, however, DESQ-COUNT enumerates many sequences that turn out to be infrequent, which is expensive. Many of these sequences are pruned early by DESQ-DFS.

Summary. (1) Many subsequence constraints can be expressed with pattern expressions. (2) DESQ has acceptable overhead over state-of-the-art specialized sequence miners for common subsequence constraints. (3) DESQ-COUNT was consistently faster than Naïve. (4) DESQ-COUNT and DESQ-DFS had similar performance in cases where the average number of P-subsequences per input sequence was small. (5) When many subsequences per input were generated, DESQ-DFS was more than an order of magnitude faster than DESQ-COUNT and Naïve. (6) cFSTs sped up pattern matching by multiple orders of magnitude when compared to the state-of-the-art FST library OpenFST (see [21]). Our results indicate that DESQ is a suitable general-purpose system for a wide range of subsequence constraints.

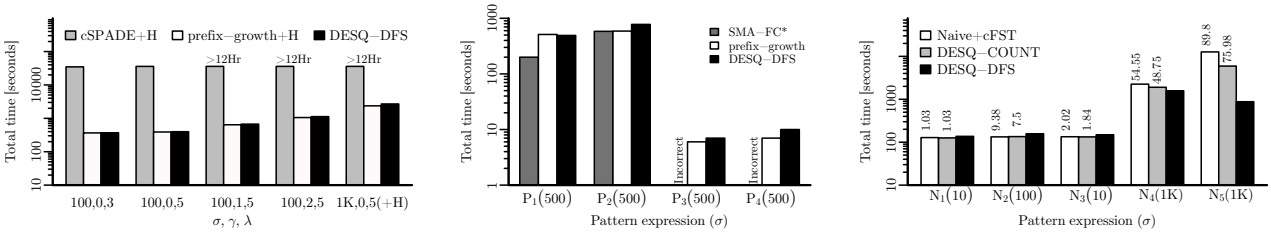
VII. Related Work

Subsequence constraints. Prior work on FSM has mostly focused on specific notions of subsequence constraints. GSP [1] and LASH [8], for example, allow gap constraints and incorporate item hierarchies. cSPADE [13] handles length, gap and item constraints. Wu et al. [27] consider subsequences with periodic wild card gaps, i.e., subsequences where consecutive items are separated by the same gap in the input. RE constraints have been studied by [9], [12], [17], [18]; these methods do not support capture groups. Some of the above constraints (e.g., gap constraints) target the input sequence, whereas others (e.g., length constraints, RE constraints) target subsequences. Pattern expressions unify both targets and can express all of the above subsequence constraints (e.g., see Tab. I) as well as customized subsequence constraints that arise in FSM applications (e.g., see Tab. III).

Pattern matching. Our work is also related to to pattern matching. There are many languages and systems for pattern matching over sequences; e.g., for information extraction [28], [29], computational linguistics [30], complex event processing [31], and sequence databases [32], [33]. Our pattern expressions are simpler than most pattern matching languages, yet expressive enough to specify many subsequence constraints that arise in applications. The existing pattern matching languages can conceivably be used to specify subsequence predicates and mine P-frequent sequences using Naïve, i.e., by first enumerating all matches and subsequently counting frequencies. Our experiments indicate that this approach is infeasible for many subsequence constraints. Instead, it is beneficial to integrate pattern matching and mining, e.g., along the lines of DESQ-COUNT and DESQ-DFS.

TABLE III: Pattern expr. for traditional FSM (T_1–T_3), RE-constrained FSM (P_1–P_4), and IE and NLP applications (N_1–N_5)

