Methods for Frequent Sequence Mining with Subsequence Constraints

Inauguraldissertation

zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von
Kaustubh Beedkar
geburtsort in Bhopal, Indien

Mannheim, 2016

Dekan:	Prof. Dr. Heinz Jürgen Müller, Universität Mannheim
Referent:	Prof. Dr. Rainer Gemulla, Universität Mannheim
Korreferent:	Prof. Dr. Simone Paolo Ponzetto, Universität Mannheim
Korreferent:	Prof. Dr. Gerhard Weikum, Max-Planck-Institut für Informatik

Tag der mündlichen Prüfung: 10 Februar 2017

Abstract

In this thesis, we study scalable and general purpose methods for mining frequent sequences that satisfy a given subsequence constraint. Frequent sequence mining is a fundamental task in data mining and has many real-life applications like information extraction, market-basket analysis, web usage mining, or session analysis. Depending on the underlying application, we are generally interested in discovering certain frequent sequences, which are described using subsequence constraints. There exists many tools and algorithms for this task, however, they are not sufficiently scalable to deal with large amounts of data that may arise in applications and are generally not extensible across range of applications.

We propose scalable, distributed sequence mining algorithms that target MapReduce. Our work builds on MG-FSM, which is a distributed framework for frequent sequence mining. We propose novel algorithms that improve and extend the basic MG-FSM framework to efficiently support traditional subsequence constraints that arise in applications. Additionally, we show that many subsequence constraints—including and beyond the traditional ones considered in 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. To this end, we propose a general purpose framework that provides a set of simple and intuitive "pattern expressions", which allows to describe any subsequence constraint of interest and explore algorithms for efficiently mining frequent subsequences under such general constraints.

Our experimental study on real-world datasets indicates that our proposed algorithms are scalable and effective across wide range of applications.

Kurzfassung

In dieser Arbeit untersuchen wir skalierbare und universale Methoden zum Mining von häufigen Sequenzen, die eine vorgegebene Teilsequenzbeschränkung erfüllen. Das Mining von häufigen Sequenzen ist eine wesentliche Aufgabe in Data Mining und hat viele Anwendungen in der echten Welt wie Informationsextraktion, Warenkorbanalyse, Mining von Webnutzung oder Websitzungsanalyse. Abhängig von der zugrunde liegenden Anwendung sind wir allgemein daran interessiert, bestimmte häufige Sequenzen zu entdecken, die mithilfe von Teilsequenzbeschränkungen beschrieben werden. Viele Werkzeuge und Algorithmen existieren bereits für diese Aufgabe, allerdings sind diese nicht in der Lage, mit großen Mengen von Daten umzugehen, welche im Anwendungsfall auftreten können und sich im Allgemeinen nicht über eine Reihe von Anwendungen erweitern lassen.

Wir stellen skalierbare, verteilte Sequenz Mining-Algorithmen vor, die auf MapReduce abzielen. Unsere Arbeit baut auf MG-FSM auf, ein verteiltes Framework für das Mining von häufigen Sequenzen. Wir stellen neuartige Algorithmen vor, die das grundlegende MG-FSM-Framework verbessern und erweitern, um traditionelle in Anwendungen auftretende Subsequenzbeschränkungen effizient zu unterstützen, Zusätzlich zeigen wir, dass viele Subsequenzbeschränkungen – einschließlich der traditionell in der Literatur berücksichtigten Beschränkungen und darüber hinaus – in einem einzigen Framework vereinheitlicht werden können. Eine einheitliche Behandlung erlaubt Forschern, gleichzeitig viele Arten von Subsequenzbeschränkungen zu untersuchen (anstatt jede einzeln) und hilft dabei, die Nutzbarkeit von Pattern Mining-Systemen für Anwender zu verbessern. Zu diesem Zweck stellen wir ein universales Framework vor, das eine Menge von einfachen und intuitiven "Pattern Expressions" bereitstellt. Diese erlauben, jede beliebige Beschränkung zu beschreiben und Algorithmen zum effizienten Mining von häufigen Subsequenzen unter solch allgemeinen Bedingungen zu untersuchen.

