A Unified Framework for Frequent Sequence Mining with Subsequence Constraints

KAUSTUBH BEEDKAR, Technische Universität Berlin
RAINER GEMULLA, Universität Mannheim
WIM MARTENS, Universität Bayreuth

Frequent sequence mining methods often make use of constraints to control which subsequences should be mined. A variety of such subsequence constraints has been studied in the literature, including length, gap, span, regular-expression, and hierarchy constraints. In this article, we show that many subsequence constraints—including and beyond those considered in the literature—can be unified in a single framework. 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. 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. Our algorithms translate pattern expressions to succinct finite-state transducers, which we use as computational model, and simulate these transducers in a way suitable for frequent sequence mining. Our experimental study on real-world datasets indicates that our algorithms—although more general—are efficient and, when used for sequence mining with prior constraints studied in literature, competitive to (and in some cases superior to) state-of-the-art specialized methods.

CCS Concepts: • **Information systems → Data mining**;

Additional Key Words and Phrases: Data mining, frequent sequence mining, sequential pattern mining, subsequence constraints, hierarchies, finite-state transducers

ACM Reference format:
Kaustubh Beedkar, Rainer Gemulla, and Wim Martens. 2019. A Unified Framework for Frequent Sequence Mining with Subsequence Constraints. *ACM Trans. Database Syst.* 44, 3, Article 11 (May 2019), 42 pages.
https://doi.org/10.1145/3321486

1 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 [43], web usage mining and session analysis [44], natural language processing [29], information extraction [21, 37], or computational biology [13]. 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

Authors' addresses: K. Beedkar, Technische Universität Berlin, Berlin, Germany; email: kaustubh.beedkar@tu-berlin.de; R. Gemulla, Universität Mannheim, Mannheim, Germany; email: rgemulla@uni-mannheim.de; W. Martens, Universität Bayreuth, Bayreuth, Germany; email: wim.martens@uni-bayreuth.de.

0362-5915/2019/05-ART11 $15.00
https://doi.org/10.1145/3321486

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 [21, 37].

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 [9, 10, 23, 33, 39, 40, 43, 50]. 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 [12], 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 that 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 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.[1] We show that many subsequence constraints—including and beyond the constraints mentioned above—can be unified in a single framework. A unified framework offers advantages to both researchers and practitioners. In particular, it allows researchers to study algorithms and properties of subsequence constraints in general instead of focusing on certain special cases individually. It also helps to improve usability of pattern mining systems for practitioners: They only need to familiarize themselves with one framework and, perhaps more importantly, do not need to develop customized mining algorithms for a particular subsequence constraint of interest. In fact, we propose a number of general-purpose mining algorithms that operate within our framework. Our experimental study (Section 7) suggests that our methods are often competitive (and sometimes exponentially more efficient) to state-of-the-art specialized algorithms for the above-mentioned subsequence 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 [40, 47]—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 Table 1 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 Table 4.

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,[2] which includes two efficient mining algorithms termed DESQ-COUNT and DESQ-DFS. Both algorithms translate a given pattern expression to a *succinct*

[1] A preliminary version of this article appeared in Reference [11].

[2] https://www.uni-mannheim.de/dws/research/resources/desq/.

FST (sFST), which is 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 [39].

Both algorithms heavily rely on efficient sFST simulation. We discuss various optimizations for sFST simulation, which often improve mining performance substantially. First, we show how sFSTs can be partially determinized and minimized. Second, we discuss methods that allow us to early-abort sFST simulation whenever possible and without affecting correctness. Third, we propose a pruning method that enables us to quickly prune irrelevant input sequences, i.e., input sequences that cannot affect the mining results. Finally, we propose a two-pass approach to sFST simulation that additionally avoids unnecessary backtracking and show that the two-pass approach can be exponentially more efficient than the one-pass approach for certain pattern expressions.

We conducted an experimental study on multiple real-world datasets to investigate the expressiveness of our pattern expression language, the efficiency of our mining algorithms, and the effectiveness of our proposed optimizations. We found that our pattern expressions are sufficiently powerful to express many subsequence constraints that arise in sequence mining applications. Our algorithms were generally efficient, and when used for pattern expressions that express prior subsequence constraints, competitive to—and sometimes more efficient than—state-of-the-art specialized methods. Our sFST optimizations were effective and significantly improved performance of our mining algorithms. Our results suggests that DESQ is an efficient general-purpose FSM framework for wide range of sequence mining tasks.

The remainder of this article is organized as follows. In Section 2, we summarize basic concepts for FSM and establish the notation used throughout this work. In Section 3, we introduce subsequence predicates and formally define the problem of frequent sequence mining with general subsequence constraints. In Section 4, we propose our pattern expression language and finite-state transducers as the underlying computational model. Based on these transducers, we derive algorithms for frequent sequence mining in Section 5. In Section 6, we propose various optimizations for efficiently simulating finite-state transducers. Section 7 reports on our experimental study and its results. Section 8 discusses additional related work, and Section 9 concludes the article.

2 PRELIMINARIES

Sequence Databases. A *sequence database* is a set[3] 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 finite set $\Sigma = \{ w_1, w_2, \ldots, w_{|\Sigma|} \}$ that we call *vocabulary.*[4] We refer to T as a *sequence over* Σ. We denote by ε the empty sequence, by $|T|$ the length of sequence T, by Σ^* (resp., Σ^+) the set of all (respectively, all non-empty) sequences that can be constructed from items in Σ. Figure 1(a) shows an example sequence database $\mathscr{D}_{ex}$ consisting of six sequences over $\Sigma = \{A, a_1, a_2, B, b_1, b_2, b_{11}, b_{12}, c, d, e\}$.

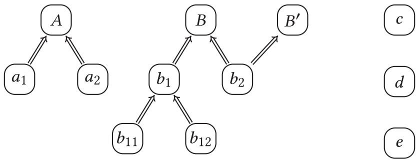
Item Hierarchy. The items in Σ are arranged in an *item hierarchy*, which expresses how items can be generalized (or that they cannot be generalized). Figure 1(b) shows an example hierarchy in which, for example, item a_1 generalizes to item A. In general, we say that an item *u directly generalizes to* an item v, denoted $u \Rightarrow v$, if u is a child of v in the hierarchy. We further denote by $\Rightarrow^*$ the reflexive transitive closure of $\Rightarrow$. For the example of Figure 1(b), we have $b_{11} \Rightarrow b_1$, $b_1 \Rightarrow B$,

[3] The restriction to sets is for expository reasons. In practice, sequence databases are more accurately abstracted as *multisets*, but we chose sets to make our definitions clearer. It is not difficult to generalize our approach from sets to multisets and, in fact, our implementation uses multisets.

[4] A more general variant of this setting is often considered in literature, in which sequences are formed of itemsets rather than individual items. In this article, we focus on the special case of sequences composed of individual items (e.g., textual data, user sessions, event logs, protein sequences, etc.)

$T_1 : ca_1b_{12}e$
$T_2 : a_1b_2e$
$T_3 : da_2a_1a_2b_{11}e$
$T_4 : da_1Be$
$T_5 : ea_1b_2d$
$T_6 : ca_1a_1a_1b_2e$

(a) Example database $\mathscr{D}_{ex}$



(b) Example hierarchy

Fig. 1. A sequence database and its vocabulary.

and $b_{11} \Rightarrow^* B$. For each item $w \in \Sigma$, we denote by

$$\text{anc}(w) = \{\, w' \mid w \Rightarrow^* w' \,\}$$

the set of *ancestors* of w (including w) and by

$$\text{desc}(w) = \{\, w' \mid w' \Rightarrow^* w \,\}$$

the set of *descendants* of w (again, including w). In our running example, we have $\text{anc}(b_1) = \{ b_1, B \}$ and $\text{desc}(b_1) = \{ b_1, b_{11}, b_{12} \}$.

Subsequences. Let $S = s_1s_2 \ldots s_{|S|}$ and $T = t_1t_2 \ldots t_{|T|}$ be two sequences over 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 $1 \leq i_1 < i_2 < \cdots < i_{|S|} \leq |T|$ 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 $a_1c \not\sqsubseteq T_1$.

3 FSM WITH SUBSEQUENCE CONSTRAINTS

The goal of FSM is to discover subsequences that occur in sufficiently many input sequences. This problem can be challenging, because the total number of distinct subsequences of one input sequence T can be exponential in the length of T. This poses two problems: (1) enumerating or mining frequent subsequences can be expensive and (2) many of the subsequences may not be useful to applications. To alleviate these problems, FSM methods have focused on specialized subsequence constraints to control which subsequences should be mined and developed specialized mining algorithms to improve efficiency. We will tackle a more general and unified problem, which we define in this section.

Subsequence Constraints. A *subsequence constraint* describes which subsequences of a given input sequence should be considered for frequent sequence mining. Commonly proposed subsequence constraints are summarized in Table 1 and include: *gap constraints* [33, 43], where items in the subsequences need to appear "close" in the input sequence; *length constraints* [50], where the number of items in the subsequences is bounded; *hierarchy constraints* [10], where items in the subsequences generalize according to the item hierarchy; and *regular expression (RE) constraints* [2, 23, 40, 47], where subsequences must match a given RE. In this article, our goal is to provide a general framework to express subsequence constraints, including and going beyond previously proposed constraints. Our extensions allow to mix-and-match constraints (e.g., gap-constrained subsequences that match an RE constraint) or to incorporate context constraints (e.g, frequent relational phrases between named entities [21]).

Consider the following (admittedly contrived) subsequence constraint. We use it as a running example to explain the features of our framework.

Example 3.1. Consider our example database $\mathscr{D}_{ex}$ of input sequences. Suppose that we are interested in doing the following:

(1) We only want to consider input sequences $t_1 t_2 \cdots t_n$, where (i) $n \geq 3$, (ii) t_1 is c or d, (iii) t_n is e, and (iv) every t_i with $1 < i < n$ can be generalized to A or B.
(2) From these input sequences, we want to extract the subsequence $t_2 \cdots t_{n-1}$.
(3) For each such extracted subsequence, we allow the following generalizations to obtain the subsequences we are interested in: descendants of A can be generalized arbitrarily (or not at all), whereas descendants of B must be generalized to B.

Notice that $a_1 B \sqsubseteq T_1$ and $AB \sqsubseteq T_1$ satisfy this subsequence constraint, whereas $a_1 b_{12} \sqsubseteq T_1$ and $a_1 b_1 \sqsubseteq T_1$ do not (because they do not generalize descendants of B to B). Furthermore, $a_1 B \sqsubseteq T_2$ and $AB \sqsubseteq T_2$ do not satisfy the context constraint (because T_2 does not start with c or d).

The subsequence constraint of Example 3.1 combines (i) a gap constraint (condition 2: subsequence is consecutive in input), (ii) hierarchy constraints (conditions 1.iv and 3), and (iii) a context constraint (conditions 1.ii and 1.iii: subsequence occurs between c or d, and e). Prior methods (cf. Table 1) cannot handle this constraint: none of the methods supports context constraints, and methods that do support hierarchies do not support controlled (or enforced) generalizations. Note that the particular context constraint of Example 3.1 can be implemented using suitable preprocessing; such an approach is not possible in general though.

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 (generalized) subsequence. Subsequence $S \sqsubseteq T$ satisfies the constraint when $P(S, T)$ holds. Notice that P involves both the subsequence S and the 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 the subsequence constraint of Example 3.1, then $G_{P_{\text{ex}}}(T_1) = \{a_1 B, AB\}$ and $G_{P_{\text{ex}}}(T_2) = \emptyset$.

Subsequence predicates can encode different application needs, including but not limited to the various subsequence constraints discussed before. Subsequence predicates can act as a filter on the set of all subsequences of T ("only A's and B's" in Example 3.1), but may also consider the context in which these subsequences occur ("between c or d and e" in Example 3.1) and whether or not gaps are allowed ("consecutively" in Example 3.1). For example, we can construct subsequence predicates for generating 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 a wide range of subsequence predicates in Section 4.

FSM and Subsequence Predicates. Let P be a subsequence predicate. The *P-support* $\text{Sup}_P(S, \mathscr{D})$ of sequence $S \in \Sigma^+$ in sequence database $\mathscr{D}$ is the set[5] of all sequences in $\mathscr{D}$ that P-generate S, i.e.,

$$\text{Sup}_P(S, \mathscr{D}) = \{\, T \in \mathscr{D} \mid S \in G_P(T) \,\}. \tag{1}$$

The *P-frequency* of S in $\mathscr{D}$ is given by

$$f_P(S, \mathscr{D}) = |\,\text{Sup}_P(S, \mathscr{D})|,$$

that is, the number of sequences T in $\mathscr{D}$ for which $S \sqsubseteq T$ and $P(S, T)$ holds. In our example database, we have $\text{Sup}_{P_{\text{ex}}}(Aa_1 AB, \mathscr{D}_{ex}) = \{\, T_3, T_6 \,\}$ and thus $f_{P_{\text{ex}}}(Aa_1 AB, \mathscr{D}_{ex}) = 2$. Given a *support threshold* $\sigma > 0$, we say that a sequence S is *P-frequent* if $f_P(S, \mathscr{D}) \geq \sigma$.

[5] Recall our running assumption that $\mathscr{D}$ is a set.

Problem Statement 1. *Given a sequence database $\mathscr{D}$, a subsequence predicate P, and a support threshold $\sigma > 0$, find all P-frequent sequences $S \in \Sigma^+$ along with their P-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.

Discussion. The above definitions are generalizations of the notions of frequency and support used in traditional frequent sequence mining [1, 39, 51]. Efficient mining of P-frequent sequences is challenging, because the antimonotonicity property does not hold directly: We cannot generally deduce from the knowledge that sequence S is P-frequent whether or not any of the subsequences of S are P-frequent as well. For instance, in our running example, $AAAB$ is frequent but AA is not. One reason here is the context constraint: There are no sequences in the input database that satisfy condition (1) from Example 3.1 but only have descendants of A between the first and last symbol. Nevertheless, our mining algorithms make use of suitable adapted notions of antimonotonicity for subsequence predicates (Lemma 5.1) and pattern expressions (Lemma 5.2).

4 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 allow us to unify prior subsequence constraints (see Table 1) and to express constraints that cannot be handled by prior methods (see Table 4). We subsequently suggest a computational model based on finite-state transducers (FSTs), and describe the formal semantics of our language.

4.1 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, then 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, then so are $[E_1E_2]$ and $[E_1|E_2]$.

Pattern expressions are based on regular expressions but additionally include capture groups (in parentheses[6]), 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 Section 4.3 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

$$E_{ex} = [c|d]([A^{\uparrow} \mid B^{\uparrow}_{=}]^+)e \tag{2}$$

[6] We use round parentheses to denote capture groups, because this is the standard syntax in regex engines.

describes precisely the subsequence constraint of Example 3.1. 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, then it can "produce" an output item; see Table 2 for an overview. Fix some $w \in \Sigma$. The most basic item expression is $w_=$: it matches only item w and produces either ε (if uncaptured) or w (if captured). Using our example hierarchy of Figure 1(b), 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 $w' \in \operatorname{desc}(w)$ (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 a *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 $w' \in \operatorname{desc}(w)$ — as does expression w —, 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.

4.2 Comparison to Regular Expression Constraints

The use of pattern expressions allows DESQ to express many prior subsequence constraints in a unified way; example pattern expressions for common constraints are shown in Table 1. Pattern expressions are based on regular expressions but—in contrast to prior work on RE constraints (e.g., [5, 23, 40, 47])—target *input* sequences instead of output sequences and support hierarchies natively.

In more detail, the use of capture groups establishes a single formalism for expressing constraints with respect to the input (via uncaptured subexpressions; e.g., consecutive subsequences as in n-grams or non-consecutive subsequences with bounded gap) and with respect to the output (via captured subexpressions; e.g., subsequences with bounded length or regular expression constraints). For example, pattern expressions use uncaptured wildcards to express gap constraints (or the absence thereof); e.g., the pattern expressions for regular expression constraints with and without gaps at the bottom of Table 1 differ only in the use of uncaptured wildcards.

In combination with the use of hierarchies and generalizations, pattern expressions support many customized subsequence constraints that arise in applications succinctly, including constraints that cannot be expressed in prior FSM frameworks. Consider, for example, the task of mining frequent relational phrases between entities from large text corpora as in Reference [21]; e.g., the phrase *make a deal with* may be frequent between persons and/or organizations. An FSM

Table 1. Pattern Expressions for Prior Subsequence Constraints

Subsequence constraint	Examples	Pattern expression
All subsequences [5, 6, 39, 43, 51]		$[.^*(.)]^+$
Bounded length [5, 6, 50]	length 3–5	$[.^*(.)]\{3,5\}$
n-grams [12, 33]	3-, 4-, and 5-grams	$(.)\{3,5\}$
Bounded gap [5, 6, 33, 50]	each gap at most 3	$(.)[.\{0,3\}(.)]^+$
Serial episodes [32]	length 3, total gap ≤ 2	$(.)[.?.?(.) \mid .?(.).? \mid (.).?.?](.)$
Hierarchy [10, 43]	generalized 5-grams	$(.^{\uparrow})\{5\}$
Regular expression [2, 5, 6, 23, 40, 47]	subsequences matching $[a \vert b]\, c^*d$ contiguous subsequences matching $[a \vert b]\, c^*d$	$(a \vert b)[.^*(c)]^*.^*(d)$, $([a \vert b]\, c^*d)$

To increase readability, we omit a leading and trailing "$.^*$" from each pattern expression.

algorithm that does not support flexible constraints cannot solve such a task: it cannot be tailored to consider only relational phrases, thereby producing many uninteresting (i.e., non-relational) patterns, and it does not support context constraints, thereby producing spurious patterns (i.e., patterns that do not connect entities). In contrast, this constraint can be expressed succinctly via the pattern expression ENTITY (VERB$^+$ DET? NOUN$^+$? PREP?) ENTITY (similar to expression N_1 of Table 4). Here the expression inside the capture group describes relational phrases and the uncaptured part describes the context in which the phrase must occur (i.e., between two entities). Table 4 lists other examples, in which pattern expressions concisely describe customized sequence mining tasks in context of information extraction and natural language processing. For example, expression N_2 describes *semantically typed* relational phrases as in Reference [37], expression N_3 describes copular phrases as in Reference [48], expressions N_4 and N_5 are based on the subsequence constraints used to construct the well-known Google *n*-gram corpus [28]. Table 4 also includes pattern expressions (e.g., A_1–A_4) that describe customer behavior mining tasks and mining of protein sequences (e.g., P_1–P_4) that exhibit a given motif [47].