Pattern expression	Description	Example patterns from NYT dataset (frequency)
T_1: (.){1,λ}	*n-grams of up to λ words*	green tea (337), editor in chief (3275)
T_2: (.)[.{0,γ}(.)]{1,λ − 1}	*Skip n-grams with gap at most γ words and of up to length λ*	flight from to (758), son of and of (15896)
T_3: (.↑){1,λ}	*Generalized n-grams of up to λ words*	NOUN PREP DET NOUN (4.2M), PERSON be NOUN (2199)
		Example patterns from PRT dataset (frequency)
P_1: ([S|T]).*(.).*([R|K])	*subsequences that match RE≡[S|T].[R|K]*	SLR (103,093), TAK (102,941), SAK (102,946)
P_2: ([I|V]).*(D).*(L).*(G).*(T).*([S|T]).*(.).*([S|C])	*subsequences that match RE≡[I|V]DLGT[S|T].[S|C]*	IDLGTTLS (102,975), VDLGTSTC (92,662) VDLGTSDS (102,901)
P_3: ([S|T].[R|K])	*contiguous subsequences that match RE≡[S|T].[R|K]*	SLR (14,995), TAK (8,840), SAK (10,397)
P_4: ([S|T]..[D|E])	*contiguous subsequences that match RE≡[S|T]..[D|E]*	SDLE (2,015), TLEE (2,329), SGLD (1,054)
		Example patterns from NYT dataset (frequency)
N_1: ENTITY (VERB+ NOUN+? PREP?) ENTITY	*Relational phrase between entities*	lives in (847), is being advised by (15), has coached (10)
N_2: (ENTITY↑ VERB+ NOUN+? PREP? ENTITY↑)	*Typed relational phrases*	ORG headed by ENTITY (275), PER born in LOC (481)
N_3: (ENTITY↑ be$\overset{\uparrow}{=}$) DET? (ADV? ADJ? NOUN)	*Copular relation for an entity*	PER be novelist (165), LOC be great place (38),
N_4: (.↑){3} NOUN	*Generalized 3-grams before a noun*	NOUN PREP DET (4,223,219), DET ADV ADJ (350,005)
N_5: ([.↑ . .]|[. .↑ .]|[. . .↑])	*Generalized 3-grams, where at most one item is generalized*	the ADJ human (1,238), for DET book (1,704)



(a) NYT + trad. constraints (min. sup., length, gap, hierarchy?) (b) PRT + RE-constraints (min. support) (c) NYT + customized constraints (min. support)

Fig. 3: Wall-lock times of various mining tasks and mining algorithms

VIII. Conclusions

In this paper, we introduced subsequence predicates as a general model for unifying and extending subsequence constraints for FSM. We proposed pattern expressions as a simple, intuitive way to express subsequence constraints, suggested compressed finite state transducers as an underlying computation model, and proposed the DESQ-COUNT and DESQ-DFS algorithms for efficient mining. Our experiments indicate that DESQ is an efficient, general-purpose FSM framework for common as well as customized subsequence constraints.

References

[1] R. Srikant and R. Agrawal, "Mining sequential patterns: Generalizations and performance improvements," in *EBDT*, 1996, pp. 3–17.

[2] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan, "Web usage mining: Discovery and applications of usage patterns from web data," *SIGKDD Explor. Newsl.*, vol. 1, no. 2, pp. 12–23, 2000.

[3] A. Lopez, "Statistical machine translation," *ACM Comput. Surv.*, vol. 40, no. 3, pp. 8:1–8:49, 2008.

[4] A. Fader, S. Soderland, and O. Etzioni, "Identifying relations for open information extraction," in *EMNLP*, 2011, pp. 1535–1545.

[5] N. Nakashole, G. Weikum, and F. Suchanek, "Patty: A taxonomy of relational patterns with semantic types," in *EMNLP-CoNLL*, 2012, pp. 1135–1145.

[6] A. Brazma, I. Jonassen, J. Vilo, and E. Ukkonen, "Pattern discovery in biosequences," *LNCS*, vol. 1433, pp. 257–270, 1998.

[7] K. Beedkar, K. Berberich, R. Gemulla, and I. Miliaraki, "Closing the gap: Sequence mining at scale," *ACM Trans. Database Syst.*, vol. 40, no. 2, pp. 8:1–8:44, 2015.

[8] K. Beedkar and R. Gemulla, "Lash: Large-scale sequence mining with hierarchies," in *SIGMOD*, 2015, pp. 491–503.