Unsere Teststudie zu echten Datensätzen zeigt, dass unsere vorgeschlagenen Algorithmen skalierbar und effizient über eine große Breite an Anwendungen hinweg sind.

Acknowledgements

First and foremost, I am very grateful to my advisor Rainer Gemulla for giving me the opportunity to pursue this thesis under his guidance. I thank him for offering me invaluable opportunities and advise to hone my research skills, for allowing me the space and freedom I needed to work, and for continued encouragement and support throughout the many years of my Ph.D. I am very lucky to have him both as a friend and colleague.

I express my sincere thanks to Gerhard Weikum and Simone Ponzetto for serving on my thesis committee. I greatly appreciate them for their time, interest and helpful advise on my thesis. I would like to thank my coauthors—Iris Miliaraki and Klaus Berberich—for their contributions to my research; parts of this thesis would not have been possible without their hard work.

I would like to thank all my friends and colleagues, especially Arjun Jain, Luciano Del Corro, Christina Teflioudi (who translated the abstract), Iulia Bolosteanu, Arunav Mishra, Avishek Anand, Sharath Kumar, Sairam Gurajada, Sourav Dutta, Kiril Gashteovski, and Yanjie Wang; all of whom made the last few years so much fun.

Finally, I thank my Mom, my Dad, and my brother Saurabh for their unfailing support throughout my career. I am indebted for their unconditional love and sacrifice in raising me into a person that I am today. A very special thanks to Rasika, I am very fortunate to have your love and support during all these years. Thank your for being patient with me and sticking through the tough times.

Contents

Abstract iii

Abstract (German version) v

Acknowledgments vii

Contents ix

1 Introduction 1

2 Preliminaries 5
- 2.1 Basic Concepts . . . 5
- 2.2 FSM Approaches . . . 6
  - 2.2.1 Breadth First Search . . . 7
  - 2.2.2 Depth First Search . . . 9

I Traditional Subsequence Constraints: Scalability 13

3 Length and Gap Constraints 15
- 3.1 Preliminaries . . . 16
- 3.2 A Primer on MG-FSM . . . 17
  - 3.2.1 MapReduce . . . 17
  - 3.2.2 Overview of the MG-FSM framework . . . 17
  - 3.2.3 Constructing Partitions . . . 19
- 3.3 Mining Partitions . . . 21
  - 3.3.1 Sequential FSM algorithms . . . 21
  - 3.3.2 Pivot Sequence Miner . . . 22
- 3.4 Temporal Gap Constraints . . . 26
- 3.5 Handling Long Input Sequences . . . 28
- 3.6 Experiments . . . 30
  - 3.6.1 Setup . . . 30
  - 3.6.2 Results . . . 31
- 3.7 Related Work . . . 36

3.8 Summary . . . 37

4 Maximality and Closedness Constraints **39**
4.1 Definitions . . . 39
4.2 Mining Maximal Sequences . . . 41
4.3 Mining Closed Sequences . . . 48
4.4 Experiments . . . 51
4.5 Related Work . . . 53
4.6 Summary . . . 53

5 Hierarchy Constraints **55**
5.1 Preliminaries . . . 56
5.2 Distributed Generalized Sequence Mining . . . 59
5.2.1 Naïve Approach . . . 59
5.2.2 Semi-Naïve Approach . . . 60
5.2.3 Overview of LASH . . . 61
5.3 Partition Construction . . . 63
5.3.1 Generalized w-Equivalency . . . 64
5.3.2 w-Generalization . . . 65
5.3.3 Other Rewrites . . . 66
5.3.4 Putting Everything Together . . . 67
5.4 Sequential GSM Algorithms . . . 69
5.5 Experiments . . . 70
5.5.1 Setup . . . 70
5.5.2 Results . . . 72
5.6 Related Work . . . 79
5.7 Summary . . . 79