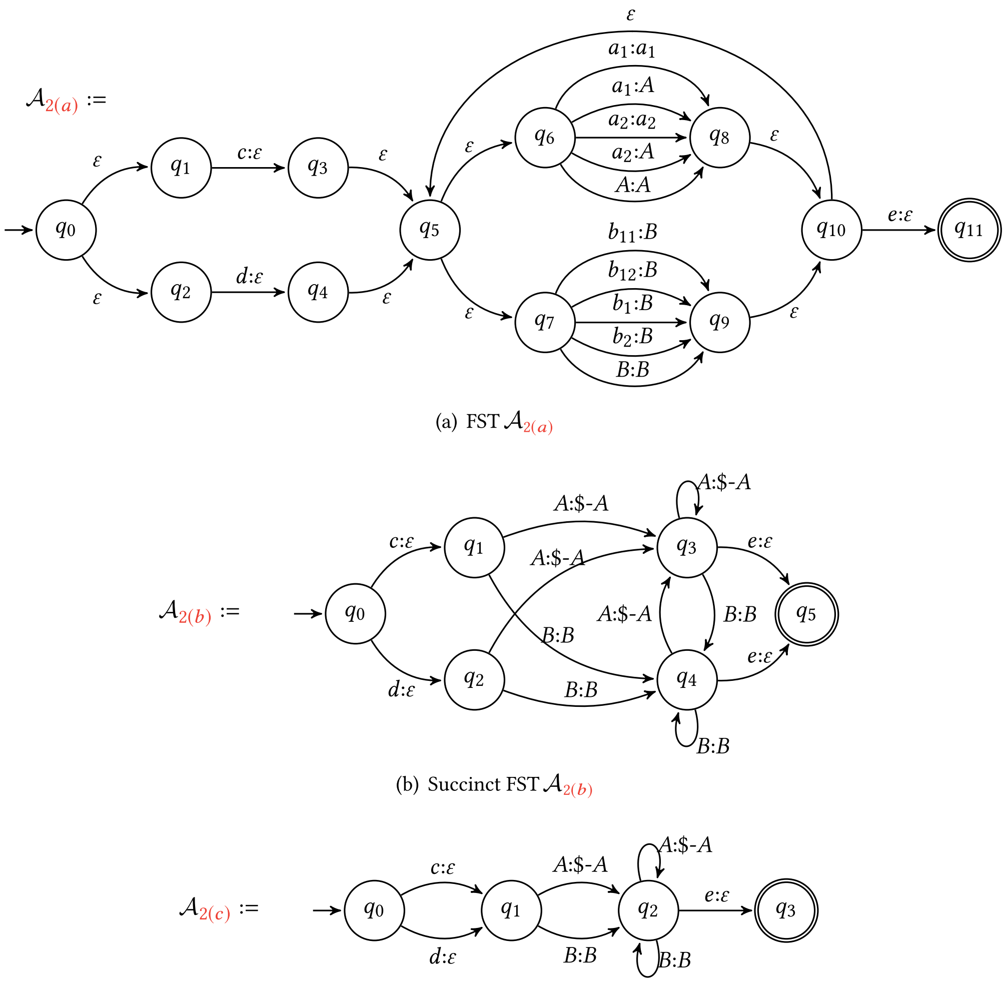
4.3 Computational Model

We translate patterns expressions into finite-state transducers (FSTs), which are a natural computational model for pattern expressions. An FST is a type of finite-state machine for string-to-string translation [34]. FSTs are similar to finite-state automata but additionally label transitions with output strings. Conceptually, they read an input string and translate it to an output string in a nondeterministic fashion. We will use FSTs to specify subsequence predicates $P(S,T)$: 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 finite 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
- $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}) \times Q$ is a transition relation.

For every transition $(q_{from}, in, out, q_{to}) \in \Delta$, we require that $out \in \text{anc}(in) \cup \{\varepsilon\}$ and that whenever $in = \varepsilon$ then $out = \varepsilon$. 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 4.2). Figure 2(a) shows an



(a) FST $\mathcal{A}_{2(a)}$

(b) Succinct FST $\mathcal{A}_{2(b)}$

(c) Minimized sFST $\mathcal{A}_{2(c)}$

Fig. 2. FST (a), sFST (b), and minimized sFST (c) for $[c|d]([A^{\uparrow} \mid B^{\uparrow}_{=}]^{+})e$.

example FST $\mathcal{A}_{2(a)}$ (for our running example), where $q_S = q_0$, $Q_F = \{ q_{11} \}$, and each transition is marked with *in*:*out* labels. We refer to transitions with *in* = ε (and thus *out* = ε) as ε-*transitions*. These transitions are marked with ε in the figure.

Runs and Outputs. Let $T = t_1 t_2 \ldots t_n$ be an input sequence. A *run* for T is a sequence $p = p_1 p_2 \ldots p_m$ of transitions where, for each $1 \leq i \leq m$, we have that $p_i = (q_i, w_i, w'_i, q'_i) \in \Delta$, $q_1 = q_S$, $q_{i+1} = q'_i$, and $w_1 w_2 \ldots w_m = T$. (Recall that $w_i \in \Sigma \cup \{ \varepsilon \}$, 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 $q_m \in Q_F$, then we refer to p as an *accepting run*. The output $O(p)$ of run p is the sequence $S = w'_1 \ldots w'_m$ of output labels, where we omit all w'_i with $w'_i = \varepsilon$ and set $S = \varepsilon$ if all $w'_i = \varepsilon$. The set of sequences generated by FST $\mathcal{A}$ is given by

$$G_{\mathcal{A}}(T) = \{ O(p) \neq \varepsilon \mid p \text{ is an accepting run of } \mathcal{A} \text{ for } T \}. \tag{3}$$

Example 4.1. Consider the FST $\mathcal{A}_{2(a)}$ of Figure 2(a). The FST corresponds to the constraint of Example 3.1. $\mathcal{A}_{2(a)}$ has two accepting runs for sequence $T_1 = ca_1b_{12}e$, which are given by

$$p_1 = q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow{c:\varepsilon} q_3 \xrightarrow{\varepsilon} q_5 \xrightarrow{\varepsilon} q_6 \xrightarrow{a_1:a_1} q_8 \xrightarrow{\varepsilon} q_{10} \xrightarrow{\varepsilon} q_5 \xrightarrow{\varepsilon} q_7 \xrightarrow{b_{12}:B} q_9 \xrightarrow{\varepsilon} q_{10} \xrightarrow{e:\varepsilon} q_{11},$$

with output $O(p_1) = a_1B$, and

$$p_2 = q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow{c:\varepsilon} q_3 \xrightarrow{\varepsilon} q_5 \xrightarrow{\varepsilon} q_6 \xrightarrow{a_1:A} q_8 \xrightarrow{\varepsilon} q_{10} \xrightarrow{\varepsilon} q_5 \xrightarrow{\varepsilon} q_7 \xrightarrow{b_{12}:B} q_9 \xrightarrow{\varepsilon} q_{10} \xrightarrow{e:\varepsilon} q_{11},$$

with output $O(p_2) = AB$. Thus, $G_{\mathcal{A}_{2(a)}}(T_1) = \{ a_1B, AB \}$, as desired. There is no accepting run for T_2, so that $G_{\mathcal{A}_{2(a)}}(T_2) = \emptyset$. Observe that $\mathcal{A}_{2(a)}$ generates precisely the P-sequences of Example 3.1.

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, accepting or not.

Lemma 4.2. *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 that $O(p) \sqsubseteq T$.*

Proof. The proof is by induction. For $T = \varepsilon$, the assertion holds, because every run for T must consist of only ε-transitions so that $G(p) = \varepsilon \sqsubseteq T$. Now suppose that the assertion holds for some sequence $T' \in \Sigma^*$. We show that it then also holds for $T = T'w$ with $w \in \Sigma$. Let p be an arbitrary run for T and let S be the sequence $O(p)$. We decompose p into two sequences of transitions: a prefix p' of a run for T' with output S' and a suffix p_w that reads w and outputs S_w. Note that such a decomposition is always possible. We have $S = S'S_w$. Since p' is a run for T', we have that $S' \sqsubseteq T'$ by the induction hypothesis. Now observe that p_w must contain exactly one transition with input label w and that all other transitions must be ε-transitions, because otherwise p would not be a run for T. Let w' be the output label of the transition with input label w. Then $S_w = w'$. By the definition of FSTs, we must have $w' \in \text{anc}(w) \cup \{ \varepsilon \}$, which implies that $w' \sqsubseteq w$. Since $S' \sqsubseteq T'$ and $S_w \sqsubseteq w$, we obtain $S = S'S_w \sqsubseteq T'w = T$. □

Note that not all subsequence predicates can be expressed with FSTs. For instance, there is no FST for predicate "all subsequences of form a^*b^* with an equal number of a's and b's," since this predicate cannot be expressed with a finite number of states. FSTs are a suitable 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 Sections 5 and 7).

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 $G_E(T) \overset{\text{def}}{=} G_{\mathcal{A}(E)}(T)$.

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 Table 2, column "FST."

The translation rules for composite expressions mirror the classical Thompson construction [46] for translating regular expressions to finite state automata.[7] For example, expression E_{ex} of Equation (2) translates to the FST of Figure 2(a).

Succinct 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 FST for "$(.^\uparrow)$" has $\Theta(nd)$ transitions. To avoid this explosion of FST size, we define a variant of FSTs that has

[7] All translation rules can be implemented without introducing any ε-transitions; we follow this approach in our actual implementation but use ε-transitions in our example FSTs for improved readability.

Table 2. Translation Rules for Item Expressions (Where $w, w', w'' \in \Sigma$)

Expr.	Matches	Captured	Produces	FST	Succinct FST
$w_=$	w	No	ε	$\{q_S \xrightarrow{w:\varepsilon} q_F\}$	$\{q_S \xrightarrow{w_=:\varepsilon} q_F\}$
		Yes	w	$\{q_S \xrightarrow{w:w} q_F\}$	$\{q_S \xrightarrow{w_=:w} q_F\}$
w	$w' \in \mathrm{desc}(w)$	No	ε	$\{q_S \xrightarrow{w':\varepsilon} q_F \mid w' \in \mathrm{desc}(w)\}$	$\{q_S \xrightarrow{w:\varepsilon} q_F\}$
		Yes	w'	$\{q_S \xrightarrow{w':w'} q_F \mid w' \in \mathrm{desc}(w)\}$	$\{q_S \xrightarrow{w:\$} q_F\}$
$.$	$w \in \Sigma$	No	ε	$\{q_S \xrightarrow{w:\varepsilon} q_F \mid w \in \Sigma\}$	$\{q_S \xrightarrow{.:\varepsilon} q_F\}$
		Yes	w	$\{q_S \xrightarrow{w:w} q_F \mid w \in \Sigma\}$	$\{q_S \xrightarrow{.:\$} q_F\}$
$w^{\uparrow}$	$w' \in \mathrm{desc}(w)$	Yes	$\mathrm{anc}(w') \cap \mathrm{desc}(w)$	$\{q_S \xrightarrow{w':w''} q_F \mid w' \in \mathrm{desc}(w), w'' \in \mathrm{anc}(w') \cap \mathrm{desc}(w)\}$	$\{q_S \xrightarrow{w:\$\text{-}w} q_F\}$
$.^{\uparrow}$	$w \in \Sigma$	Yes	$\mathrm{anc}(w)$	$\{q_S \xrightarrow{w:w'} q_F \mid w \in \Sigma, w' \in \mathrm{anc}(w)\}$	$\{q_S \xrightarrow{.:\$\text{-}\top} q_F\}$
$w^{\uparrow}_{=}$	$w' \in \mathrm{desc}(w)$	Yes	w	$\{q_S \xrightarrow{w':w} q_F \mid w' \in \mathrm{desc}(w)\}$	$\{q_S \xrightarrow{w:w} q_F\}$

compact representations for the types of transitions that we need for capturing item expressions. We refer to this variant as *succinct FSTs (sFSTs)* and summarize their types of transitions in column "succinct FST" of Table 2. The sFST 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 sFST describes a set of transitions in the corresponding FST in a concise way. More specifically, sFSTs use as input labels ., w, and $w_=$ for all $w \in \Sigma$. Here, "." matches all input items, w matches all items in $\mathrm{desc}(w)$, and $w_=$ matches only item w. They 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 that are descendants of w, and \$-$\top$ the matched item and all its ancestors. The sFST translations for composite expressions remain unmodified.

Figure 2(b) shows the sFST $\mathcal{A}_{2(b)}$ for pattern expression $[c|d]([A^{\uparrow} \mid B^{\uparrow}_{=}]^{+})e$. Observe that the sFST has fewer transitions than its non-succinct counterpart of Figure 2(a). Here, we used translation rules for composite expressions that do not introduce ε-transitions, which in this case further reduces the number of transitions. We subsequently "minimize" our sFSTs, which further reduce the number of states and transitions; see Section 6.1. For our running example, we finally obtain the sFST $\mathcal{A}_{2(c)}$ shown in Figure 2(c). The sFSTs corresponding to the pattern expressions of Table 1 are shown in Figure 16 (Appendix A).

Simulating sFSTs. Algorithm 1 shows how to "naively" simulate an sFST $\mathcal{A} = (Q, q_s, Q_F, \Sigma, \Delta)$. Here the *transition function*

$$\delta(q, w) = \{\, (out, q_{to}) \mid (q, in, out, q_{to}) \in \Delta,\ in \text{ matches } w\,\}$$

denotes the set of (output label, state)-pairs that can be reached from state q by consuming input item w (see column "Matches" in Table 2). Intuitively, we simulate the sFST by starting with the initial state q_S of the sFST (line 2) and repeatedly selecting a transition for which the input label matches the next input item t_{pos} (line 9). If there are multiple such transitions, then we select them one by one (via backtracking). As we move from state to state, we append items that are encoded by the output labels (column "Produces" in Table 2) of the selected transitions to an output buffer (S, lines 10–19). As before, if a transition encodes more than one output item, we append them one by

ALGORITHM 1: Naive sFST simulation

```
Require: sFST $\mathcal{A} = (Q, q_s, Q_F, \Sigma, \Delta)$, $T = t_1 \dots t_{|T|}$
Ensure: $G_{\mathcal{A}}(T)$
 1: $G_{\mathcal{A}}(T) \leftarrow \emptyset$                          // set of generated sequences
 2: Step($q_S, 1, \epsilon$)
 3:
 4: void Step($q, pos, S$):                    // (current state, input pos., buffer)
 5: if $q \in Q_F$ and $pos > |T|$ then
 6:    if $S \neq \epsilon$ then
 7:       $G_{\mathcal{A}}(T) \leftarrow G_{\mathcal{A}}(T) \cup \{ S \}$
 8:    return
 9: for all $(out, q_{to}) \in \delta(q, t_{pos})$ do                // empty if pos > |T|
10:    switch ($out$)
11:       case $\epsilon$:
12:       Step($q_{to}, pos + 1, S$)
13:       case $w$:
14:       Step($q_{to}, pos + 1, Sw$)
15:       case \$:
16:       Step($q_{to}, pos + 1, St_{pos}$)
17:       case \$-$x$ for $x \in \Sigma \cup \{\top\}$:
18:       for all $w' \in \text{anc}(t_{pos}) \cap \text{desc}(x)$ do
19:          Step($q_{to}, pos + 1, Sw'$)
```

one (again via backtracking, lines 18 and 19).[8] To keep notation concise, we define $\text{desc}(\top) = \Sigma$. If we reach a final state after consuming all input items, then we output the buffer, which then contains a generated sequence (lines 5–8).

Consider the sequence $T_3 = da_2a_1a_2b_{11}e$ of our example database $\mathscr{D}_{ex}$ and the sFST $\mathcal{A}_{2(c)}$ of Figure 2(c). In the first invocation of Step, we have $q = q_0$, $t_{pos} = t_1 = d$, and $S = \varepsilon$. Since $\delta(q_0, d) = \{ (\varepsilon, q_1) \}$, we proceed to line 12 and invoke Step with $q = q_1$, $t_{pos} = t_2 = a_2$, $S = \varepsilon$. We have $\delta(q_1, a_2) = \{ (\$\text{-}A, q_2) \}$, so that we proceed to line 19 and invoke Step with $q = q_2$, $t_{pos} = t_3 = a_1$, and $S = a_2$. After consuming input items a_1, a_2, and b_{11} in a similar fashion, we invoke step with $q = q_2$, $t_{pos} = t_6 = e$, and $S = a_2a_1a_2B$. Since $\delta(q_2, e) = \{ (\varepsilon, q_3) \}$, we proceed to state $q = q_3$ and $pos = 6$ without further modifying the buffer. Finally, since $q_3 \in Q_F$ is a final state and we consumed the entire input, we add buffered sequence $S = a_2a_1a_2B$ to the set $G_{\mathcal{A}_{2(c)}}(T_3)$ in line 7. The algorithm then backtracks and generates sequences a_2a_1AB, a_2Aa_2B, a_2AAB, Aa_1a_2B, Aa_1AB, AAa_2B, and $AAAB$.

Nondeterminism. Naive sFST simulation involves backtracking when multiple transitions leaving a state match the same input item and/or when a transition has an output label of form \$-w or \$-$\top$. The standard way to avoid nondeterminism is to use some form of FST determinization. Mohri [34] showed that the classical powerset construction algorithm by Rabin and Scott [41] for non-deterministic finite-state automata (NFA) can be extended to determinize *sequential* FSTs. However, this approach does not work out of the box for us, since our FSTs are not sequential. We discuss this issue in Section 6.

[8] A more efficient procedure, which reduces repeated computations, would be to append a description of all output items to buffer S. We do not follow this procedure to allow for efficient mining; see Section 5.

5 PATTERN MINING

We now turn our attention to mining P-frequent sequences from a sequence database. We assume that the subsequence predicate P is described by a sFST $\mathcal{A}$; e.g., the sFST can be obtained by translating a pattern expression and subsequently minimizing it (cf. Section 6). We propose three methods for mining P-frequent sequences: Naïve, DESQ-COUNT, and DESQ-DFS.

The naïve approach is to compute all P-generated sequences for each input sequence, count how often each sequence has been obtained, and output the ones that are frequent. DESQ-COUNT improves on the naïve approach by only generating sequences that do not contain globally infrequent items. Finally, DESQ-DFS is based on depth-first projection-based methods [39, 40] and is generally more efficient than DESQ-COUNT when the set of P-generated sequences is large.

5.1 Naïve Approach

The naïve "generate-and-count" approach is to compute $G_{\mathcal{A}}(T)$ for each input sequence $T \in \mathscr{D}$ via sFST simulation and count how often each sequence has been generated (cf. Equation (1)). The naïve approach is generally inefficient, because it considers many globally infrequent sequences. For example, we obtain

$$\begin{aligned} G_{A_{ex}}(T_3) &= \{AAAB, AAa_2B, Aa_1AB, Aa_1a_2B, \\ &\quad a_2AAB, a_2Aa_2B, a_2a_1AB, a_2a_1a_2B\} \end{aligned}$$

for input sequence T_3, but only $AAAB$ and Aa_1AB are P-frequent.