[9] M. N. Garofalakis, R. Rastogi, and K. Shim, "Spirit: Sequential pattern mining with regular expression constraints," in *VLDB*, 1999, pp. 223–234.

[10] I. Miliaraki, K. Berberich, R. Gemulla, and S. Zoupanos, "Mind the gap: Large-scale frequent sequence mining," in *SIGMOD*, 2013, pp. 797–808.

[11] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu, "PrefixSpan: Mining sequential patterns by prefix-projected growth," in *ICDE*, 2001, pp. 215–224.

[12] J. Pei, J. Han, and W. Wang, "Mining sequential patterns with constraints in large databases," in *CIKM*, 2002, pp. 18–25.

[13] M. J. Zaki, "Sequence mining in categorical domains: Incorporating constraints," in *CIKM*, 2000, pp. 422–429.

[14] K. Berberich and S. Bedathur, "Computing n-gram statistics in mapreduce," in *EDBT*, 2013, pp. 101–112.

[15] M. J. Zaki, "Spade: An efficient algorithm for mining frequent sequences," *Mach. Learn.*, vol. 42, no. 1-2, pp. 31–60, 2001.

[16] H. Mannila, H. Toivonen, and A. I. Verkamo, "Discovery of frequent episodes in event sequences," *Data Mining and Knowledge Discovery*, vol. 1, no. 3, pp. 259–289, 1997.

[17] H. Albert-Lorincz and J.-F. Boulicaut, "Mining frequent sequential patterns under regular expressions: a highly adaptative strategy for pushing constraints," in *SDM*, 2003, pp. 316–320.

[18] R. Trasarti, F. Bonchi, and B. Goethals, "Sequence mining automata: A new technique for mining frequent sequences under regular expressions," in *ICDM*, 2008, pp. 1061–1066.

[19] M. Mohri, "Finite-state transducers in language and speech processing," *Comput. Linguist.*, vol. 23, no. 2, pp. 269–311, Jun. 1997.

[20] K. Thompson, "Programming techniques: Regular expression search algorithm," *Commun. ACM*, vol. 11, no. 6, pp. 419–422, 1968.

[21] K. Beedkar and R. Gemulla, "DESQ: Frequent sequence mining with subsequence constraints," https://arxiv.org/abs/1609.08431, Tech. Rep., 2016.

[22] The New York Times corpus, http://catalog.ldc.upenn.edu/LDC2008T19.

[23] SMA, http://www-kdd.isti.cnr.it/SMA/.

[24] PROSITE, http://prosite.expasy.org/.

[25] Google n-Grams, https://books.google.com/ngrams/.

[26] L. Del Corro, A. Abujabal, R. Gemulla, and G. Weikum, "Finet: Context-aware fine-grained named entity typing," in *EMNLP*, 2015, pp. 868–878.

[27] Y. Wu, L. Wang, J. Ren, W. Ding, and X. Wu, "Mining sequential patterns with periodic wildcard gaps," *Applied intelligence*, vol. 41, no. 1, pp. 99–116, 2014.

[28] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu, "Systemt: A system for declarative information extraction," *SIGMOD Rec.*, vol. 37, no. 4, pp. 7–13, 2009.

[29] D. E. Appelt and B. Onyshkevych, "The common pattern specification language," in *TIPSTER*, 1998, pp. 23–30.

[30] O. Christ, "A modular and flexible architecture for an integrated corpus query system," *CoRR*, vol. abs/cmp-lg/9408005, 1994.

[31] N. Dindar, B. Güç, P. Lau, A. Ozal, M. Soner, and N. Tatbul, "Dejavu: Declarative pattern matching over live and archived streams of events," in *SIGMOD*, 2009, pp. 1023–1026.

[32] P. Seshadri, M. Livny, and R. Ramakrishnan, "The design and implementation of a sequence database system," in *VLDB*, 1996, pp. 99–110.

[33] E. Lo, B. Kao, W.-S. Ho, S. D. Lee, C. K. Chui, and D. W. Cheung, "Olap on sequence data," in *SIGMOD*, 2008, pp. 649–660.