II Non-traditional Subsequence Constraints: Expressibility **81**

6 Expressing Subsequence Constraints **83**
6.1 Subsequence Predicates . . . 84
6.2 Pattern Expression Language . . . 86
6.2.1 Pattern Expressions . . . 86
6.2.2 Examples . . . 87
6.3 Computational Model . . . 89
6.3.1 Finite state transducers. . . . 89
6.3.2 Translating pattern expressions . . . 93
6.4 Advanced Pattern Expression Language . . . 94
6.4.1 Advanced Pattern Expressions . . . 95
6.4.2 Examples . . . 101
6.4.3 Translating Advanced Pattern Expression to FSTs . . . 102

6.5 Summary . . . 102

7 FSM with Subsequence Constraints 103
7.1 FSM and Subsequence Predicates (recap) . . . 104
7.2 FST Optimizations . . . 104
7.2.1 Compressed FST . . . 104
7.2.2 Simulating compressed FST . . . 106
7.3 Mining *P*-Frequent Sequences . . . 109
7.3.1 Naïve Approach . . . 109
7.3.2 DESQ-COUNT . . . 109
7.3.3 DESQ-DFS . . . 111
7.4 Reducing Nondeterminism . . . 116
7.4.1 Minimization . . . 117
7.4.2 Pruning Irrelevant Input Sequences . . . 118
7.4.3 Two-pass Simulation . . . 124
7.4.4 Integrating pruning input sequences and two-pass into mining 126
7.5 Experiments . . . 129
7.5.1 Experimental Setup . . . 129
7.5.2 Results . . . 132
7.6 Related Work . . . 139
7.7 Summary . . . 141

III Wrapping Up 143

8 Conclusions 145

Bibliography 147

List of Figures 155

List of Tables 157

List of Algorithms 159

Chapter 1

Introduction

Frequent Sequence Mining (FSM) is an important problem in data mining and has many real-life applications. Examples include, market-basket analysis [Agrawal and Srikant (1995)], web usage mining and session analysis [Srivastava et al. (2000); Pei et al. (2000)], natural language processing [Lopez (2008); Manning and Schütze (1999)], information extraction [Fader et al. (2011); Nakashole et al. (2012)], or computational biology [Benson and Waterman (1994); Brazma et al. (1998); Wang et al. (2004); Hsu et al. (2007)]. In web usage mining, for example, frequent sequences describe common behavior across users (e.g., the order in which users visit web pages). Likewise, in market-basket analysis, frequent sequences are useful for identifying common purchase patterns of customers, predicting behavior of individual customers, or product recommendations. Frequent textual patterns such as "PERSON *is married to* PERSON" are indicative of typed relations between named entities and useful for natural-language processing and information extraction tasks. In this thesis, we study FSM methods that can scale to very large problem instances, and methods that are extensible across wide range of applications.

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, which is also referred to as a taxonomy. For example, *is*→*be*→*VERB* (for words), *Canon5D*→*DSLR camera*→*camera*→*electronics* (for products), or *Angela Merkel*→*politician*→*PERSON* (for named entities). The goal of FSM is to discover subsequences that "appear" in sufficiently many input sequences.

The notion of "appears" depends on the underlying application. In n-gram mining for example, the goal is to discover frequent *consecutive subsequences* of n items. When mining frequent sequences of user actions (e.g., buying a product, visiting

a webpage, or listening to a song) from log files, *non-consecutive subsequences* in which items are "close" in the input (say within hours) are more desirable. Some applications target specific subsequences like frequent *adjective-noun* pairs or frequent *verbal phrases* between two named entities. These different notions of what constitutes a subsequence has been the subject of much research in this field and has led to several algorithms [Srikant and Agrawal (1996); Garofalakis et al. (1999); Zaki (2001b, 2000); Pei et al. (2000, 2001, 2002); Wang and Han (2004); Wang et al. (2004); Berberich and Bedathur (2013); Trummer et al. (2015)], each of which is designed for a certain notion of subsequence (e.g., consecutive, non-consecutive, or application-specific).