5.2 DESQ-COUNT

DESQ-COUNT reduces the number of sequences that are generated and counted by making use of item frequencies. In more detail, denote by $f(w, \mathscr{D}) = |\{T \in \mathscr{D} \mid w \sqsubseteq T\}|$ the *frequency* of item w. We say that item w is *frequent* if $f(w, \mathscr{D}) \geq \sigma$. Similar to many prior FSM algorithms, DESQ-COUNT first generates an *f-listF*, which contains all items along with their frequencies. For our example database, we obtain f-list

$$F_{ex} = \{A{:}6, e{:}6, B{:}6, a_1{:}6, d{:}3, b_2{:}3, b_1{:}2, c{:}2, b_{12}{:}1, b_{11}{:}1, a_2{:}1\}. \tag{4}$$

Note that the f-list is independent of the subsequence constraint and can be precomputed. In DESQ-COUNT, we make use of the f-list to reduce the size of $G_{\mathcal{A}}(T)$. Denote by

$$G^F_{\mathcal{A}}(T) = \{\, S \in G_{\mathcal{A}}(T) \mid \forall w \in S : f(w, \mathscr{D}) \geq \sigma \,\}$$

the subset of generated sequences that do not contain infrequent items. For T_3, we have $G^{F_{ex}}_{\mathcal{A}_{2(c)}}(T_3) = \{\, AAAB, Aa_1AB\}$, which is much smaller than the full set $G_{\mathcal{A}_{2(c)}}(T_3)$ given above. DESQ-COUNT proceeds as the naïve approach, but replaces $G_{\mathcal{A}}(T)$ by $G^F_{\mathcal{A}}(T)$ for each $T \in \mathscr{D}$. Note that we do not fully compute $G_{\mathcal{A}}(T)$ to obtain $G^F_{\mathcal{A}}(T)$; see below.

The correctness of DESQ-COUNT is established by Lemma 4.2, which states that FSTs specify subsequence predicates, and the following observation.

Lemma 5.1. *Let P be a subsequence predicate and $S \in \Sigma^+$ be an arbitrary sequence. Then for all items $w \in S$, $f(w, \mathscr{D}) \geq f_P(S, \mathscr{D})$.*

Proof. Pick any $w \in S$ and input sequence $T \in \mathscr{D}$ such that $S \in G_P(T)$. Since P is a subsequence predicate, $S \sqsubseteq T$. Since $w \in S$, we have $w \sqsubseteq S$ and thus also $w \sqsubseteq T$. We obtain

$$\begin{aligned} f_P(S, \mathscr{D}) &= |\{\, T \in \mathscr{D} \mid S \in G_P(T) \,\}| \\ &\leq |\{\, T \in \mathscr{D} \mid w \sqsubseteq T \,\}| = f(w, \mathscr{D}). \end{aligned}$$ □

The lemma implies that P-frequent sequences must be composed of frequent items. We thus can safely prune all sequences that contain infrequent items from $G_{\mathcal{A}}(T)$.

As mentioned above, we directly compute the reduced set $G^F_{\mathcal{A}}(T)$ by adapting Algorithm 1 to work with the f-list. In more detail, we stop exploring a path through the sFST (via Step) as soon as an infrequent item is produced. To do so, we execute lines 14, 16, and 19 of Algorithm 1 only if the item appended to the buffer S is frequent.

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 $G^F_{\mathcal{A}}(T)$ are very large. The DESQ-DFS algorithm, which we present next, targets such cases.

5.3 DESQ-DFS

DESQ -DFS adapts the pattern-growth framework of PrefixSpan [39] to FSTs. Pattern growth methods arrange the output sequences in a tree, in which each node corresponds to a sequence S and is associated with a *projected database*, which consists of the set of input sequences in which S occurs. Starting with an empty sequence and the full sequence database, the tree is built recursively by performing a series of *expansions*. In each expansion, a frequent sequence S (of l items) is expanded to generate sequences (of $l + 1$ items) with prefix S, their projected databases, and their supports. In what follows, we describe how we adapt these concepts to mine P-frequent sequences. The corresponding algorithm for sFSTs is shown as Algorithm 2 and illustrated on our running example in Figure 3.

Projected Databases. For each sequence S, we store in its projected database the state of the simulations of $\mathcal{A}$ on all input sequences that generate S as a partial output. We refer to such a state as a *snapshot* for S. The snapshot concisely describes which items have been consumed, which state the sFST simulation is in, and which output has been produced so far. In more detail, suppose that we simulate a sFST $\mathcal{A}$ on input sequence $T = t_1 \cdots t_n$. Consider a partial run $p = p_1 \cdots p_m$ consisting of $m \leq n$ transitions. We generated output $S = O(p)$ and, under our running assumption that $\mathcal{A}$ does not contain ε-transitions (see Footnote 7), consumed prefix $T' = t_1 \cdots t_m$ of T at this time. If the output item of the last transition p_m is not empty (and thus agrees with the last item of S), then we say that triple $T[pos@q]$ is a *snapshot* for S, where $pos = m + 1$ is the position of next input item and q is the last state in p (that is, the current state of $\mathcal{A}$). The *projected database* for S consists of all snapshots for S and is given by

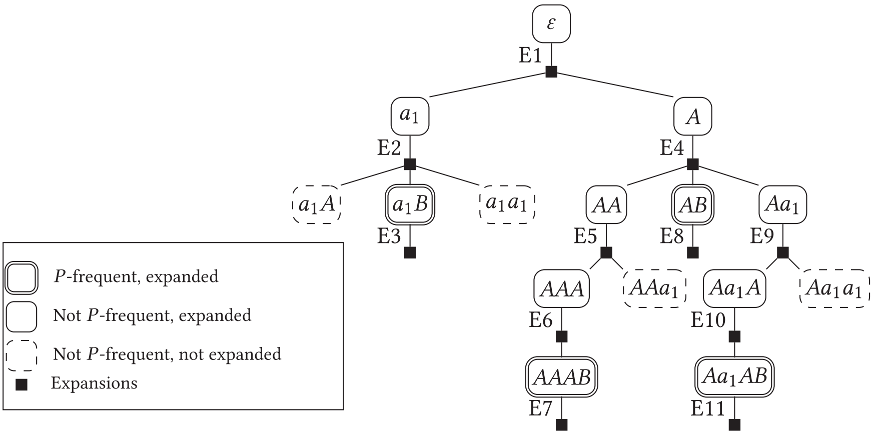
$$\mathrm{Proj}_{\mathcal{A}}(S, \mathscr{D}) = \{\, T[pos@q] \mid T \in \mathscr{D} \text{ and } T[pos@q] \text{ is a snapshot for } S \text{ on } \mathcal{A} \,\}.$$

Figure 3(b) shows some projected databases associated with some sequences for our running example. For example, we obtained the partial output a_1 only from input sequences T_1, T_4, and T_6. In each case, we consumed two items (the next item is at position 3) and ended in state q_2. We refer to the number of input sequences that can generate S as a partial output as the *prefix support* of S:

$$\mathrm{Presup}_{\mathcal{A}}(S, \mathscr{D}) = \{T \mid \exists pos, q : T[pos@q] \in \mathrm{Proj}_{\mathcal{A}}(S, \mathscr{D})\}.$$

In our example, $\mathrm{Presup}_{\mathcal{A}_{2(c)}}(a_1, \mathscr{D}_{ex}) = \{\, T_1, T_4, T_6 \,\}$. Note that even if an input sequence has multiple snapshots for S, it contributes only once to the prefix support.

Expansions. We now discuss Algorithm 2. We start with root node ε and all snapshots for ε (lines 1 and 2) and then perform a series of expansions (lines 3 and 10). In each expansion, we scan the projected database sequentially. For each snapshot $T[pos@q]$ (lines 6 and 7), we resume the sFST for T at item t_{pos} in state q (via IncStep, lines 12–27). The transducer is stopped as soon as an



(a) Search space

S	S.Proj	$\lvert S.\text{Presup}\rvert$	$\lvert S.\,\text{Sup}\rvert$
ε	$\langle T_1[1@q_0], T_2[1@q_0], T_3[1@q_0], T_4[1@q_0], T_5[1@q_0], T_6[1@q_0]\rangle$	6	0
a_1	$\langle T_1[3@q_2], T_4[3@q_2], T_6[3@q_2]\rangle$	3	0
a_1A	$\langle T_6[4@q_2]\rangle$	1	0
a_1B	$\langle T_1[3@q_2], T_4[3@q_2]\rangle$	2	2
a_1a_1	$\langle T_6[4@q_2]\rangle$	1	0

(b) Some projected databases, prefix supports, and supports

Fig. 3. Illustration of DESQ-DFS for $\mathscr{D}_{ex}$, $\mathcal{A}_{Fex}$, and $\sigma = 2$.

output item is produced or the entire input is consumed. In the former case, suppose we produce item w after consuming k more input items from T and thereby reach state q'. We then add the new snapshot $T[pos{+}k@q']$ to the projected database of child node Sw (lines 22, 24, and 27). In the latter case, if we end up in a final state (lines 13–15), then we conclude that $T \in \text{Sup}_{\mathcal{A}}(S, \mathscr{D})$ (see below). For example, both snapshots of a_1B reach final state q_3 by consuming all input items and without producing further output, so that $a_1B.\text{Sup} = \{\, T_1, T_4 \,\}$.

Pruning. The above expansion procedure allows us to prune partial sequences as soon as it becomes clear that they cannot be expanded to a P-frequent sequence. We use two pruning techniques. First, as in DESQ-COUNT, we consider item w only if it is frequent; otherwise, we ignore the new snapshot. For example, when expanding a_1, we do not create nodes for sequences that contain infrequent items; e.g., a_1b_{12} has snapshot $T_1[4@q_2]$ but contains infrequent item b_{12} (see Equation (4)). Second, we expand only those nodes S that have a sufficiently large prefix support—i.e., $\text{Presup}_{\mathcal{A}}(S, \mathscr{D}) \geq \sigma$—and stop as soon as there is no such node anymore. For example, we do not expand node a_1a_1, because it contains only one snapshot, but we require snapshots from at least $\sigma = 2$ different input sequences.

Correctness. Note that the size of the prefix support is monotonically decreasing as we go down the tree but always stays at least as large as the support. This property, which we establish next, is key to the correctness of DESQ-DFS.

LEMMA 5.2. *For every sequence $S \in \Sigma^*$ and item $w \in \Sigma$, we have* $\text{Presup}_{\mathcal{A}}(Sw, \mathscr{D}) \subseteq \text{Presup}_{\mathcal{A}}(S, \mathscr{D})$.

ALGORITHM 2: DESQ-DFS

```
Require: $\mathscr{D}$, sFST $\mathcal{A} = (Q, q_S, Q_F, \Sigma, \Delta)$, $\sigma$, f-list $F$
Ensure: $P$-frequent sequences for $\mathcal{A}$ in $\mathscr{D}$
$S \leftarrow \epsilon$                 // create root node; initially fields $S$.Proj = $S$.Sup = $\emptyset$
$S$.Proj $\leftarrow \{ T_1[1@q_S], \dots, T_{|\mathscr{D}|}[1@q_S] \}$
Expand($S$)

void Expand($S$):
for all $T[pos@q] \in S$.Proj do                 // simulate sFST for all snapshots
    IncStep($T, pos, q, S$)
if $|S$.Sup$| \geq \sigma$ then yield ($S$,$|S$.Sup$|$)                 // Output if $P$-frequent
for all $S' \in S$.Children do                 // expand if prefix support large enough
    if $|\{ T \mid \exists pos, q\colon T[pos@q] \in S$.Proj $\}| \geq \sigma$ then Expand($S'$)

void IncStep($T, pos, q, S$):                 // simulate until an item is produced
if $q \in Q_F$ and $pos > |T|$ then
    if $S \neq \epsilon$ then
        $S$.Sup $\leftarrow S$.Sup $\cup \{ T \}$                 // initially empty
    return
for all $(out, q_{to}) \in \delta(q, t_{pos})$ do
    switch (out)
        case $\epsilon$:
        IncStep($T, pos$+1$, q_{to}, S$)
        case $w$:
        if $f(w, \mathscr{D}) \geq \sigma$ then Append($S$, $w$, $T$, $pos$+1, $q_{to}$)
        case \$:
        if $f(t_{pos}, \mathscr{D}) \geq \sigma$ then Append($S$, $t_{pos}$, $T$, $pos$+1, $q_{to}$)
        case \$-$x$, $x \in \Sigma \cup \{ \top \}$:
        for all $w' \in$ anc($t_{pos}$) $\cap$ desc($x$) do
            if $f(w', \mathscr{D}) \geq \sigma$ then Append($S$, $w'$, $T$, $pos$+1, $q_{to}$)

void Append($S, w, T, pos, q$):
$S$.Children $\leftarrow S$.Children $\cup \{ Sw \}$                 // node $Sw$ is created if new
$Sw$.Proj $\leftarrow Sw$.Proj $\cup \{ T[pos@q] \}$                 // initially empty
```

Proof. Pick any $S \in \Sigma^*$, $w \in \Sigma$, and $T = t_1 \cdots t_n \in \mathscr{D}$ with $T \in \text{Presup}_{\mathcal{A}}(Sw, \mathscr{D})$. Then there exists a run $p = p_1 \cdots p_m$ for prefix $T' = t_1 \cdots t_m$ and some $m \leq n$ such that $O(p) = Sw$. Recall that inputs (outputs) are consumed (generated) from left to right. We conclude that there exists some $m' < m$ such that run $p' = p_1 \cdots p_{m'}$ satisfies $O(p') = S$. Pick the shortest such run; then $p_{m'}$ outputs the last item of S. Since p' is additionally a run for $t_1 \cdots t_{m'}$, which is a prefix of T, we conclude that $T \in \text{Presup}_{\mathcal{A}}(S, \mathscr{D})$. □

We now establish the correctness of DESQ-DFS.

Theorem 5.3. *DESQ-DFS outputs each P-frequent sequence $S \in \Sigma^+$ with frequency $f_P(S, \mathscr{D})$. No other sequences are output.*

Proof. Let $\mathcal{A} = (Q, q_S, Q_F, \Sigma, \Delta)$ be a sFST and pick any sequence $S \in \Sigma^+$. We start with showing that Algorithm 2 correctly computes the P-support of S when expanding node S, i.e.,

S.Sup $= \mathrm{Sup}_{\mathcal{A}}(S, \mathscr{D})$ after the expansion. First pick any $T \in \mathrm{Sup}(S, \mathscr{D})$ with $T = t_1 \cdots t_n$. Then there is an accepting run $p = p_1 \cdots p_n$ for T. By arguments as in the proof of Lemma 5.2, there must be a smallest run $p' = p_1 \cdots p_m$, $m \leq n$, such that $O(p') = S$ as well. Let q_m (q_n) be the state reached in transition p_m (p_n). We conclude that snapshot $T[pos@q_m] \in \mathrm{Proj}_{\mathcal{A}}(S, \mathscr{D})$, where $pos = m + 1$, and thus $T \in \mathrm{Presup}(S, \mathscr{D})$. Since by definition $p_{m+1} \cdots p_n$ must output ε, Algorithm 2 follows transitions $p_{m+1} \cdots p_n$ without stopping when resuming snapshot $T[pos@q_m]$. By doing so, it consumes all the remaining items $t_{m+1} \cdots t_n$ of T and reaches final state q_n. It thus includes T into S.Sup (lines 13–15). Now pick $T \notin \mathrm{Sup}_{\mathcal{A}}(S, \mathscr{D})$. Since there is no accepting run for T, Algorithm 2 cannot reach a final state after consuming T so that it does not include T into S.Sup. Putting both together, S.Sup $= \mathrm{Sup}_{\mathcal{A}}(S, \mathscr{D})$ after expanding S, as desired. We conclude that Algorithm 2 computes the correct frequency $f_P(S, \mathscr{D}) = |\mathrm{Sup}_{\mathcal{A}}(S, \mathscr{D})|$. Therefore, S is output only if it is P-frequent (line 8). Note that for $S = \varepsilon$, we have ε.Sup $= \emptyset$ (see line 13) so that ε is not output.

Let $S \in \Sigma^+$ be a P-frequent sequence. It remains to show that Algorithm 2 reaches and expands node S. First observe that for any prefix S' of S, we have

$$\mathrm{Presup}(S', \mathscr{D}) \supseteq \mathrm{Presup}(S, \mathscr{D}) \supseteq \mathrm{Sup}(S, \mathscr{D}).$$

Here the first inclusion follows from Lemma 5.2, and the second inclusion follows from the above arguments. Since S is P-frequent, we have $|\mathrm{Sup}(S, \mathscr{D})| \geq \sigma$, which implies $|\mathrm{Presup}(S', \mathscr{D})| \geq \sigma$. Since every node on the path from ε to S corresponds to a prefix of S, Algorithm 2 does not prune any of these nodes due to too low prefix support (line 10). To complete the proof, recall that S cannot contain an infrequent item by Lemma 5.1. Thus, none of the nodes on the path from ε to S are pruned due to too low item frequency either (lines 22, 24, or 27). We conclude that Algorithm 2 reaches and expands node S. □

To improve efficiency, our actual implementation of Algorithm 2 does not explicitly compute supports and prefix supports but directly counts their sizes.

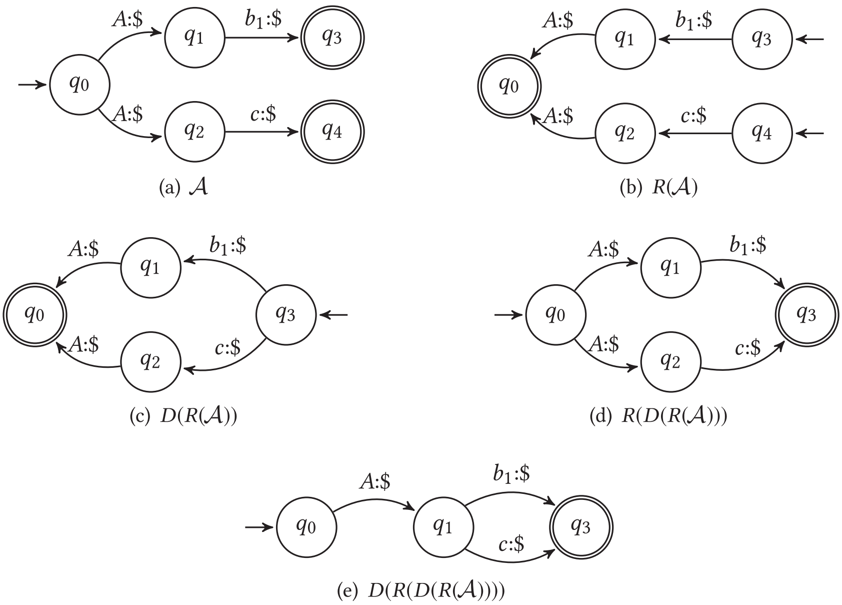
6 OPTIMIZATIONS

sFST simulation forms the basis of our mining algorithms discussed above. In this section, we discuss four optimizations for sFST simulation, which we also implemented for our experiments in Section 7. In Section 6.1, we explain how we partially determinize sFSTs to reduce backtracking and to reduce the number of states and transitions. In Section 6.2, we present a technique that enables us to detect and stop early runs that provably accept without producing further output, no matter which input items are still unread. In Section 6.3, we propose a pruning method that enables us to prune irrelevant input sequences during mining, i.e., input sequences for which the simulation is guaranteed to have no accepting run. Finally, in Section 6.4, we propose a two-pass approach that prunes irrelevant input sequences and further removes unnecessary nondeterminism by avoiding transitions leading to non-accepting runs.

All of our optimizations involve algorithms that take worst-case exponential time in the size of the FST $\mathcal{A}$ (but not in the size of the sequence database). At least for the pattern expressions considered in our experiments, the methods proposed here did not suffer from such exponential blow-up and, in fact, led to greatly improved overall running times.