In this thesis, we study methods for frequent sequence mining with *subsequence constraints*, where the subsequence constraint describes which frequent subsequences should be discovered. A key problem with existing methods is that they typically operate on a single machine and, therefore, cannot deal with today's vast amounts of data. Consider for example a document collection with billions of sequences or a website with millions of registered users. At such massive scales, distributed and scalable FSM algorithms are essential for many applications. Another key problem of existing methods is that they are tailored for a fixed notion of subsequence constraint, which limits their usability across applications. For example, an algorithm to discover frequent n-grams cannot be used to discover frequent adjective-noun pairs or vice-versa. As a result, practitioners often end up developing customized algorithms for their specific notion of a subsequence constraint, which is cumbersome and time consuming. To avoid this, we require a unified treatment of subsequence constraints. This thesis contributes FSM methods that are scalable and that are general-purpose for wide range of applications.

Contributions

We propose scalable distributed (i.e., shared nothing) FSM algorithms that target MapReduce. MapReduce, developed at Google in 2004, constitutes a natural environment for large-scale distributed data processing on clusters of many commodity machines and can handle hardware and software failures transparently. Our methods make use of Apache Hadoop, which is an open-source implementation of MapReduce widely used in industry and constitutes a core building block within big-data initiatives. We build on the work of Miliaraki et al. (2013), which contributed the MG-FSM framework for distributed sequence mining. MG-FSM carefully partitions the input sequence sequences into many smaller partitions in a way that they can be processed independently. We extend this framework in multiple ways:

- We propose a novel, special-purpose algorithm caller *pivot sequence miner* to efficiently process the partitions created by MG-FSM. In contrast to exiting FSM algorithms, our approach is "partition-aware" and completely avoids any

post-processing of results, which significantly boosts the performance of MG-FSM. We also propose techniques to handle *time-annotated sequences*, which commonly arise in applications such as session analysis and, techniques to efficiently handle datasets with very long input sequences.

- We propose a distributed algorithm called MG-FSM$^+$ to mine *maximal* and *closed* sequences in a scalable fashion. Such sequences compactly represent the set of all frequent sequences and can significantly reduce the number frequent sequences being discovered with minimal loss of information.
- Finally, we propose our LASH algorithm, which is based on MG-FSM, but efficiently handles item hierarchies. The confluence of hierarchies and frequent sequence mining allows us to discover subsequences that would otherwise be hidden. Example of such subsequences include shopping patterns such as "customers frequently buy some *DSLR camera*, then some *tripod*, then some *flash*" or textual patterns like "the ADJECTIVE house".

Additionally, we propose a general-purpose framework for frequent sequence mining that allows applications to express their notion of subsequence constraint. We introduce *subsequence predicates* to model subsequence constraints in a general way and show that many subsequence constraints—including and beyond those considered in literature—can be unified in a single framework. A unified treatment allows researchers to study subsequence constraints in general instead of focusing on certain combinations individually. It also helps to improve usability of pattern mining systems for practitioners because it avoids the need to develop customized mining algorithms for particular subsequence constraint of interest. For example, our system named DESQ, can efficiently mine frequent n-grams, adjective-noun pairs, typed relational phrases, sequences of products bought after purchase of a camera, or sequences that match a regular expression. In more detail,

- We propose a simple and intuitive *pattern expression language* which can express many subsequence constraints in a unified way. Our pattern expressions are based on regular expressions but allows the use of additional features such as item hierarchies. For example, the pattern expression "(ENTITY$^\uparrow$ VERB$^+$ NOUN* PREP? ENTITY$^\uparrow$)" expresses typed relational phrases in natural language text.
- We propose finite state transducers (FST) as the underlying computational model for pattern expressions and propose methods to extend, compress and optimize FSTs to handle sequence mining tasks. We develop algorithms based on FST simulations to efficiently mine frequent sequences.

All of our methods are publicly available as open-source software.

Publications

The work presented in this thesis is based on the following peer-reviewed publications.

- Beedkar, K., Berberich, K., Gemulla, R., and Miliaraki, I. (2015). Closing the gap: Sequence mining at scale. *ACM Trans. Database Syst.*, 40(2):8:1–8:44.
- Beedkar, K. and Gemulla, R. (2015). LASH: Large-scale sequence mining with hierarchies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 491–503.
- Beedkar, K. and Gemulla, R. (2016). DESQ: Frequent sequence mining with subsequence constraints. In *IEEE International Conference on Data Mining*, pages 793–798.

Outline

In Chapter 2, we introduce some basic concepts and approaches related to FSM. Thereafter, the thesis is organized organized into three parts. In Part I (Chapters 3–5), we propose scalable methods for frequent sequence mining focusing on traditional subsequence constraints including length, gap, maximality and closedness, and hierarchy constraints. In Part II (Chapters 6 and 7), we discuss our general-purpose framework for FSM. Finally, Part III; Chapter 8 presents a summary of the contributions made in this thesis and provides an outlook on future research directions.

Chapter 2

Preliminaries

We start by introducing basic concepts in frequent sequence mining (FSM), some notations and terminology that we will use through out the course of this thesis, and a formal definition of the FSM problem. Next, we discuss popular approaches based on breadth-first search and depth-first search to solve the FSM problem.

2.1 Basic Concepts

Sequence database

A *sequence database* $\mathscr{D} = \{T_1, T_2, \ldots, T_{|\mathscr{D}|}\}$ is a multiset of *input sequences*.[a] Each *input sequence* $T = t_1 t_2 \ldots t_{|T|}$ is an ordered list of items from a vocabulary $\Sigma = \{w_1, w_2, \ldots, w_{|\Sigma|}\}$. We denote by ϵ the empty sequence, by $|T|$ the length of sequence T, and by $\Sigma^*(\Sigma^+)$ the set of all (all non-empty) sequences that be constructed from items in Σ. We will often use symbol T to refer to an input sequence and symbol S to refer to an arbitrary sequence. An example sequence database $\mathscr{D}_{ex}$ consisting of four input sequences is shown in Figure 2.1.

Subsequences

Let $S = s_1 s_2 \ldots s_{|S|}$ and $T = t_1 t_2 \ldots t_{|T|}$ be two sequences composed of items from Σ. We say that S is a *subsequence* of T, denoted $S \subseteq T$, if S can be obtained by deleting items in T. More formally, $S \subseteq T$ iff there exists integers $1 \leq i_1 < i_2 <$

[a] We indicate both sets and multisets using { }; the appropriate type is always clear from the context. The operators $\uplus$, $\uplus$, and $\setminus^+$ correspond to multiset union, multiset intersection, and multiset difference.

$T_1 : bbcac$
$T_2 : bdca$
$T_3 : cbba$
$T_4 : daca$

Figure 2.1: *An example sequence database*

$\ldots < i_{|S|} \leq |T|$ such that $s_k = t_{i_k}$ for $1 \leq k \leq |S|$. For input sequence $T_2 = bdca$ of our example sequence database, we have $bca \subseteq T_2$, $ba \subseteq T_2$, and $cba \not\subseteq T_2$.