6.1 Determinizing and Minimizing sFSTs

The naive sFST simulation algorithm in Section 4.3 (Algorithm 1) has some obvious optimization potentials. The first is to try to reduce nondeterminism in the sFST before simulation to avoid backtracking, which can be very costly. Mohri [34] studied FST determinization and showed that the classical powerset construction for non-deterministic finite-state automata (NFA) [41] can be



(a) $\mathcal{A}$

(b) $R(\mathcal{A})$

(c) $D(R(\mathcal{A}))$

(d) $R(D(R(\mathcal{A})))$

(e) $D(R(D(R(\mathcal{A}))))$

Fig. 4. Minimizing sFST for expression $(Ab_1|Ac)$.

extended to determinize *sequential* FSTs. Here, an FST is *sequential* if for each input there is at most one output. Unfortunately, this algorithm cannot be used naively, because the transformations expressed by our pattern expressions (and therefore also our sFSTs) are not sequential in general.

An orthogonal way to reduce nondeterminism is to delay some of the output. More precisely, an FST is *p-subsequential* if there are at most p outputs per input. For p-sequential transducers, it is possible to delay output until after the input has been consumed entirely, thus avoiding nondeterminism and backtracking. However, our FSTs are often even not p-subsequential. For example, the number of outputs for expression $[.^*(.)]^+$ (all subsequences) is exponential in the input size and thus not bounded by a constant p. Furthermore, delaying the output prevents us from doing other optimizations (as in DESQ-DFS, in particular). For these reasons, avoiding nondeterminism without limiting our pattern language is challenging.

We therefore apply a simple strategy that can remove much of the nondeterminism. A bonus is that the technique also allows us to eliminate unnecessary sFST states and, therefore, to "minimize" the sFST (according to a criterion that we make clear later). Observe that FSTs are isomorphic to NFAs if we treat input and output labels as atomic labels, that is, instead of having transitions $(q_{from}, in, out, q_{to}) \in \Delta$, we just consider the NFA with transitions $(q_{from}, in{:}out, q_{to})$ and treat *in*:*out* as a single symbol. Denote by $N(\mathcal{A})$ the NFA obtained from sFST $\mathcal{A}$ in such a way. We partially determinize and minimize $\mathcal{A}$ by determinizing and minimizing $N(\mathcal{A})$.

In principle, we can apply any NFA minimization algorithm to minimize $N(\mathcal{A})$. We implemented Brzozowski's [15] algorithm, because it is simple and was empirically shown to be very fast on NFAs in practice, especially when the alphabet is large [4]. Figure 4 illustrates such an application of Brzozowski's algorithm to an sFST $\mathcal{A}$ for pattern expression $(Ab_1|Ac)$. We start with $\mathcal{A}$ (Figure 4(a)) and construct a *reverse sFST* $R(\mathcal{A})$ (Figure 4(b)) by (i) reversing the direction of the transitions of $\mathcal{A}$, and (ii) swapping initial and final states. We then obtain the sFST $D(R(\mathcal{A}))$ (Figure 4(c)) by applying the powerset construction algorithm for converting NFAs to DFAs on $N(R(\mathcal{A}))$. We then repeat the process one more time (Figures 4(d) and 4(e)) to obtain the minimized sFST (Figure 4(e)).

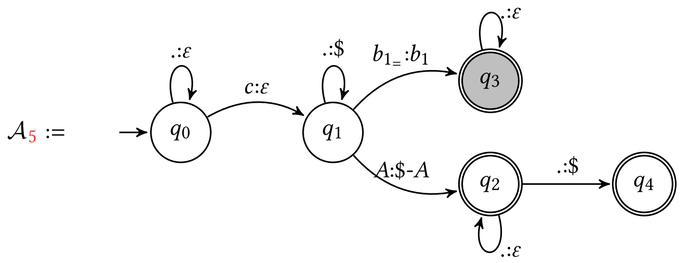


Fig. 5. Example sFST $\mathcal{A}_5$ with a final-complete annotation (shown in gray).

The above minimization also helps to reduce nondeterminism in cases when a state has two transitions with the same input and output label. Consider for example input sequence $T = a_1c$. When we simulate the sFST of Figure 4(a), we have $\delta(q_0, a_1) = \{ (a_1, q_1), (a_1, q_2) \}$ so that sFST simulation tries both options via backtracking. However, for the minimized sFST of Figure 4(e), we have $\delta(q_0, a_1) = \{ (a_1, q_1) \}$ and thus simulation avoids any backtracking.

Note that the algorithm does not always result in a minimal sFST as it is limited by treating input and output labels as atomic labels. Instead, it computes an sFST that is isomorphic to the minimal DFA for the language with atomic labels. The sFST in question can still have some nondeterminism, because, for instance, the two labels .:\$ and $b_{1_=}$:b_1 are different in the DFA but in the sFST they both match the input item b_1. Indeed, the sFST $\mathcal{A}_5$ in Figure 5 is nondeterministic even though $N(\mathcal{A}_5)$ is deterministic. In principle, the resulting sFST can be optimized even more by concatenating successive output labels as in classical FST minimization [35], but we do not explore this direction further.

We conclude with a note on complexity. Any algorithm that computes a minimal DFA from a given NFA has an unavoidable exponential blow-up in the worst case [24, Section 2.3.6]. Brzozowski's minimization algorithm on an NFA also runs in (single) exponential time. First, the computation of $D(R(N(\mathcal{A})))$ is in exponential time and generates an automaton exponential in $|R(N(\mathcal{A}))|$. However, there is no double exponential blow-up. Since we implement the computation of $D(R(D(R(N(\mathcal{A})))))$ such that only reachable states are considered, and since the final output is the minimal DFA for $N(\mathcal{A})$, which is single exponential in $|N(\mathcal{A})|$ the total output size is indeed single exponential in $|N(\mathcal{A})|$.

It is difficult to avoid the worst-case exponential blow-up upon determinization. Typical classes of languages for which the translation from NFA to DFA can have an exponential blow-up are defined by $(a|b)^*a(a|b)^n$ or by $(a|b)^*a(a|b)^nb(a|b)^*$ (with parameter n). The former class tests if the $n + 1$st symbol from the right is an a. The latter class tests if the sequence has an a somewhere, such that $n + 1$ positions later, there is a b. Each such expression has an equivalent NFA with $\Theta(n)$ states, but the minimal DFA requires $2^{\Theta(n)}$ states. However, for typical pattern expressions we have seen in frequent sequence mining (see Table 1 and Appendix A), we did not notice a significant blow-up.

In principle, one can try to avoid such a worst-case blow-up by imposing some kind of determinism constraint already on pattern expressions. But this does not solve the problem, for various reasons. First, the exponential blow-up is only avoided, because the expressions themselves can become exponentially larger. Second, and more seriously, the literature on determinism in ordinary regular expressions tells us that the determinism constraints restrict the expressiveness of regular expressions [14, 17], to such an extent that it breaks closure properties that make regular languages convenient to work with [30] (e.g., union, intersection, complement). Since we did not observe a major practical disadvantage in our experiments by allowing non-determinism, we decided that having a worst-case exponential blow-up was the lesser disadvantage.

6.2 Final-Complete Annotations

The simulation algorithm (Algorithm 1) generates output(s) only when the entire input sequence is matched. We noticed that this often results in overhead of processing input items that do not affect the final output(s) for accepting runs. To see this, consider the sFST $\mathcal{A}_5$ shown in Figure 5 and the input sequence

$$T_{ex} = e\, d\, a_1\, c\, b_1\, c\, d\, c\, b_2.$$

T_{ex} has one accepting run

$$q_0 \xrightarrow{e:\varepsilon} q_0 \xrightarrow{d:\varepsilon} q_0 \xrightarrow{a_1:\varepsilon} q_0 \xrightarrow{c:\varepsilon} q_1 \xrightarrow{b_1:b_1} q_3 \xrightarrow{c:\varepsilon} q_3 \xrightarrow{d:\varepsilon} q_3 \xrightarrow{c:\varepsilon} q_3 \xrightarrow{b_2:\varepsilon} q_3$$

with output b_1. Observe that the input items in the prefix $e\, d\, a_1$ and suffix $c\, d\, c\, b_2$ of T_{ex} do not affect the final output produced by the accepting run. This overhead of processing prefixes and suffixes that do not affect the final output is also incurred when we are interested in partially matching a pattern expression, i.e., when pattern expressions are of the form $.^*E.^*$ or $.^*E$, or $E.^*$.

In an sFST, we want to identify which final states are such that every suffix of the input is accepted and no more output can be produced. More formally, we call a final state $q \in Q_F$ *final-complete* if the following two conditions are satisfied:

(1) the sub-sFST with initial state q accepts every sequence $S \in \Sigma^*$, and
(2) no transitions that produce an output can be reached from q.

For example, sFST $\mathcal{A}_5$ has final-complete state q_3, which we mark in gray. Final state q_4 is not final-complete, since it violates both conditions. Final state q_2 satisfies the first condition—i.e., accepts $.^*$—but has a reachable transition $q_2 \xrightarrow{.:\$} q_4$ producing an output and therefore violating the second condition.

We make use of the final-complete annotations during sFST simulation as follows. Whenever we reach a final-complete state, we add the output buffered so far to $G_{\mathcal{A}}(T)$, whether or not the entire input has been consumed. In more detail, line 5 of Algorithm 1 (and line 13 of Algorithm 2) changes to

```
5: if ($q \in Q_F$ and $pos > |T|$) or ($q$ is final-complete).
```

In our running example, we can safely abort the run for T_{ex} after reaching final-complete state q_3, which avoids processing suffix eb_1a_1d. This approach short-circuits unnecessary processing of input items that will not affect the produced output for accepting runs. This is useful for our mining algorithms as we can safely avoid scanning to the end.

We now turn attention to how to determine the set of final-complete states. We consider each final state in turn. Given a final state q, condition (2) is easily verified by, say, depth-first search starting from q. Condition (1) is more challenging to verify: The problem of deciding whether the sub-sFST starting at state q accepts all inputs is equivalent to the problem of deciding whether a given NFA (obtained from the FST by dropping all output labels) accepts all inputs, which is a PSPACE-complete problem [45]. Instead, we only perform a test that is sound, but may be incomplete. More precisely, we consider the sub-sFST that has q as its starting state, and we only consider the transitions labeled $.:\varepsilon$. Notice that this sub-sFST can be seen as an NFA over the unary alphabet $\{\,.\,\}$ (by ignoring the output). We then determinize this NFA via the powerset construction and test if the resulting DFA accepts every word over the alphabet $\{\,.\,\}$. This happens if and only if the DFA has no reachable non-final state. This condition is easy to test, since the DFA is over a unary

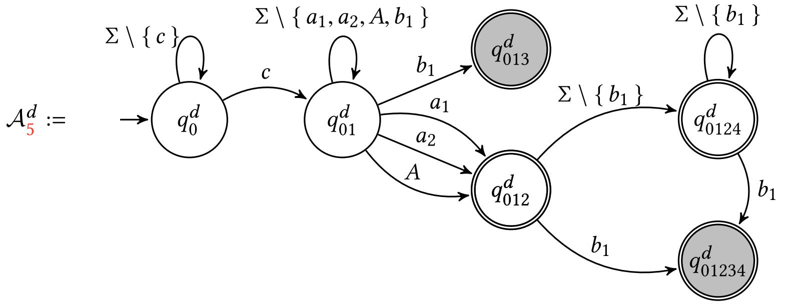

$\mathcal{A}_5^d :=$
$\Sigma \setminus \{ c \}$
c
$\Sigma \setminus \{ a_1, a_2, A, b_1 \}$
b_1
a_1
a_2
A
$\Sigma \setminus \{ b_1 \}$
$\Sigma \setminus \{ b_1 \}$
b_1
b_1
q_0^d
q_{01}^d
q_{013}^d
q_{012}^d
q_{0124}^d
q_{01234}^d

Fig. 6. DFA $\mathcal{A}_5^d$ corresponding to sFST $\mathcal{A}_5$. Final-complete states are shown in gray.

alphabet, i.e., its set of reachable states form a chain with a single backloop at the end. Note that this algorithm has worst-case exponential runtime due to determinization.[9]

Note that final-complete annotations described above only help to avoid processing of suffixes that do not affect the final output of accepting runs. In Section 6.4, we show how we leverage final-complete annotations to also avoid repeated processing of certain "unnecessary" prefixes.

6.3 Pruning Irrelevant Input Sequences

We now discuss a pruning technique, which allows us to prune input sequences that are guaranteed to have no accepting runs. More formally, we say that an input sequence T is $\mathcal{A}$*-relevant* for an sFST $\mathcal{A}$ if there is at least one accepting run for T. Similarly, we say the T is $\mathcal{A}$*-irrelevant* if there is no accepting run for T in $\mathcal{A}$. For example, sequence T_{ex} is $\mathcal{A}_5$-relevant, whereas $T = a_1\, c\, b_2\, c\, d\, c\, b_2$ is $\mathcal{A}_5$-irrelevant.[10] sFST simulation on $\mathcal{A}$-irrelevant input sequences never reaches a final state and leads to wasted computation of partial output sequences. Thus, pruning such input sequences can significantly improve overall efficiency of our mining algorithms.

We quickly determine whether or not an input sequence T is $\mathcal{A}$-relevant via a DFA $\mathcal{A}^d$ that accepts T if and only if it is $\mathcal{A}$-relevant. As mentioned earlier, FSTs are similar to finite-state automata but transitions are additionally labeled with outputs. Denote by $N_{in}(\mathcal{A})$ the NFA obtained from the canonical (non-succinct) FST for $\mathcal{A}$ by dropping all output labels. By construction, $N_{in}(\mathcal{A})$ accepts an input sequence T if and only if $\mathcal{A}$ has an accepting run for T. We convert $N_{in}(\mathcal{A})$ to an equivalent DFA $\mathcal{A}^d \overset{\text{def}}{=} D(N_{in}(\mathcal{A}))$. Given this DFA, we can determine in linear time whether or not an input sequence is relevant. In practice, we construct $\mathcal{A}^d$ directly from $\mathcal{A}$ instead of from $N_{in}(\mathcal{A})$ by adapting the subset construction algorithm to sFSTs.

Figure 6 shows the DFA $\mathcal{A}_5^d$ corresponding to sFST $\mathcal{A}_5$. Observe that $\mathcal{A}_5^d$ will reject the input sequence $T = a_1\, c\, b_2\, c\, d\, c\, b_2$ (which can thus be safely pruned during mining), whereas it will correctly accept the sequence T_{ex}.

In our running example, the DFA only has six states. In general, the number of DFA states can grow exponentially with the number of sFST states. Although in our experimental study the DFA construction worked well for most pattern expressions used there, our implementation also supports *lazy DFA construction* for situations where the exponential blow-up is unavoidable. Also note that the DFA is not succinct, which imposes additional memory overhead. In particular, item expressions that match many inputs produce many DFA transitions (e.g., . or w when desc(w) is

[9] And there is little hope to do better: The problem of deciding whether an NFA over a unary alphabet accepts every word is coNP-complete [45]. As before, we do not expect to see worst-case behavior in practice.

[10] Cf. Equation (3). $G_{\mathcal{A}}(T) = \emptyset$ does not necessarily imply that T is $\mathcal{A}$-irrelevant; this can happen when $\mathcal{A}$ has an accepting run with ε output.

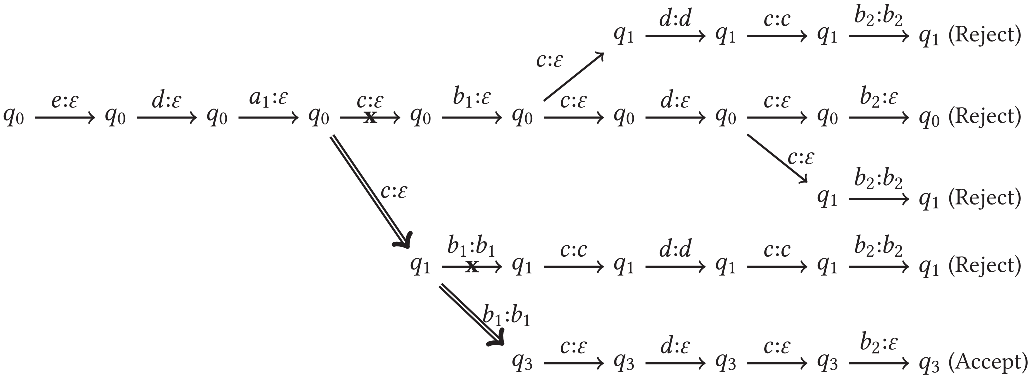

$q_0 \xrightarrow{e:\varepsilon} q_0 \xrightarrow{d:\varepsilon} q_0 \xrightarrow{a_1:\varepsilon} q_0 \xrightarrow{c:\varepsilon} q_0 \xrightarrow{b_1:\varepsilon} q_0 \xrightarrow{c:\varepsilon} q_0 \xrightarrow{d:\varepsilon} q_0 \xrightarrow{c:\varepsilon} q_0 \xrightarrow{b_2:\varepsilon} q_0$ (Reject)
$c:\varepsilon$ $q_1 \xrightarrow{d:d} q_1 \xrightarrow{c:c} q_1 \xrightarrow{b_2:b_2} q_1$ (Reject)
$c:\varepsilon$ $q_1 \xrightarrow{b_2:b_2} q_1$ (Reject)
$c:\varepsilon$
$q_1 \xrightarrow{b_1:b_1} q_1 \xrightarrow{c:c} q_1 \xrightarrow{d:d} q_1 \xrightarrow{c:c} q_1 \xrightarrow{b_2:b_2} q_1$ (Reject)
$b_1:b_1$
$q_3 \xrightarrow{c:\varepsilon} q_3 \xrightarrow{d:\varepsilon} q_3 \xrightarrow{c:\varepsilon} q_3 \xrightarrow{b_2:\varepsilon} q_3$ (Accept)
x
x

Fig. 7. Runs for input sequence $T_{ex} = e\,d\,a_1\,c\,b_1 c\,d\,c\,b_2$ on sFST $\mathcal{A}_5$.

large). Our implementation uses a number of engineering tricks to reduce the memory footprint; e.g., handling . specially, using bitmap indexes, or sharing data and index structures between DFA states to the extent possible.

The final-complete annotations described in the previous section also improve DFA construction and its simulation. During DFA construction, we can ignore all outgoing transitions of final-complete states, and mark a DFA state as final-complete if any of its corresponding sFST states is final-complete (cf. Figure 6). Final-complete annotations in the DFA enable us to stop a DFA simulation early: the DFA accepts an input sequence if it reaches a final state upon consuming the entire input or if it reaches a final-complete state (even when the input is not entirely consumed). For example, we can determine the $\mathcal{A}_5$-relevance of T_{ex} as soon as we reach state q^d_{013} of $\mathcal{A}^d_5$; we thus do not need to process suffix $c\,d\,c\,b_2 d$.