Support

Denote by,

$$\mathrm{Sup}(S, \mathscr{D}) = \{\, T \in \mathscr{D} : S \subseteq T \,\}$$

the *support set* of sequence S in the database $\mathscr{D}$, i.e., the multiset of input sequences in which S occurs. In our example database, we have $\mathrm{Sup}(ba, \mathscr{D}_{ex}) = \{\, T_1, T_2, T_3 \,\}$. Denote by,

$$f(S, \mathscr{D}) = |\mathrm{Sup}(S, \mathscr{D})|$$

the *frequency* (or *support*) of S in the database $\mathscr{D}$; e.g., $f(ba, \mathscr{D}_{ex}) = 3$. Our measure of frequency corresponds to the notion of *document frequency* used in text mining, i.e., we count the number of input of sequences (documents) in which S occurs (as opposed to the total number of occurrences of S). We say that sequence S is *frequent* in database $\mathscr{D}$ if its frequency passes the *support threshold* $\sigma > 0$, i.e., $f(S, \mathscr{D}) \geq \sigma$.

FSM problem (unconstrained)

> Given a sequence database $\mathscr{D}$ and a minimum support threshold $\sigma > 0$,
> find all frequent sequences S along with their frequencies $f(S, \mathscr{D})$.

In our example database $\mathscr{D}_{ex}$ and for $\sigma = 2$ we obtain (sequence-frequency)-pairs: $(a, 4)$, $(b, 3)$, $(c, 4)$, $(d, 2)$, $(ac, 2)$, $(ba, 3)$, $(bb, 2)$, $(bc, 2)$, $(ca, 4)$, $(bba, 2)$, and $(bca, 2)$.

A more general variant of this problem is often considered in the literature, in which input sequences are formed of itemsets rather than of individual items. We focus on the important special case of individual items in this thesis (e.g., textual data, user sessions, event logs).

2.2 FSM Approaches

The complete search space consisting of all possible subsequences is a lattice formed by the set Σ^+ and the partial order $\subseteq$. Although in theory such a lattice is infinite, in practice the depth of the lattice is bounded by length of the input sequence that has maximum length. Moreover, the set of all frequent sequences forms a meet-lattice

Algorithm 2.1 Breadth-first search

Require: $\mathscr{D}, \sigma$
Ensure: Frequent sequences in $\mathscr{D}$
1: scan $\mathscr{D}$ to compute length-1 frequent sequences F_1
2: $l \leftarrow 1$
3: **while** $F_l \neq \emptyset$ **do**
4: $C_{l+1} \leftarrow F_l \bowtie F_l$ // *Generate candidate length-($l+1$) sequences*
5: $F_{l+1} \leftarrow \emptyset$
6: **for** $S \in C_{l+1}$ **such that** $f(S, \mathscr{D}) \geq \sigma$ **do** // *Determine frequent length-($l+1$) sequences*
7: $F_{l+1} \leftarrow F_{l+1} \cup S$
8: **end for**
9: $l \leftarrow l + 1$
10: **end while**

that leads to the downward closure property of frequent sequences also known as *support anti-monotonicity* [Zaki (2001b)].

Lemma 2.1 (Support Anti-monotonicity). *For any two sequences S_1 and S_2 such that $S_1 \subseteq S_2$, we have* $\text{Sup}(S_1, \mathscr{D}) \supseteq \text{Sup}(S_2, \mathscr{D})$ *and consequently* $f(S_1, \mathscr{D}) \geq f(S_2, \mathscr{D})$.

The lemma states that all frequent sequences must be composed of frequent subsequences. In other words, if a sequence S is infrequent we can safely prune the search space comprising of sequences in the sub-lattice that meet at S.

The above lemma has lead to different search strategies for enumerating frequent sequences, which can be categorized into breadth-first search (BFS) or depth-first search (DFS) approaches.

2.2.1 Breadth First Search

Methods based on BFS use a *level-wise* approach to generate frequent sequences, i.e., they iteratively generate all frequent sequences of length 1, then length 2, and so on. In each iteration we perform a *candidate-generation-and-test* step in which we make use of frequent length-l sequences to generate candidate sequences of length-$(l+1)$ that are potentially frequent and then determine which of these candidates are actually frequent.