We integrate pruning $\mathcal{A}$-irrelevant sequences into DESQ-COUNT and DESQ-DFS by only considering $\mathcal{A}$-relevant input sequences; all other input sequences are pruned upfront. The algorithms remain unmodified otherwise. Pruning $\mathcal{A}$-irrelevant sequences is beneficial if the sequence database contains many $\mathcal{A}$-irrelevant sequences. In the worst case, when all input sequences are $\mathcal{A}$-relevant, nothing is pruned and DFA simulation leads to additional overhead. Moreover, even when an input sequence is $\mathcal{A}$-relevant, sFST simulation may still involve unnecessary backtracking, i.e., it may still process non-accepting runs. In the next section, we propose a two-pass approach that prunes $\mathcal{A}$-irrelevant input sequences and additionally avoids processing non-accepting runs.

6.4 Two-Pass Approach

The naive sFST simulation involves backtracking whenever multiple transitions leaving a state match the same input item or when a transition has an output label of the form \$-w or \$-$\top$. While we try to avoid backtracking whenever possible, we feel that it is acceptable when each backtracking procedure is useful for producing an output sequence. For example, recall the sFST simulation example at the end of Section 4.3, where simulating sFST $\mathcal{A}_{2(c)}$ of Figure 2(c) on sequence T_3 of our example database involved backtracking to generate all output sequences. In this example, there was no "unnecessary" backtracking in the sense that it always led to an accepting run. But this is not always the case. For example, simulating sFST $\mathcal{A}_5$ on input sequence T_{ex} involves backtracking from non-accepting runs. Figure 7 illustrates all runs for T_{ex}, arranged in a trie. Here, the transitions $q_0 \xrightarrow{c:\varepsilon} q_0$ and $q_1 \xrightarrow{b_1:b_1} q_1$ (marked with "**x**") lead to four non-accepting runs. Such backtracking leads to wasted computation; we therefore want to avoid it. In what follows, we propose a

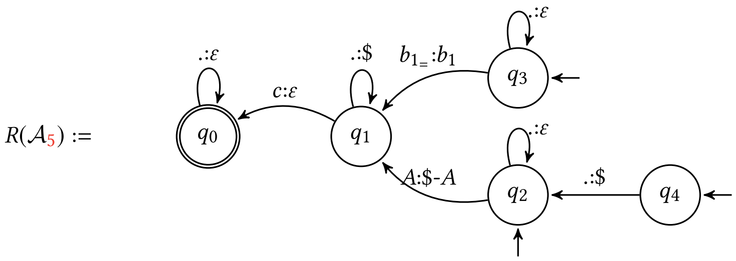


Fig. 8. Reverse sFST $R(\mathcal{A}_5)$ corresponding to sFST $\mathcal{A}_5$.

two-pass approach that completely avoids transitions leading to non-accepting runs and therefore also avoids such backtracking.

Consider an input sequence $T = t_1 \dots t_{|T|}$. The key idea in the two-pass approach is to efficiently precompute (before sFST simulation) sets $Q_1, \dots, Q_{|T|}$ of FST states such that, for each $1 \leq pos \leq |T|$ and each $q \in Q_{pos}$, the suffix $t_{pos+1} \dots t_{|T|}$ is accepted by $\mathcal{A}$ if it starts in state q. We make use of this information during simulation to only select transitions that lead to accepting runs.

More precisely, let $\mathcal{A}$ be an sFST and denote by $R(T) = t_{|T|} \dots t_1$ the reverse of input sequence T. Similarly, denote by $R(\mathcal{A})$ the sFST obtained from $\mathcal{A}$ by reversing all transitions, i.e., replacing every transition $(q_{from}, in, out, q_{to})$ by $(q_{to}, in, out, q_{from})$ and swapping initial and final states. For example, Figure 8 shows $R(\mathcal{A}_5)$. Here q_2, q_3, q_4 are the initial states and q_0 is the final state. For a given $R(\mathcal{A})$ and $R(T)$, we have

$$G_{R(\mathcal{A})}(R(T)) = \{\, R(S) \mid S \in G_{\mathcal{A}}(T) \,\},$$

i.e., the reverse sFST will produce all outputs in reverse. Further, consider an accepting run p for output sequence $S = s_1 \dots s_{|S|} \in G_{\mathcal{A}}(T)$. Consider the prefix of the run that consumes input $t_1, \dots, t_{pos}$; up to this time, we generated partial output $s_1 \dots s_{pos'}$, and the last transition has form (q', t_{pos}, out, q). Now consider $R(\mathcal{A})$, $R(T)$ and the reverse run $R(p)$. After consuming partial input $t_{|T|} \dots t_{pos+1}$, the reverse run has generated partial output $s_{|S|} \dots s_{pos'+1}$ and also reaches state q, ending with a transition of form (q'', t_{pos+1}, out, q).

We make use of the above observation in the two-pass approach as follows. Denote by $\delta'(q, w)$ the set of states in $R(\mathcal{A})$ than can be reached from state q by consuming item w:

$$\delta'(q, w) = \{\, q_{from} \mid (q_{from}, in, out, q) \in \Delta,\ in \text{ matches } w \,\}, \tag{5}$$

where Δ is the set of transitions of $\mathcal{A}$. We make two passes over an input sequence T. In the first pass, we read $R(T)$ and incrementally compute the sets $Q_{|T|}, \dots, Q_1, Q_0$ where, for every $|T| \geq pos \geq 1$, we have

$$\begin{aligned} Q_{|T|} &= Q_F \\ Q_{pos-1} &= \bigcup_{q \in Q_{pos}} \delta'(q, t_{pos}). \end{aligned} \tag{6}$$

Intuitively, for each state $q \in Q_{pos}$, there exists a path in $R(\mathcal{A})$ from some state in Q_F to q for the partial input $t_{|T|} \dots t_{pos+1}$. (Equivalently, there exists a path from state q to a final state in $\mathcal{A}$ for the partial input $t_{pos+1} \dots t_{|T|}$.) Moreover, if $q_S \notin Q_0$, then there exists no accepting run for $R(T)$ in $R(\mathcal{A})$ and consequently T is $\mathcal{A}$-irrelevant.

The second pass consists of a simulation of $\mathcal{A}$ on T that avoids runs that do not lead to acceptance. The central idea is the following. Consider an arbitrary partial run of $\mathcal{A}$ on a prefix $t_1 \cdots t_{pos}$

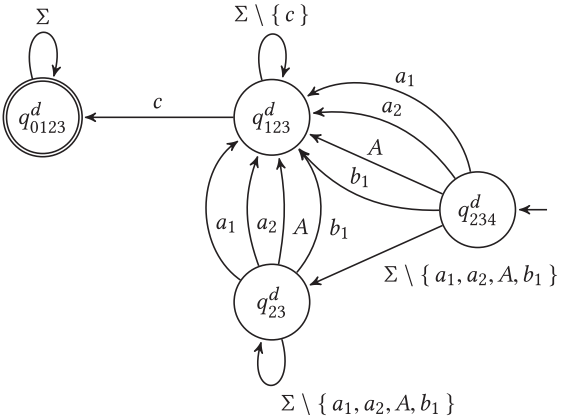

Σ
$\Sigma \setminus \{ c \}$
q^d_{0123}
c
q^d_{123}
a_1
a_2
A
b_1
q^d_{234}
a_1
a_2
A
b_1
$\Sigma \setminus \{ a_1, a_2, A, b_1 \}$
q^d_{23}
$\Sigma \setminus \{ a_1, a_2, A, b_1 \}$

Fig. 9. Reverse DFA corresponding to sFST $\mathcal{A}_5$.

of input T, ending in state q. Then we have

$$q \in Q_{pos} \iff \text{there exists an accepting run } q_0 q_1 \cdots q_{|T|} \text{ of } \mathcal{A} \text{ on } T \text{ with } q_{pos} = q. \tag{7}$$

Therefore, when we simulate $\mathcal{A}$ on T, we will only consider runs in which all states belong to one of the sets Q_{pos}.

Before we explain our implementation of the algorithm, we illustrate the main idea of the two-pass approach on input sequence T_{ex} and sFST $\mathcal{A}_5$. The following table illustrates the sets Q_{pos} computed for each t_{pos} (ignore the last row q^d_{pos} for now). Since $q_0 \in Q_0$, we know that T_{ex} is relevant.

pos	0	1	2	3	4	5	6	7	8	9
t_{pos}	–	e	d	a_1	c	b_1	c	d	c	b_2
Q_{pos}	$\{q_0, q_1, q_2, q_3\}$	$\{q_0, q_1, q_2, q_3\}$	$\{q_0, q_1, q_2, q_3\}$	$\{q_0, q_1, q_2, q_3\}$	$\{q_1, q_2, q_3\}$	$\{q_2, q_3\}$	$\{q_2, q_3\}$	$\{q_2, q_3\}$	$\{q_2, q_3\}$	$\{q_2, q_3, q_4\}$
q^d_{pos}	q^d_{0123}	q^d_{0123}	q^d_{0123}	q^d_{0123}	q^d_{123}	q^d_{23}	q^d_{23}	q^d_{23}	q^d_{23}	q^d_{234}

The second pass of the algorithm consists of simulating $\mathcal{A}_5$, while avoiding states not in the sets Q_{pos} for t_{pos}. For example, if after consuming eda_1, we are at state q_0, the automaton $\mathcal{A}_5$ can go to states q_0 and q_1 when reading the next (fourth) item c. But since $q_0 \notin Q_4$, we can safely avoid the transition $q_0 \xrightarrow{c:\varepsilon} q_0$ and immediately move to state $q_1 \in Q_4$. For the next input item b_1, we can avoid transition $q_1 \xrightarrow{c:\varepsilon} q_1$, since $q_1 \notin Q_5$. Figure 7 illustrates various runs for T_{ex} on $\mathcal{A}_5$. The two-pass approach avoids the transitions marked with "**x**" as desired.

6.4.1 Implementation. We now explain how we implement the two-pass algorithm. In the first pass, instead of computing the sets $Q_{|T|}, \ldots, Q_1, Q_0$ directly using $R(\mathcal{A})$ and Equation 6, we make use of a *reverse DFA*, i.e., a DFA obtained from $R(\mathcal{A})$. As the reverse FST, the reverse DFA processes the input sequence in reverse order. In contrast to the reverse FST, the DFA allows us to process each input item in constant time in this pass (since we do not need to construct output).

Figure 9 shows the reverse DFA corresponding to sFST $R(\mathcal{A}_5)$ shown in Figure 8. Each DFA state is annotated with the set of states of the FST $R(\mathcal{A})$ (and thus also $\mathcal{A}$) to which it corresponds; in the figure, these states are given as subscripts. For $R(T_{ex})$, the sequence of DFA states is

$$q^d_{0123} \xleftarrow{e} q^d_{0123} \xleftarrow{d} q^d_{0123} \xleftarrow{a_1} q^d_{0123} \xleftarrow{c} q^d_{123} \xleftarrow{b_1} q^d_{23} \xleftarrow{c} q^d_{23} \xleftarrow{d} q^d_{23} \xleftarrow{c} q^d_{23} \xleftarrow{b_2} q^d_{234}.$$

By construction, when we simulate the DFA backwards and reach state q^d_{pos} after consuming $t_{|T|} \dots t_{pos+1}$, the set Q_{pos} is directly given by the set of FST states corresponding to q_{pos}. Thus, to determine the sets $Q_{|T|}, \dots, Q_1, Q_0$, it suffices to run the reverse DFA and record its sequence of states.

If the reverse DFA does not accept $R(T)$, then we know that T is not relevant and we can immediately abort. Otherwise, we proceed to the second pass, which is the simulation of $\mathcal{A}$ on T, using only states in Q_{pos}. The implementation of the second pass is then straightforward: It essentially consists of Algorithm 1 with one change: In line 9, we additionally test whether $q_{to} \in Q_{pos+1}$ (stored as an annotation of the corresponding DFA state) and ignore the transition if so.

In our actual implementation, we also leverage final-complete annotations in the two-pass approach to avoid simulating $\mathcal{A}$ on the entire input sequence. In particular, such annotations in the reverse DFA allow us to determine suffixes in $R(T)$, and thereby prefixes in T, that do not affect the output; these prefixes are ignored in the second pass. We also use final-complete annotations in $\mathcal{A}$ (as described in Section 6.2) to avoid processing suffixes in T that do affect the output.

6.4.2 *Worst-Case Runtime of Two-Pass Approach.*

To shed some light on the potential improvements that the two-pass approach offers, we would like to investigate the worst-case runtime of computing,

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

with the two-pass approach. Recall that this set is directly used in the DESQ-COUNT algorithm; any improved running time here thus directly translates to an improved running time of DESQ-COUNT. We consider *data complexity* throughout, i.e., we treat the FST $\mathcal{A}$ and the hierarchy Σ as constants (so that, for example, the construction of $D(R(\mathcal{A}))$ takes constant time).

Naive sFST simulation computes $G_{\mathcal{A}}(T)$ by iterating through all (accepting) *runs* of $\mathcal{A}$ on T, that is, it implicitly computes

$$G'_{\mathcal{A}}(T) = \{(O(p), p) \mid \ p \text{ is is an accepting run of } \mathcal{A} \text{ for } T\}.$$

That is, it iterates through all accepting runs and, for each such run, adds $O(p)$ to the output if it is not empty and not in the output already. Algorithm 1 may require exponential time in $|T|$ between discovering two consecutive elements in $G'_{\mathcal{A}}(T)$, i.e., between two consecutive times it reaches line 7. We provide an example where Algorithm 1 considers exponentially many partial runs that turn out to be useless. Assume an item hierarchy with items $\{a_1, \dots, a_n\}$ that generalize to A and likewise for $\{b_1, \dots, b_n\}$ and B. Then, if Algorithm 1 has the input sequence $T = a_1 b_1 \cdots a_n b_n c$ and an sFST for the pattern expression $[.^*(A).^*]^*c \mid [.^*(B).^*]^*d$, it will consider $2^{\Theta(n)}$ many partial runs that capture the b_i, but none of them lead to acceptance, since T does not end with d.[11] We will prove that the two-pass approach successfully avoids such exponential computations. More precisely, it can enumerate the elements of $G'_{\mathcal{A}}(T)$ in linear delay, i.e., it can compute a first element in $G'_{\mathcal{A}}(T)$ in time $O(|T|)$ and, from there on, we can always compute a new element $G'_{\mathcal{A}}(T)$ in time $O(|T|)$ or conclude that no such element exists. The total time to produce all accepting runs is thus $O(|T||G'_{\mathcal{A}}(T)|)$, where $|G'_{\mathcal{A}}(T)|$ corresponds to the number of accepting runs.

The backward pass simply consists of running $D(R(\mathcal{A}))$ once over $R(T)$. This pass only needs to be performed once and costs time $O(|T|)$. (Given a DFA state q^d and an item w, our implementation precomputes the *DFA transition function* $\delta(q^d, w)$ that returns the next DFA state in constant time.) As a result, we obtain the sequence of sets $Q_0 \cdots Q_{|T|}$, where $q_S \in Q_0$ if $G'_{\mathcal{A}}(T)$ is non-empty.

[11] Similar examples can also be constructed without item hierarchies; or where the reason why exponentially many partial runs are unsuccessful is in the middle of T instead of at the end.

The forward pass then simulates $\mathcal{A}$ on T, taking into account the sets Q_i. That is, if we are in a state from Q_i and read input item t_{i+1}, we only consider states from Q_{i+1}. From here on, we essentially perform Algorithm 1, where in line 9, we additionally test if $q_{to} \in Q_{pos+1}$. By performing this change only, the algorithm will implicitly enumerate $G'_{\mathcal{A}}(T)$: Every execution of line 6 corresponds to a new accepting run. We can compute $G'_{\mathcal{A}}(T)$ by maintaining in Algorithm 1 the current partial run r of $\mathcal{A}$. This partial run is easy to maintain, because it is the sequence of states q for which $\textsc{Step}(q, pos, S)$ has been invoked on the recursion stack. In line 6, the algorithm would find a new element $(O(r), r) \in G'_{\mathcal{A}}(T)$.

From the correctness of Algorithm 1, it immediately follows that the just described algorithm computes $G'_{\mathcal{A}}(T)$. (It computes exactly the same elements of $G_{\mathcal{A}}(T)$ as before, but now we output on line 6 each S together with the run in which S was obtained.)

We now argue that the time between two consecutive outputs is at most $O(|T|)$, thereby establishing linear delay. Indeed, after the algorithm produced an output, it backtracks until it discovers

(1) in line 9, a next (out, q_{to})-pair with $q_{to} \in Q_{pos+1}$, or
(2) in line 18, a next output $w' \in \text{anc}(t_{pos}) \cap \text{desc}(x)$, or
(3) that there are no additional accepting runs.

This takes time at most $O(|T|)$. If the algorithm finds another pair or output (conditions (1) and (2) above), then it resumes the forward recursion. Since it only calls $\textsc{Step}$ using states $q_{to} \in Q_{pos+1}$, it does not need to backtrack again until it produces the next output (r, S); every call to $\textsc{Step}$ leads to an accepting run. Each test in condition (1) and (2) takes $O(1)$ time under our assumption that $\mathcal{A}$ and Σ are constants (since all required sets can be precomputed in constant time). The total time between outputs is therefore $O(|T|)$, since the algorithm has at most $|T|$ calls on the recursion stack. By a similar argument, the first output can also be produced in time $O(|T|)$.

THEOREM 6.1. *Assuming that the sFST $\mathcal{A}$ and the dictionary Σ have constant size, the elements of $G'_{\mathcal{A}}(T)$ can be enumerated, without repetitions, with $O(|T|)$ delay.*

We conclude with a note explaining why the two-pass approach does not compute $G_{\mathcal{A}}(T)$ in linear delay. Assume that $T = a^{2n}$ (a length $2n$ sequence only consisting of a's) and $\mathcal{A}$ is an FST for the pattern expression $[.^*(a).^*]\{n\}$. In this case, $G_{\mathcal{A}}(T) = \{a^n\}$, but the number of accepting runs of the FST on T is in $2^{\Theta(n \log n)}$.[12]

6.4.3 Integration in the Mining Algorithms. We now discuss how to integrate the two-pass approach into our mining algorithms. For both DESQ-COUNT and DESQ-DFS, we first construct the reverse DFA for $\mathcal{A}$ and proceed as follows.

For DESQ-COUNT, we simply replace sFST simulation by the two-pass approach. The second pass is performed only if the first pass determined that the input sequence is $\mathcal{A}$-relevant.

Integrating the two-pass approach in DESQ-DFS is slightly more involved, since we incrementally simulate $\mathcal{A}$ on all input sequences. We proceed as follows: We perform the first pass on all input sequences during the construction of the initial projected database for root note ε. In more detail, we read each input sequence T and simulate the reverse DFA. If sequence T is $\mathcal{A}$-relevant, then we add snapshot $T[1@q_S]$ to the projected database of root node ε and record the sequence of (reverse) DFA states $T.q^d_{|T|}, \ldots, T.q^d_1, T.q^d_0$ for later reference. We then perform expansions similar to DESQ-DFS: The only modification is that, in the $\textsc{IncStep}$ procedure, we additionally check whether $q_{to} \in T.Q_{pos}$ (line 17 of Algorithm 2).

[12] The number of accepting runs is proportional to the number of different n-element subsets of a $2n$-element set. Due to the binomial formula, this number is $2n!/(n! \cdot n!)$, which lies between $n!$ and $(2n!)$, both of which are in $2^{\Theta(n \log n)}$ due to Stirling's approximation.

7 EXPERIMENTAL EVALUATION

We conducted an experimental study on three publicly available real-world datasets: a collection of text documents (for text mining), a collection of product reviews (for customer behavior mining), and a collection of protein sequences. 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, and whether the optimizations of Section 6 are effective. A summary of our results is as follows:

(1) Many subsequence constraints can be expressed with pattern expressions.
(2) DESQ's sFST simulation algorithms are more than one order of magnitude faster than the (more general) methods of the state-of-the-art FST library OpenFST.
(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 P-subsequences per input were generated, DESQ-DFS was more than an order of magnitude faster than DESQ-COUNT and Naïve.
(6) The pruning of $\mathcal{A}$-irrelevant sequences sometimes led to substantial runtime improvements. When no or few input sequences were irrelevant, pruning led to only a small overhead in runtime.
(7) The two-pass approach had similar or better performance than the one-pass approach for all subsequence constraints.
(8) DESQ was competitive (up to 1.7× slower) to the state-of-the-art specialized sequence miner prefix-growth for traditional subsequence constraints.
(9) DESQ consistently outperformed (up to 4× faster) RE-constrained FSM miner SMA. For RE constraints on all output subsequences, DESQ was competitive (up to 2.5× slower) to the state-of-art FSM constrained miner PPICt and for RE constraints on contiguous subsequences, DESQ outperformed (up to 10× faster) PPICt.

Our results indicate that DESQ is a suitable general-purpose system for a wide range of subsequence constraints.

7.1 Experimental Setup

Datasets. We used three real-world datasets: The *New York Times* corpus (NYT)[13] for text mining, the Amazon product review dataset (AMZN)[14] for mining product sequences, and the protein dataset (PRT)[15] for mining protein sequences. Key statistics of these datasets are summarized in Table 3.

The NYT dataset consists of roughly 50M sentences from 1.8M news articles published during 1987 and 2007. We treat each sentence as an input sequence and each word (token) as an item. We generated an item hierarchy using annotations from the Stanford CoreNLP tools.[16] The NYT hierarchy consists of named entities, which generalize to their type (PERSON, ORGANIZATION, and LOCATION) and then to ENTITY, and of words, which generalize to their lemma and then to their part-of-speech tag. For example, "Maradona"⇒PERSON⇒ENTITY and "is"⇒"be"⇒VERB.

The AMZN dataset consists more than 82M product reviews from over 21M users. We extracted sequences of products (ordered by review timestamps) for each user. We used the Amazon product

[13] https://catalog.ldc.upenn.edu/LDC2008T19.
[14] snap.stanford.edu/data/web-Amazon.html.
[15] http://www-kdd.isti.cnr.it/SMA/.
[16] http://nlp.stanford.edu/software/corenlp.shtml.

Table 3. Statistics of the Datasets Used in Our Experimental Study

		NYT	AMZN	PRT
Sequence database	# Sequences	49,593,066	21,176,522	103,120
	Avg. length	20.15	3.90	482
	Max. length	15,009	44,557	600
	Total items	997,559,483	82,677,131	49,729,890
	Distinct items	7,155,771	9,874,211	25
Hierarchy	Total items	9,792,609	10,557,785	103,120
	Leaf items	7,155,769	10,528,545	103,120
	Interm. items	2,636,817	29,155	0
	Root items	23	85	0
	Max depth	3	10	1
	Avg. fan-out	3.71	482	0
	Max. fan-out	2,832,744	1,940,285	0
	Avg. fan-in	1.0	1	0
	Max. fan-in	1	58	0

hierarchy as our item hierarchy. For example, "Canon 5D" ⇒"Digital Cameras"⇒"Camera & Photo"⇒ "Electronics."

The PRT dataset consists of over 100,000 amino acid sequences where each sequence is composed from 25 amino acid codes (items). The hierarchy is flat, i.e., there are no generalizations.

Pattern expressions. We created a set of pattern expressions, which express tasks in information extraction (IE), natural language processing (NLP), customer behavior mining, and protein sequence mining. Our pattern expressions are shown in Table 4 and fall into four categories. The first category (N_1–N_5) expresses relevant patterns useful for IE and NLP applications and were inspired by [21, 37, 48] and Google's *n*-grams[17]; these expressions were used on the NYT dataset. The second category (A_1–A_4) expresses patterns relevant to market-basket analysis and apply to AMZN. Expressions from the third category (P_1–P_4) are used to mine protein sequences that match a protein motif (described by regular expressions) from the PRT dataset; the regular expression constraints were taken from the PROSITE database.[18] The dataset and expressions P_1 and P_3 were used by Trasarti et al. [47] to evaluate the SMA algorithm for RE-constrained FSM. The PRT dataset was also used by Aoga et al. [5, 6] to evaluate the PPIC algorithms for FSM constraints. The fourth category (T_1–T_3) models traditional subsequence constraints commonly studied in the literature [10, 33, 39, 40, 43, 50, 51]. Note that these expressions are parameterized (see Section 7.6). We used the NYT with these expressions.

Implementation and setup. We implemented DESQ in Java (JDK 1.8).[19] We used ANTLR4[20] to generate a parser for pattern expressions. The sFST is constructed from the resulting parse tree and subsequently minimized (as described in Section 6.1). For all pattern expressions E in Table 4, we construct an sFST for $.^*E.^*$, i.e., we allow partial matching as discussed in Section 6.2. We preprocessed the datasets to compute the f-list and assign integer identifiers to each item. Item identifiers were assigned in descending order of item frequency, thus a more frequent item received a smaller item identifier. In our implementations, we encoded the sequence database compactly as

[17] https://books.google.com/ngrams/.

[18] http://prosite.expasy.org/.

[19] The source code is publicly available at https://www.uni-mannheim.de/dws/research/resources/desq.

[20] http://www.antlr.org/.

Table 4. Example Pattern Expressions for IE and NLP Applications (N_1–N_5), Customer Behavior Mining Applications (A_1–A_4), Protein Sequence Mining (P_1–P_4), and Traditional Sequence Mining (T_1–T_3)

Pattern expression	Description	Example patterns from NYT dataset (frequency)
N_1: ENTITY (VERB$^+$ NOUN$^+$? PREP?) ENTITY	*Relational phrase b/w entities*	was advised by (118), is president of (388)
N_2: (ENTITY$^\uparrow$ VERB$^+$ NOUN$^+$? PREP? ENTITY$^\uparrow$)	*Typed relational phrases*	ORG owned by ENTITY (1280), PER born in LOC (1195)
N_3: (ENTITY$^\uparrow$ be$^{\uparrow}_{=}$) DET? (ADV? ADJ? NOUN)	*Copular relation for an entity*	PER be novelist (326), LOC be great place (107),
N_4: $(.^\uparrow)\{3\}$ NOUN	*Generalized 3-grams before a noun*	NOUN PREP DET (9854820), DET ADV ADJ (822377)
N_5: $([.^\uparrow\ .\ .] \mid [.\ .^\uparrow\ .] \mid [.\ .\ .^\uparrow])$	*Generalized 3-grams, where at most one item is generalized*	a ADJ house (12972), degree from ENTITY (12748)
		Example patterns from AMZN dataset (frequency)
A_1: (Electr$^\uparrow$)[.{0,2}(Electr$^\uparrow$)]{1,4}	*Generalized sequences of (up to 5) electronic items, which are at most 2 items apart in the input sequences*	"Mice", "Keyboards", "Computers & Accessories" (875), "Tablets", "Cases & Sleeves" (8507)
A_2: (Book)[.{0,2}(Book)]{1,4}	*Sequences of books*	"The Bourne Supremacy", "The Bourne Ultimatum" (16)
A_3: DigitalCamera[.{0,3}(.$^\uparrow$)]{1,4}	*Type of products bought after a digital camera*	"Lenses", "Tripods" (158), "Batteries", "SD Cards" (149)
A_4: (MInstr$^\uparrow$)[.{0,2}(MInstr$^\uparrow$)]{1,4}	*Generalized sequences of musical instruments*	"Guitars", "Bags & Cases", "Instr. Accessories" (170)
		Example patterns from PRT dataset (frequency)
P_1: ([S$\mid$T]) .*(.) .*([R$\mid$K])	*subsequences that match RE≡[S$\mid$T] . [R$\mid$K]*	S L R (103093), T A K (102941), S A K (102946)
P_2: ([I$\mid$V]) .*(D) .*(L) .*(G) .*(T) .*([S$\mid$T]) .*(.) .*([S$\mid$C])	*subsequences that match RE≡[I$\mid$V] D L G T [S$\mid$T] .[S$\mid$C]*	I D L G T T L S (102975), V D L G T S T C (92662) V D L G T S D S (102901)
P_3: ([S$\mid$T] . [R$\mid$K])	*contiguous subsequences that match RE≡[S$\mid$T] . [R$\mid$K]*	S L R (14995), T A K (8840), S A K (10397)
P_4: ([S$\mid$T] . . [D$\mid$E])	*contiguous subsequences that match RE≡[S$\mid$T] . . [D$\mid$E]*	S D L E (2015), T L E E (2329), S G L D (1054)
		Example patterns from NYT dataset (frequency)
T_1: $(.)\{1,\lambda\}$	*n-grams of up to λ words*	green tea (993), editor in chief (7431)
T_2: $(.)[.\{0,\gamma\}(.)]\{1,\lambda-1\}$	*Skip n-grams with gap at most γ words and of up to length λ*	flight from to (2038), son of and of (37605)
T_3: $(.^\uparrow)\{1,\lambda\}$	*Generalized n-grams of up to λ words*	NOUN PREP DET NOUN (9854820), PERSON be NOUN (40669)

To increase readability, we omit a leading and trailing ".*" from each pattern expression.

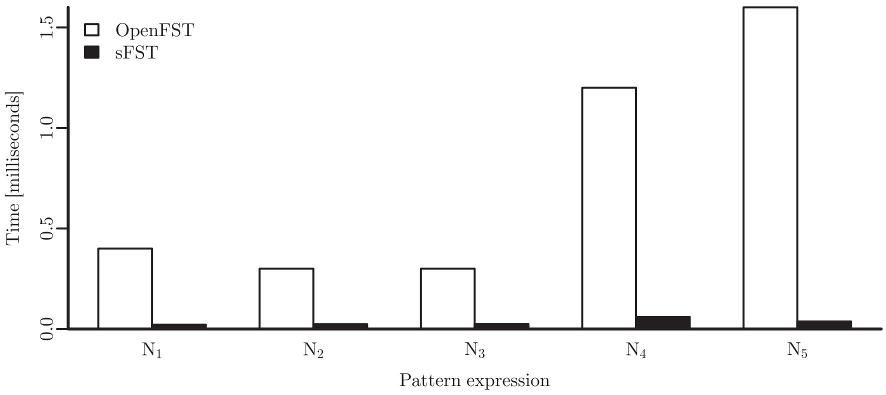


Fig. 10. Average simulation time per input sequence from a random sample of 10,000 input sequences.

arrays of item identifiers and use variable-length byte encoding to compress projected databases. Unless specified otherwise, DESQ-COUNT and DESQ-DFS refer to the basic one-pass approach of Sections 5.2 and 5.3, respectively.

To evaluate sFSTs, we compared it against the state-of-the-art FST library OpenFST 1.6.3.[21] To measure the overhead of DESQ for common subsequence constraints, we compared it with various state-of-the-art methods. For length and gap constraints, we used (1) the Scala implementation of PPICt [5, 6] available from the authors,[22] (2) the C++ implementation of cSPADE [50] from the author, (3) our implementation of SPADE in Java that additionally handles hierarchy constraints, and (4) our implementation of prefix-growth [40] in Java. For RE constraints, we used PPICt, prefix-growth, and a C++ executable of SMA [47] obtained from the authors.

Experiments on the NYT and AMZN datasets 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 i7-7560U 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. All experiments were run single-threaded and with the same JVM memory budget (120GB for NYT and AMZN, 10GB for PRT).

Methodology. For each experiment, we report the performance in terms of the total wall-clock time between launching the mining task and receiving the final result (excluding I/O to and from disk). All measurements were averaged over three independent runs. Unless stated otherwise, all methods produced the same results.

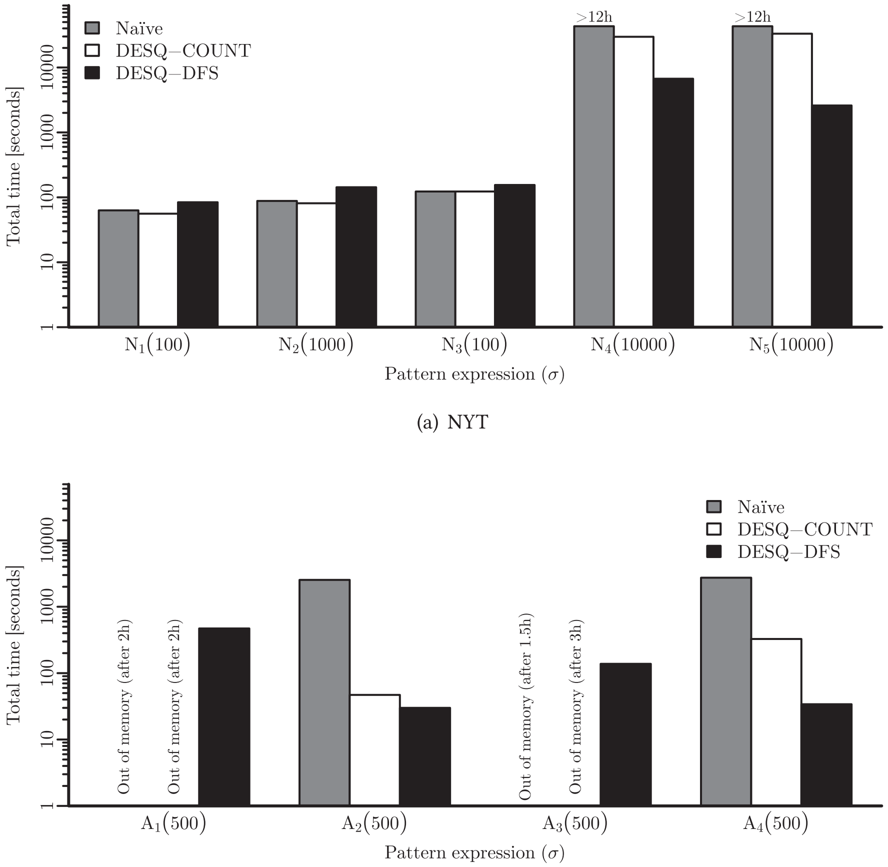
7.2 Comparison of sFST Simulation and OpenFST

We first evaluated the effectiveness of our FST optimizations. We compared sFST simulation using Algorithm 1 with uncompressed FST simulation using a state-of-the-art FST library OpenFST on pattern expressions N_1–N_5. The results are shown in Figure 10 in which we report the average simulation time per input sequence to generate all P-subsequences (excluding time to construct the FST) from a random sample of 10000 input sequences from the NYT dataset.

We observed that sFST simulation was 10–40× faster than uncompressed FST simulation using OpenFST. This is because pattern expressions often translate to excessively large FSTs, which are inefficient to simulate (see Table 2 and the discussion on sFSTs in Section 4.3). Moreover, OpenFST

[21] http://www.openfst.org.

[22] https://sites.uclouvain.be/cp4dm/spm/ppict/.



(a) NYT

(b) AMZN

Fig. 11. Performance of DESQ mining algorithms on (a) NYT and (b) AMZN datasets.

cannot directly handle hierarchies so that uncompressed FSTs get large. Finally, as discussed in Section 6.1, many of our pattern expressions cannot be determinized, which curtails the classical FST optimizations supported by OpenFST.

Overall, we conclude that our FST optimizations were effective.

7.3 Comparison of Naïve, DESQ-COUNT, and DESQ-DFS

In our next set of experiments, we evaluated the performance of Naïve, DESQ-COUNT and DESQ-DFS on pattern expressions N_1–N_5 and A_1–A_4. The results are shown in Figures 11(a) and 11(b) using log-scale for NYT and AMZN datasets, respectively. For each pattern expression, we empirically chose the minimum support threshold σ.

We first discuss the results on NYT shown in Figure 11(a). For pattern expressions N_1–N_3, Naïve, DESQ-COUNT and DESQ-DFS had similar performance and finished in under a couple of minutes. For N_4 and N_5, however, runtimes were higher and DESQ-DFS was significantly faster than Naïve (more than 16×) and DESQ-COUNT (up to 13×).

To gain insight into these results, we computed the average number $\mu_{\text{Naïve}}$ and μ_{Count} of P-generated sequences per input sequence for Naïve (average of $|G_P(T)|$) and DESQ-COUNT (of

Table 5. sFST Simulation Statistics

Pat. Expr. (σ)	$\mu_{\text{Naïve}}$	μ_{Count}	$\mathcal{A}$-rel. inputs	η_{Pruning}	$\eta_{\text{Two-pass}}$
$N_1(100)$	1.04	1.04	1.9%	2.7%	0.3%
$N_2(1000)$	9.37	6.61	1.8%	3.2%	0.7%
$N_3(100)$	2.02	1.71	0.9%	1.3%	0.2%
$N_4(1000)$	133.48	105.08	89.6%	97.9%	48.5%
$N_5(1000)$	130.76	93.09	97.2%	99.9%	73.1%
$A_1(500)$	10,790.01	4,394.37	9.8%	98%	95.9%
$A_2(100)$	180.48	38.54	8.9%	49.4%	12.3%
$A_3(100)$	43,533.84	25,716.88	0.6%	84.4%	83.7%
$A_4(100)$	10,824.27	3,787.64	0.9%	59.8%	55.2%

$|G_P^F(T)|$), respectively.[23] These numbers are shown in Table 5 (second and third columns) for each pattern expression. From these statistics, we observed that μ_{Count} is always less than or equal to $\mu_{\text{Naïve}}$ as asserted by Lemma 5.1, and thus DESQ-COUNT can significantly reduce the number of candidate sequences that are generated and counted. In what follows, we discuss how these statistics relate to different runtimes of DESQ's mining algorithms.

We observed that for small values of μ, Naïve, 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 Naïve and DESQ-COUNT is expected to work well, because few sequences are generated and the advanced pruning methods of DESQ-DFS are not needed. When μ is large, however, both Naïve and DESQ-COUNT can enumerate many sequences that turn out to be infrequent, which is expensive. DESQ-DFS prunes many of these sequences early on and is thus more efficient.

On the AMZN dataset (expressions A_1–A_4; Figure 11(b)), DESQ-DFS consistently outperformed Naïve and DESQ-COUNT. For A_1 and A_3, the large number of candidate P-sequences (cf. Table 5) led to a memory overflow for both Naïve and DESQ-COUNT. However, DESQ-DFS finished in few hundred seconds, benefiting from its advanced pruning techniques. For A_2, DESQ-COUNT and DESQ-DFS had similar performance (with DESQ-DFS being faster), whereas for A_4, DESQ-DFS was more than an order of magnitude faster than DESQ-COUNT. This behavior is explained by the observation that μ was large for all pattern expressions.