BFS approach is given as Algorithm 2.1. We start by scanning the input sequence database and compute frequent length-1 sequences F_1 (line 1). Thereafter, in each iteration, for $l \geq 1$, we compute candidate length-$(l+1)$ sequences C_{l+1} by *joining* frequent length-l sequences F_l (line 4). Two sequences $S = s_1 s_2 \ldots s_l \in F_l$ and $S' = s'_1 s'_2 \ldots s'_l \in F_l$ are joined if length-$(l-1)$ suffix of S is the same as length-$(l-1)$ prefix of S', i.e., $s_2 \ldots s_l = s'_1 \ldots s'_{l-1}$. For example, sequences ac and ca can be joined to generate sequence aca. As another example, sequences bdc and dca

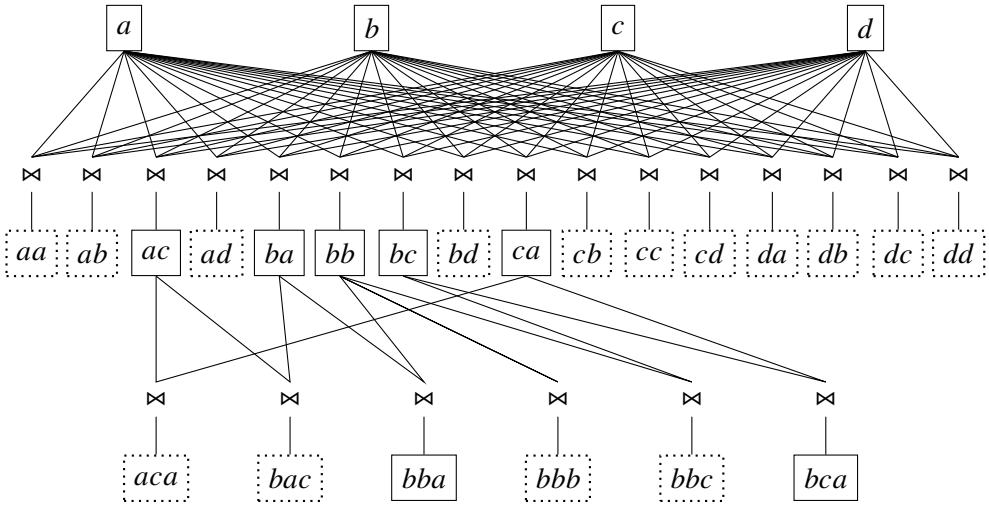


Figure 2.2: *Frequent sequence enumeration using BFS approach.*

Iteration (l)	F_l	Posting list	Support
1	a	$T_1\langle 4\rangle$, $T_2\langle 4\rangle$, $T_3\langle 4\rangle$, $T_4\langle 2\rangle$, $T_4\langle 4\rangle$	4
	b	$T_1\langle 1\rangle$, $T_1\langle 2\rangle$, $T_2\langle 1\rangle$, $T_3\langle 2\rangle$, $T_3\langle 3\rangle$	3
	c	$T_1\langle 5\rangle$, $T_2\langle 3\rangle$, $T_3\langle 1\rangle$, $T_4\langle 3\rangle$	4
	d	$T_2\langle 2\rangle$, $T_4\langle 1\rangle$	2
2	~~aa~~	$T_4\langle 2\rangle$	1
	~~ab~~	$\emptyset$	0
	ac	$T_1\langle 4\rangle$, $T_4\langle 2\rangle$	2
	~~ad~~	$\emptyset$	0
	ba	$T_1\langle 1\rangle$, $T_1\langle 2\rangle$, $T_2\langle 1\rangle$, $T_3\langle 2\rangle$, $T_3\langle 3\rangle$	3
	bb	$T_1\langle 1\rangle$, $T_3\langle 2\rangle$	2
	bc	$T_1\langle 1\rangle$, $T_1\langle 2\rangle$, $T_2\langle 1\rangle$	2
	~~bd~~	$T_2\langle 1\rangle$	1
	ca	$T_1\langle 3\rangle$, $T_2\langle 3\rangle$, $T_3\langle 1\rangle$, $T_4\langle 3\rangle$	4
	...		
3	~~aca~~	$T_4\langle 2\rangle$	1
	~~bac~~	$T_1\langle 1\rangle, T_1\langle 2\rangle$	1
	bba	$T_1\langle 1\rangle$, $T_3\langle 1\rangle$	2
	~~bbb~~	$\emptyset$	0
	~~bbc~~	$T_1\langle 1\rangle$	1
	bca	$T_1\langle 1\rangle$, $T_1\langle 2\rangle$, $T_2\langle 1\rangle$	2
4	$\emptyset$	-	-