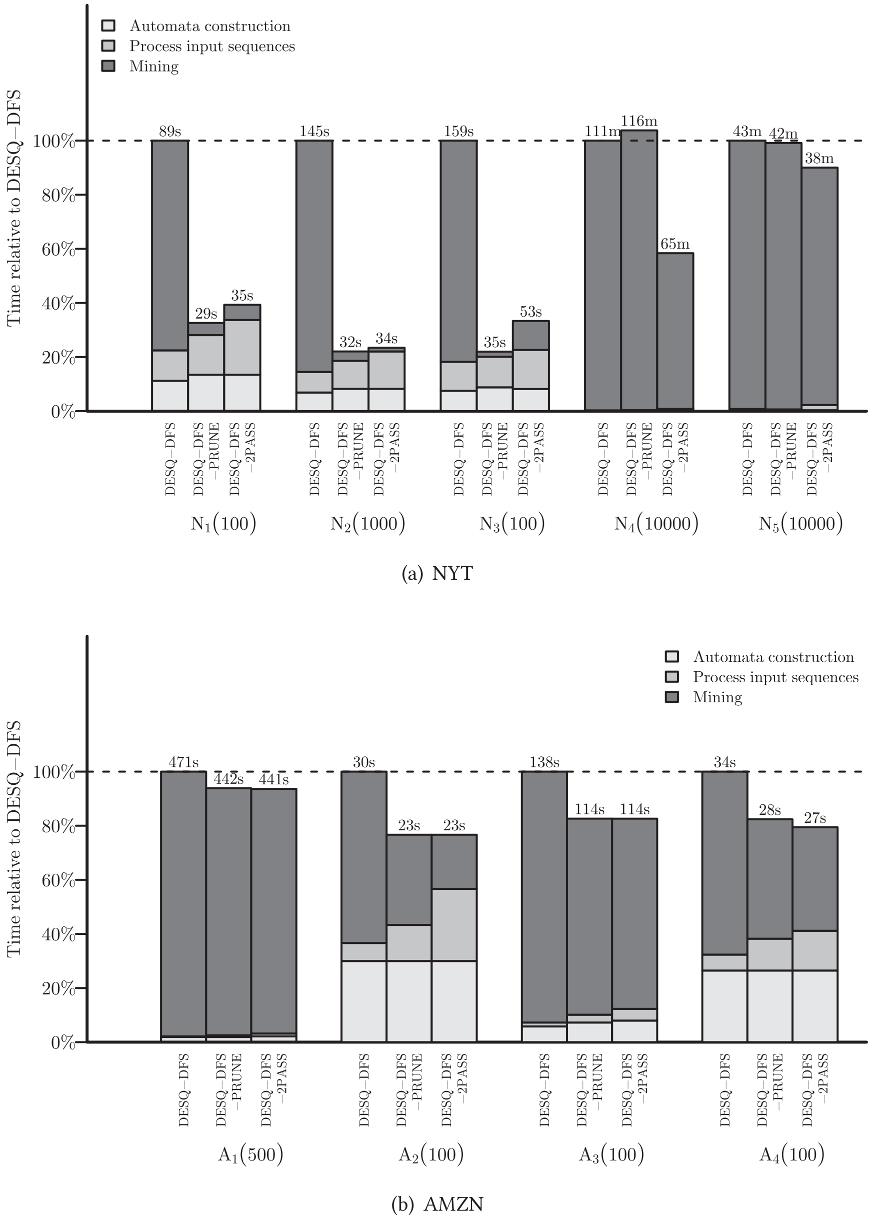
We conclude that DESQ-DFS consistently worked well in our experiments. Although DESQ-COUNT was slightly faster in some simpler cases, its performance substantially fell behind DESQ-DFS for more difficult ones. Thus, we consider it generally safer to use DESQ-DFS in practice.

7.4 Effectiveness of Optimizations

We now study the effectiveness of pruning $\mathcal{A}$-irrelevant inputs (Section 6.3) and of the two-pass approach (Section 6.4) when integrated with DESQ-DFS. We refer to these methods as DESQ-DFS-PRUNE and DESQ-DFS-2PASS, respectively.

The results are shown in Figures 12(a) and 12(b) for NYT and AMZN datasets, respectively. For ease of comparison, we normalize runtime DESQ-DFS time to 100% and show relative times for DESQ-DFS-PRUNE and DESQ-DFS-2PASS. The absolute (total) runtime of each method is shown on top of each bar. For each experiment, we split the total mining time into:

[23] We only considered $\mathcal{A}$-relevant input sequences.



(a) NYT

(b) AMZN

Fig. 12. Effectiveness of pruning irrelevant inputs and the two-pass approach on the (a) NYT and (b) AMZN datasets. The number on top of each bar denotes the absolute runtime of the corresponding method.

(i) *Automata construction*, which is the time required for parsing the pattern expression and constructing the resulting sFST. With pruning and the two-pass, the time required to construct the DFA and reverse-DFA, respectively, is also included.

(ii) *Process input sequences*, which is the time to read input sequences (from memory) and construct the initial projected database. With pruning, this additionally includes time to check each input sequence against the DFA for relevance. For the two-pass approach, it additionally includes the time for the first pass (in which we compute the reachable states in the reverse DFA).

(iii) *Mining*, which is the remaining time required produce the final output (mainly exploration of the search space via expansions).

Overall, we observed that both techniques significantly improved the performance of DESQ-DFS. In particular, the performance improvements stem from mining, which was up to 40× faster with DESQ-DFS-PRUNE and up to 60× faster with DESQ-DFS-2PASS. Both methods, however, incurred a small overhead in automata construction and processing input sequences. For automata construction, DESQ-DFS-PRUNE and DESQ-DFS-2PASS additionally require to construct a DFA and reverse DFA, respectively. This overhead was negligible (up to 2s) for all pattern expressions. The time to process input sequences increased by up to 1.3× with DESQ-DFS-PRUNE. This is expected as it additionally simulates the DFA for checking relevance. This increase in runtime is even more pronounced in DESQ-DFS-2PASS as it additionally computes and stores the reachable states in the reverse DFA. However, this overhead of processing input sequences is amortized by the mining time thus making our optimizations effective. As seen in the Figures 12(a) and 12(b), the effectiveness of these techniques vary depending on the pattern expression.

To gain further insights, we computed for each expression the percentage of $\mathcal{A}$-relevant inputs and the fraction η of sFST transitions taken by DESQ-DFS-PRUNE and DESQ-DFS-2PASS w.r.t.DESQ-DFS without these optimization. These numbers are shown in Table 5; columns 4–6. We will refer to these numbers in what follows.

Effectiveness of Pruning $\mathcal{A}$-irrelevant Inputs. We first discuss the effectiveness of DESQ-DFS-PRUNE. On the NYT dataset (Figure 12(a)), we observed an overall speedup of up to 4.5× for expressions N_1, N_2, and N_3, for which only a small fraction (<2%, cf. Table 5) of input sequences were $\mathcal{A}$-relevant. Thus the overhead of additionally constructing and simulating the DFA pays off during mining, since pruning is very effective. For expressions N_4 and N_5, however, a large fraction (90% and 97% resp.,) of input sequences were $\mathcal{A}$-relevant and thus DESQ-DFS-PRUNE did not offer benefits. On the AMZN dataset, expressions A_1–A_4, DESQ-DFS-PRUNE consistently performed well as most of the input sequences turn out be $\mathcal{A}$-irrelevant. Overall, pruning of $\mathcal{A}$-irrelevant input sequences can lead to substantial runtime improvements. When no or few input sequences were irrelevant, pruning led to only a small overhead in runtime.

Effectiveness of the Two-pass Approach. We now turn attention to DESQ-DFS-2PASS. On NYT dataset, for pattern expressions N_1, N_2, and N_3, DESQ-DFS-2PASS was up to 4× faster than DESQ-DFS. But it was however slightly slower (by up to 1.2×) than DESQ-DFS-PRUNE. Although, when compared to DESQ-DFS-PRUNE, DESQ-DFS-2PASS computes a much smaller fraction of transitions as a result of avoiding unnecessary backtracking (cf. η_{Pruning} and $\eta_{\text{Two-pass}}$ in Table 5), it offers limited additional benefit in comparison to DESQ-DFS-PRUNE, because only a small fraction of total input sequences turn out be $\mathcal{A}$-relevant. However, for expression N_4, where a large fraction of input sequences were $\mathcal{A}$-relevant, DESQ-DFS-2PASS offered a speed-up of up to 2× during mining, which stems from avoiding unnecessary backtracking. This is also supported by the statistics in Table 5, which shows that DESQ-DFS-2PASS computes less than 50% of transitions compared to 97% computed by DESQ-DFS-PRUNE. Pattern expression N_5 is a notable case, since the expression is composed of only wild cards, which results in a DFA that accepts every sequence of length at least 3 and thus more than 99% of the input sequences turn out to be $\mathcal{A}$-relevant. Compared to DESQ-DFS-PRUNE, which did not offer any significant benefits, DESQ-DFS-2PASS resulted in a 10% speedup as it was able to avoid 30% of the transitions. Although expression N_5 is composed of only wild cards, DESQ-DFS-2PASS benefits from leveraging final-complete

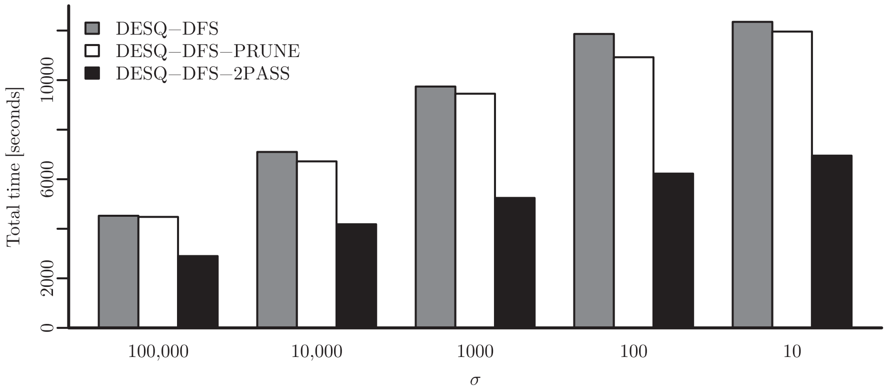


Fig. 13. Impact of minimum support threshold σ on optimizations for pattern expression N_4.

annotations to avoid processing both prefixes and suffixes that produce only infrequent items. On AMZN, for expressions A_1–A_4, DESQ-DFS-2PASS consistently outperformed both DESQ-DFS and DESQ-DFS-PRUNE as most of the input sequences were $\mathcal{A}$-irrelevant and also because it computes much smaller fraction of transitions.

Overall, we found that the effectiveness of our optimizations depend on the pattern expression and the data. We generally consider using DESQ-DFS-2PASS to be the best overall option: its performance was either similar or better than DESQ-DFS in all our experiments. This is supported by the theoretical evidence: a guarantee such as the one of Theorem 6.1 cannot be given for DESQ-DFS and DESQ-DFS-PRUNE.

7.5 Impact of Minimum Support Threshold

We also investigated to what extent the performance and effectiveness of pruning $\mathcal{A}$-irrelevant inputs and the two-pass approach is affected by the minimum support threshold (σ). We use pattern expression N_4 on the NYT dataset as it is relatively complex (i.e., high number of $\mathcal{A}$-relevant inputs and high μ-value) and varied σ from 100,000 down to 10. The results are shown in Figure 13.

We observed that for high values of σ (e.g., $\sigma = 100{,}000$), DESQ-DFS-PRUNE had a similar performance as DESQ-DFS. This is because at very high support thresholds, few items (0.02%) are frequent and DESQ-DFS avoids transitions that output infrequent items (see Lemma 5.1). Therefore, the benefit of pruning decreases with increasing minimum support threshold. We observed a similar behavior for the two-pass approach. In particular, for $\sigma = 1000{,}00$, DESQ-DFS-2PASS was 1.5× faster than DESQ-DFS; for $\sigma = 10$, it was 2× faster.

7.6 Performance with Traditional Subsequence Constraints

Next, we investigated the overhead of DESQ compared to specialized miners cSPADE [50] and prefix-growth [40] (based on PrefixSpan [39]) for traditional subsequence constraints. We also compared against the CP-based FSM miner PPICt. In particular, we considered length and gap constraints as well as item hierarchies. We map these constraints to pattern expressions and obtain T_1–T_3 of Table 4. The expressions are parameterized by maximum-length parameter λ and/or maximum-gap parameter γ. We used the NYT dataset and ran FSM for different configurations of increasing difficulty w.r.t. output size. The results are shown in Figure 14 using log-scale.

For length and gap constraints (first four groups), PPICt terminated with an out-of-memory exception. PPICt does not support hierarchies, so we exclude it from the final experiment (fifth group). We observed that for all configurations, cSPADE was significantly slower than both

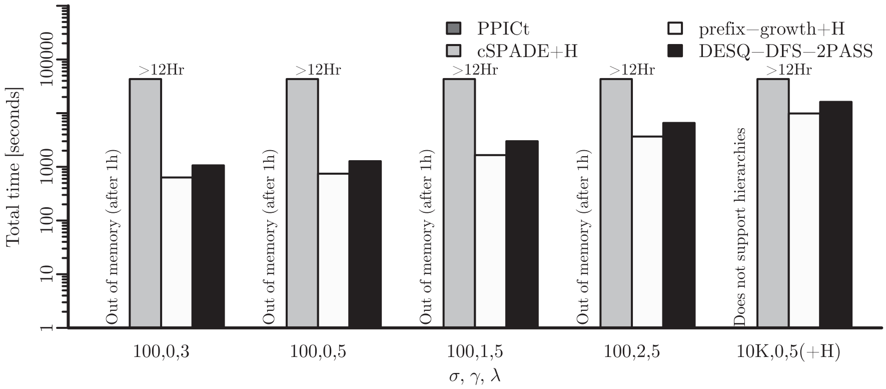


Fig. 14. Performance for traditional subsequence constraints.

prefix-growth and DESQ-DFS-2PASS. This is because cSPADE follows a candidate-generation-and-test approach and suffered from an excessive number of generated candidates [39]. To keep our study manageable, we stopped cSAPDE after 12h. Compared to prefix-growth, DESQ-DFS-2PASS was up to 1.5× slower for n-grams (first two groups). For skip grams (third and fourth group), DESQ-DFS-2PASS was up to 1.7× slower than prefix-growth. The overhead was slightly more pronounced, because pattern expressions for gap constraints have uncaptured wildcards (cf. T_2 in Table 4), which results in non-determinism and increases the amount of (acceptable) backtracking during sFST simulation. For generalized n-grams (last group), where we additionally considered item hierarchies, DESQ-DFS-2PASS was again up to 1.7× slower than prefix-growth as the amount of backtracking performed by DESQ increased with the depth of hierarchy (cf. line 26 of Algorithm 2 and the discussion in Section 4.3).

Overall, our experiments indicate that DESQ is competitive with prefix-growth. Although prefix-growth was indeed faster for FSM with traditional subsequence constraints, the overhead of DESQ was acceptable.

7.7 Performance with RE Constraints

In our final set of experiments, we evaluated the efficiency of DESQ for mining frequent subsequences (all or contiguous) that match a given regular expression. Our pattern expressions allow us to express REs with their equivalent pattern expressions (cf. Table 1 and expressions P_1–P_4 of Table 4). We compared DESQ's performance against state-of-the-art RE-constrained FSM methods, namely SMA [47], prefix-growth [40], and CP-based PPICt [5, 6]. We used DESQ-DFS-2PASS on the PRT dataset for which we obtained suitable RE constraints from the PROSITE database[24]; the runtimes are shown in log-scale in Figure 15.

We observed that DESQ-DFS-2PASS was up to 2–4× faster than SMA, up to 3–6× faster than prefix-growth, and up to 2–2.5× slower than PPICt for P_1 and P_2, respectively. We do not give SMA results for P_3 and P_4, because the implementation produced incorrect results (acknowledged in private communication by the original authors). We did not investigate this further as the SMA source is not available. Both DESQ-DFS-2PASS and prefix-growth finished in few seconds on P_3 and P_4, and were up to 10× faster than PPICt with DESQ-DFS-2PASS being slightly faster.

Our results indicate that DESQ is a suitable method for RE-constrained FSM as well.

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

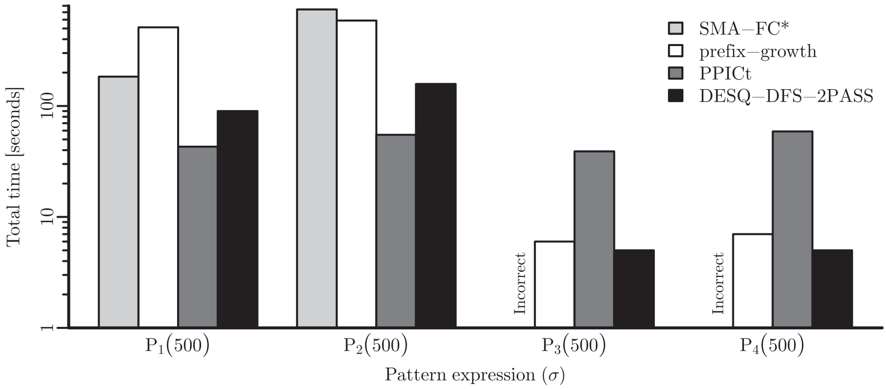


Fig. 15. Performance for RE-constrained FSM.

8 RELATED WORK

We now relate ideas put forward in this article to existing prior work, which can be coarsely categorized into:

Sequential pattern mining. The problem of mining frequent sequential patterns was introduced by Agrawal and Srikant [1]. Their Apriori algorithm follows a candidate-generation-and-test approach to identify sequential patterns that are frequent in the database. The subsequent GSP algorithm [43] exploits the antimonotonicity property of sequential patterns to efficiently generate and prune candidate sequences. SPADE by Zaki [51] also generates and prunes candidates, but it operates on an inverted index structure representation of the database. Pei et al. [39] proposed the PrefixSpan algorithm, which is based on a more efficient pattern-growth approach that recursively grows frequent prefixes using database projections. DESQ-DFS can be seen as a generalization of PrefixSpan to support arbitrary pattern expressions. SPAM [8], which is similar to SPADE, uses an internal bitmap structure for database representation and employs a pattern-growth approach to mine frequent sequential patterns. A comprehensive discussion of these methods is given in Reference [31].

Subsequence constraints. There are many extensions to the basic sequential pattern mining framework for supporting subsequence constraints. GSP [43] and LASH [10], for example, allow gap constraints and incorporate item hierarchies. cSPADE [50] handles length, gap and item constraints. Wu et al. [49] consider subsequences with periodic wild card gaps, i.e., subsequences where consecutive items are separated by the same gap in the input. Garofalakis et al. [23] introduced regular expression (RE) constraints that subsequences need to satisfy. The proposed SPIRIT algorithms translate a given RE into a deterministic finite-state automata and adapts GSP-like candidate-generation-and-test approach to mine frequent sequential patterns. Along these lines, Albert-Lorincz and Boulicaut [2] proposed RE-Hackle algorithm, which represents RE via a tree structure. Pei et al. [40] advocate the prefix-growth method—which we also compare to in our experimental study—to handle RE as well as length and gap constraints. RE constraints have also been studied by Trasarti et al. [47]. They proposed the SMA algorithm, which uses Petri nets to match an RE. In contrast to DESQ, the above methods are less general, because they consider regular expressions on the output sequence only and do not support capture groups. Some of the above constraints (e.g., gap constraints), however, target the input sequence, whereas others (e.g., length constraints, RE constraints) target subsequences. Our pattern expressions unify both targets and allows us to express all of the above subsequence constraints (e.g., see Table 1).

More recently, constraint programming (CP) methods have been applied to support various subsequence constraints in sequential pattern mining. In particular, Negrevergne and Guns [38] modeled sequence mining with length, gap, item, and RE-constraints as a constraint satisfaction problem, which can be solved using CP. Their approach is not limited to frequent sequence mining but also supports other constraints on the multiset of mined patterns (e.g., maximality and closedness). Such pattern-set constraints are currently not supported by DESQ. Kemmar et al. [25, 26] proposed prefix-based projection techniques to efficiently handle constraints in CP-based approaches. Aoga et al. [5, 6] proposed the PPIC and PPICt algorithms, which outperformed other CP-based algorithms in their experimental study. Like DESQ , their approach allows to mix traditional constraints like length, item, and gap/span with RE constraints on the output sequences. DESQ additionally supports hierarchy constraints and context constraints (via REs with capture groups), which allows DESQ to express many customized subsequence constraints that arise in FSM applications (e.g., see Table 4).

Finally, this article is an extended version of [11], which originally proposed the DESQ system. Here, we (1) provide a more accessible and more detailed exposition, (2) include various proofs of correctness, (3) propose multiple optimizations to extend DESQ, and (4) performed an extended experimental study. Our optimizations include methods to partially determinize and minimize sFSTs, to use early-abort during sFST simulation whenever possible and without affecting correctness, for pruning irrelevant input sequences, and for avoiding unnecessary backtracking via the two-pass approach. Our experimental study suggests that our optimizations can substantially improve performance when compared to the basic DESQ system.

Pattern matching. Systems and languages for pattern matching over sequences have been extensively studied in literature and are related to our work. For example, SystemT's AQL language [22, 27] provides an SQL-like syntax to specify and extract pattern matches from text documents. Languages based on cascaded grammars such as CPSL [7] are also used in many information extraction engines. Christ [16] proposed a Corpus Query Language (CQL) based on regular expressions for searching pattern matches in text corpora. Pattern matching is also crucial for complex event processing tasks [18, 19], which aim to detect pattern matches in (live or archived) event sequences. In contrast to these systems, we propose a language and system for *frequent sequence mining*. This means that we are not interested in finding all *matches* of a specified pattern expression as in pattern matching, but are instead looking for frequent sequential *patterns* themselves. Our pattern expressions are in some sense simpler than most pattern matching languages, yet expressive enough to specify many subsequence constraints that arise in sequence mining applications. Nevertheless, 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. An interesting direction for future work is to investigate to what extent such integration is possible for more powerful pattern languages.

Finite state transducers. Finite state transducers (FST, [34, 36]) have been applied in areas such as speech recognition, machine translation, information extraction, and data mining. In DESQ, we make use of FSTs as a computational model for pattern matching and mining. In contrast to existing work on FSTs, our FSTs are often neither sequential nor p-subsequential [35] so that many existing optimization methods do not apply (e.g., minimization and determinization). We provide methods to extend, compress, and optimize our special FSTs to effectively handle pattern mining tasks and large hierarchies. Although traditional FST libraries such as OpenFST [3] can also be used within DESQ, our succinct FSTs support more efficient matching and mining (see Sections 4.3 and 7.2).

9 CONCLUSIONS

In this article, we introduced subsequence predicates as a general model for unifying and extending subsequence constraints for frequent sequence mining. We proposed pattern expressions as a simple, intuitive way to express subsequence constraints, suggested succinct finite-state transducers as an underlying computation model, and proposed the DESQ-COUNT and DESQ-DFS algorithms for efficient mining. We discussed various optimizations that improve simulation efficiency of our succinct finite-state transducers. Our experiments indicate that DESQ is an efficient, general-purpose FSM framework for various subsequence constraints that arise in applications.

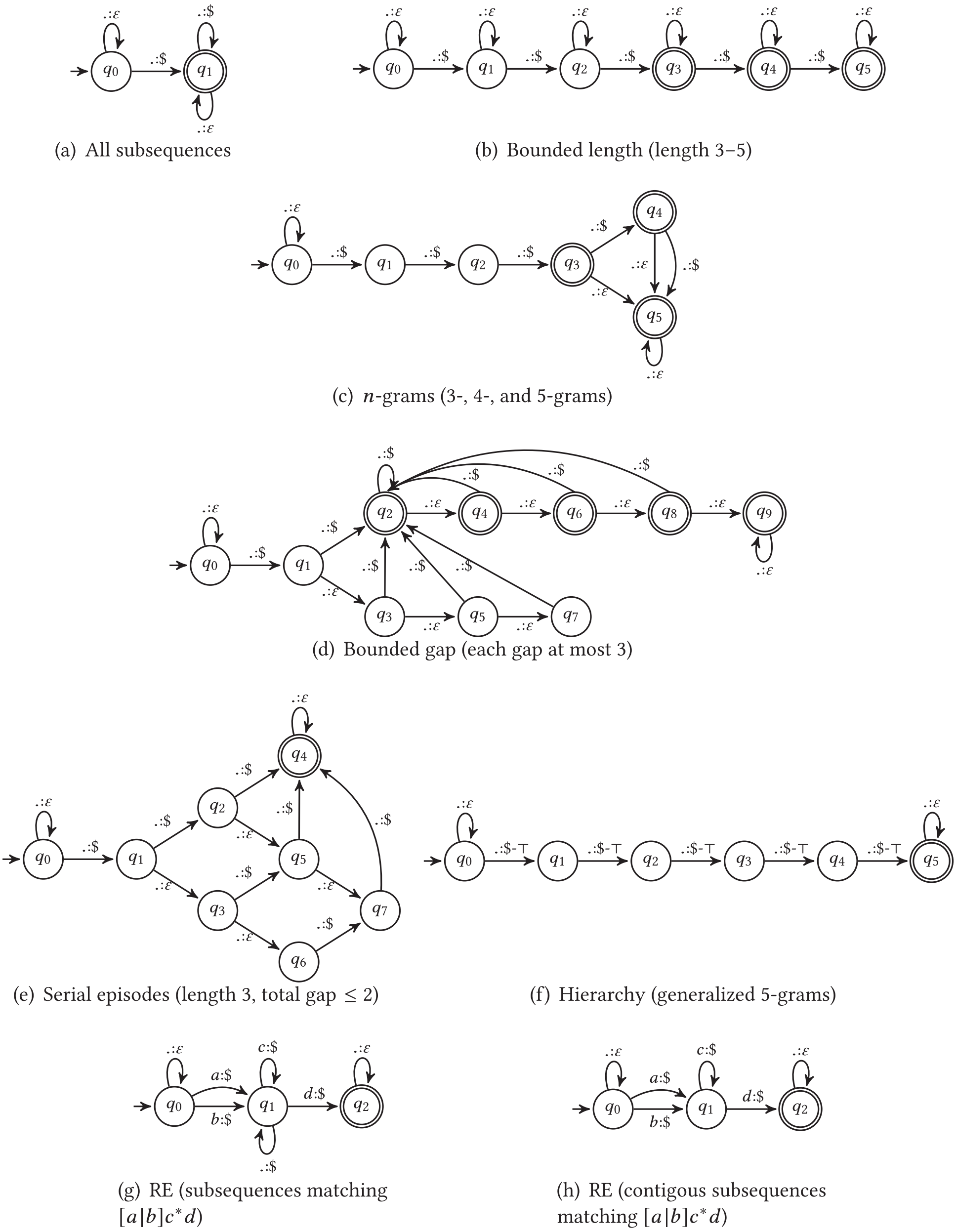
There are a number of directions for extending DESQ, both in terms of efficiency and scalabilty as well as in terms of functionality. In particular, DESQ's mining algorithms are sequential and cannot deal with massive amounts of data. Parallel and distributed mining algorithms that support flexible constraints are important future work. A first step was recently taken by Renz-Wieland et al. [42], who proposed distributed mining algorithms based on DESQ for platforms such as MapReduce and Spark. Another recent vein of work [20] investigates static FST analysis problems that ask if a given task can be distributed or not.

In this article, we focused solely on sequences of items, although some datasets are more naturally modeled as sequences of itemsets. Extending the pattern expression language as well as the mining algorithms to support itemsets is an interesting direction for future work. DESQ also limits its notion of interestingness to subsequence constraints and frequency; it neither supports set-based constraints, such as maximality and closedness, nor other notions of interestingness (such as utility), nor mining of partial orders. Adapting these notions in the context of flexible subsequence constraints is non-trivial, but forms an important next step towards general-purpose sequential pattern mining systems.

APPENDIX

A EXAMPLE SFSTS FOR PRIOR SUBSEQUENCE CONSTRAINTS

Figure 16 below gives sFSTs corresponding the pattern expressions of Table 1.



(a) All subsequences

(b) Bounded length (length 3–5)

(c) n-grams (3-, 4-, and 5-grams)

(d) Bounded gap (each gap at most 3)

(e) Serial episodes (length 3, total gap ≤ 2)

(f) Hierarchy (generalized 5-grams)

(g) RE (subsequences matching $[a|b]c^*d$)

(h) RE (contigous subsequences matching $[a|b]c^*d$)

Fig. 16. sFSTs, for example, pattern expressions of Table 1.

REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. 1995. Mining sequential patterns. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'95)*. 3–14.

[2] Hunor Albert-Lorincz and Jean-François Boulicaut. 2003. Mining frequent sequential patterns under regular expressions: A highly adaptative strategy for pushing constraints. In *SIAM International Conference on Data Mining (SDM'03)*. 316–320.

[3] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. OpenFst: A general and efficient weighted finite-state transducer library. In *Implementation and Application of Automata*. Springer. Vol. 4783, 11–23.

[4] Marco Almeida, Nelma Moreira, and Rogério Reis. 2007. *On the performance of automata minimization algorithms.* Technical report DCC-2007-03. Universidade do Porto.

[5] John O. R. Aoga, Tias Guns, and Pierre Schaus. 2016. An efficient algorithm for mining frequent sequence with constraint programming. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD'16).* 315–330.

[6] John O. R. Aoga, Tias Guns, and Pierre Schaus. 2017. Mining time-constrained sequential patterns with constraint programming. *Constraints* 22, 4 (2017), 548–570.

[7] Douglas E. Appelt and Boyan Onyshkevych. 1998. The common pattern specification language. In *Proceedings of the Text Retrieval Conference (TIPSTER'98).* 23–30.

[8] Jay Ayres, Jason Flannick, Johannes Gehrke, and Tomi Yiu. 2002. Sequential Pattern mining using a bitmap representation. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (SIGKDD'02).* 429–435.

[9] Kaustubh Beedkar, Klaus Berberich, Rainer Gemulla, and Iris Miliaraki. 2015. Closing the gap: Sequence mining at scale. *ACM Trans. Database Syst.* 40, 2, Article 8 (2015), 8:1–8:44 pages.

[10] Kaustubh Beedkar and Rainer Gemulla. 2015. LASH: Large-scale sequence mining with hierarchies. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'15).* 491–503.

[11] Kaustubh Beedkar and Rainer Gemulla. 2016. DESQ: Frequent sequence mining with subsequence constraints. In *Proceedings of the IEEE International Conference on Data Mining (ICDM'16).* 793–798.

[12] Klaus Berberich and Srikanta Bedathur. 2013. Computing N-gram statistics in MapReduce. In *Proceedings of the International Conference on Extending Database Technology (EDBT'13).* 101–112.

[13] Alvis Brazma, Inge Jonassen, Jaak Vilo, and Esko Ukkonen. 1998. Pattern discovery in biosequences. *Lecture Notes in Computer Science (LNCS'98)* 1433 (1998), 257–270.

[14] A. Brüggemann-Klein and D. Wood. 1998. One-unambiguous regular languages. *Info. Comput.* 142, 2 (1998), 182–206.

[15] J.A. Brzozowski. 1962. Canonical regular expressions and minimal state graphs for definite events. *Math. Theory Automata* 12 (1962), 529–561.

[16] Oliver Christ. 1994. A modular and flexible architecture for an integrated corpus query system. *CoRR* abs/cmp-lg/9408005 (1994).

[17] Wojciech Czerwinski, Claire David, Katja Losemann, and Wim Martens. 2017. Deciding definability by deterministic regular expressions. *J. Comput. Syst. Sci.* 88 (2017), 75–89.

[18] Alan Demers, Johannes Gehrke, and P. Biswanath. 2007. Cayuga: A general purpose event monitoring system. In *Proceedings Conference on Innovative Data Systems Research (CIDR'07).* 412–422.

[19] Nihal Dindar, Baris Güç, Patrick Lau, Asli Ozal, Merve Soner, and Nesime Tatbul. 2009. DejaVu: Declarative pattern matching over live and archived streams of events. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'09).* 1023–1026.

[20] Johannes Doleschal, Benny Kimelfeld, Wim Martens, Yoav Nahshon, and Frank Neven. 2019. Split-correctness in information extraction. In *Proceedings of the ACM SIGMOD Symposium on Principles of Database Systems (PODS'19).* To appear.

[21] Anthony Fader, Stephen Soderland, and Oren Etzioni. 2011. Identifying relations for open information extraction. In *Proceedings of the Conference on Empirical Methods on Natural Language Processing (EMNLP'11).* 1535–1545.

[22] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. 2015. Document spanners: A formal approach to information extraction. *J. ACM* 62, 2, Article 12 (May 2015).

[23] Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. 1999. SPIRIT: Sequential pattern mining with regular expression constraints. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'99).* 223–234.

[24] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2007. *Automata Theory, Languages, and Computation* (3rd ed.). Addison Wesley.

[25] Amina Kemmar, Yahia Lebbah, Samir Loudni, Patrice Boizumault, and Thierry Charnois. 2017. Prefix-projection global constraint and top-k approach for sequential pattern mining. *Constraints* 22, 2 (2017), 265–306.

[26] Amina Kemmar, Samir Loudni, Yahia Lebbah, Patrice Boizumault, and Thierry Charnois. 2016. A global constraint for mining sequential patterns with GAP constraint. In *Proceedings of the Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR'16).* 198–215.

[27] Rajasekar Krishnamurthy, Yunyao Li, Sriram Raghavan, Frederick Reiss, Shivakumar Vaithyanathan, and Huaiyu Zhu. 2009. SystemT: A system for declarative information extraction. *SIGMOD Rec.* 37, 4 (2009), 7–13.

[28] Yuri Lin, Jean-Baptiste Michel, Erez Lieberman Aiden, Jon Orwant, Will Brockman, and Slav Petrov. 2012. Syntactic annotations for the Google books ngram corpus. In *Proceedings of the ACL System Demonstrations (ACL'12).* 169–174.

[29] Adam Lopez. 2008. Statistical machine translation. *ACM Comput. Surv.* 40, 3 (2008), 8:1–8:49.

[30] Katja Losemann, Wim Martens, and Matthias Niewerth. 2016. Closure properties and descriptional complexity of deterministic regular expressions. *Theor. Comput. Sci.* 627 (2016), 54–70.

[31] Nizar R. Mabroukeh and C. I. Ezeife. 2010. A taxonomy of sequential pattern mining algorithms. *ACM Comput. Surv.* 43, 1, Article 3 (2010), 3:1–3:41 pages.

[32] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. 1997. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov.* 1, 3 (1997), 259–289.

[33] Iris Miliaraki, Klaus Berberich, Rainer Gemulla, and Spyros Zoupanos. 2013. Mind the gap: Large-scale frequent sequence mining. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'13)*. 797–808.

[34] Mehryar Mohri. 1997. Finite-state transducers in language and speech processing. *Comput. Linguist.* 23, 2 (June 1997), 269–311.

[35] Mehryar Mohri. 2000. Minimization algorithms for sequential transducers. *Theor. Comput. Sci.* 234, 1–2 (Mar. 2000), 177–201.

[36] Mehryar Mohri, Fernando Pereira, and Michael Riley. 2002. Weighted finite-state transducers in speech recognition. *Comput. Speech Lang.* 16, 1 (2002), 69–88.

[37] Ndapandula Nakashole, Gerhard Weikum, and Fabian Suchanek. 2012. PATTY: A taxonomy of relational patterns with semantic types. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CONLL'12)*. 1135–1145.

[38] Benjamin Negrevergne and Tias Guns. 2015. Constraint-based sequence mining using constraint programming. In *Proceedings of the Integration of AI and OR Techniques in Constraint Programming (CPAIOR'15)*. Springer International Publishing, 288–305.

[39] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. 2001. PrefixSpan: Mining sequential patterns by prefix-projected growth. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'01)*. 215–224.

[40] Jian Pei, Jiawei Han, and Wei Wang. 2002. Mining sequential patterns with constraints in large databases. In *Proceedings of the Conference on Information and Knowledge Management (CIKM'02)*. 18–25.

[41] M. O. Rabin and D. Scott. 1959. Finite automata and their decision problems. *IBM J. Res. Dev.* 3, 2 (1959), 114–125.

[42] Alexander Renz-Wieland, Matthias Bertsch, and Rainer Gemulla. 2019. Scalable frequent sequence mining with flexible subsequence constraints. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'19)*. 1490–1501.

[43] Ramakrishnan Srikant and Rakesh Agrawal. 1996. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of the International Conference on Extending Database Technology (EDBT'96)*. 3–17.

[44] Jaideep Srivastava, Robert Cooley, Mukund Deshpande, and Pang-Ning Tan. 2000. Web usage mining: Discovery and applications of usage patterns from web data. *SIGKDD Explor. Newslett.* 1, 2 (2000), 12–23.

[45] L. Stockmeyer and A. Meyer. 1973. Word problems requiring exponential time: Preliminary report. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC'73)*. 1–9.

[46] Ken Thompson. 1968. Programming techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (1968), 419–422.

[47] Roberto Trasarti, Francesco Bonchi, and Bart Goethals. 2008. Sequence mining automata: A new technique for mining frequent sequences under regular expressions. In *Proceedings of the IEEE International Conference on Data Mining (ICDM'08)*. 1061–1066.

[48] Immanuel Trummer, Alon Halevy, Hongrae Lee, Sunita Sarawagi, and Rahul Gupta. 2015. Mining subjective properties on the web. In *Proceedings of the ACM International Conference on Management of Data(SIGMOD'15)*. 1745–1760.

[49] Youxi Wu, Lingling Wang, Jiadong Ren, Wei Ding, and Xindong Wu. 2014. Mining sequential patterns with periodic wildcard gaps. *Appl. Intell.* 41, 1 (2014), 99–116.

[50] Mohammed J. Zaki. 2000. Sequence mining in categorical domains: Incorporating constraints. In *Proceedings of the Conference on Information and Knowledge Management (CIKM'00)*. 422–429.

[51] Mohammed J. Zaki. 2001. SPADE: An efficient algorithm for mining frequent sequences. *Mach. Learn.* 42, 1–2 (2001), 31–60.

Received April 2018; revised December 2018; accepted February 2019