Figure 2.3: *Some inverted indexes and supports for each sequence explored by by BFS.*

can be joined to generate sequence *bdca*. Candidates with sufficient support are then added to the set F_{l+1} of frequent length-$(l+1)$ sequences (lines 6–8), which are then used in the next iteration. Figure 2.2 illustrates the BFS approach on our example sequence database $\mathscr{D}_{ex}$ and $\sigma = 2$. Here solid nodes represent frequent sequences, dotted nodes represent infrequent sequences, and joins are shown using the $\bowtie$ symbol.

BFS approaches mainly differ in the candidate-generation-and-test step. For example, the GSP algorithm [Srikant and Agrawal (1996)] uses a hash-tree data structure to store candidates and repeatedly scans the input sequence database to determine frequent candidates. The PSP algorithm [Masseglia et al. (1998)] emulates the GSP approach but uses a prefix-tree data structure to store candidates. The SPADE algorithm [Zaki (2001b)], which we also use in our work, efficiently performs the candidate-generation-and-test step by making use of a vertical representation of the sequence database. In vertical representation, we make use of an inverted index which maps each frequent length-l sequence S to its *posting list* consisting of the set of input sequences in which S occurs as well as its corresponding positions. For example, the sequence *ac* occurs in input sequences T_1 and T_4 at positions 4 and 2 respectively. We generate candidate length-$(l+1)$ sequences by intersecting posting lists of its corresponding length-l prefix and suffix and add it to the inverted index if its frequent, i.e., when the number of distinct input sequences in the posting list is greater than the support threshold σ. At the end of each iteration, we delete length-l sequences F_l from the index. Figure 2.3 shows some posting lists for our example sequence database in each iteration of BFS. Entries that are struck out show infrequent candidate sequences generated in each iteration and are not added to the inverted index.

2.2.2 Depth First Search

An alternate to BFS is to use DFS to explore the search space (lattice). DFS approaches recursively grow shorter frequent to generate longer candidate sequences and differ in the way they generate candidates and compute their supports. For example, Zaki (2001b) suggested the vertical database format and posting list intersections for DFS; the difference to its BFS counterpart is that it grows sequences by intersecting their posting lists with length-1 posting lists. Ayres et al. (2002) proposed the SPAM algorithm, which uses a vertical representation of the sequence database like SPADE, but using a bitmap structure count sequences. Han et al. (2000) proposed the FreeSpan and Pei et al. (2001) later proposed its more efficient version called PrefixSpan, which follows a *pattern-growth* approach to recursively generate frequent sequences. Both approaches arrange the output sequences in a tree, in which each node corresponds to a sequence S and is associated with a *projected database* $\mathscr{D}_S$, which consists of the input sequences in which S occurs. Starting with an empty sequence and a full sequence database, the tree is built recursively by

Algorithm 2.2 Depth-first search

Require: $\mathscr{D}, \sigma$
Ensure: Frequent sequences in $\mathscr{D}$
1: $S \leftarrow \epsilon$
2: $\mathscr{D}_S \leftarrow \mathscr{D}$
3: Expand($S, \mathscr{D}_S$)
4:
5: Expand($S, \mathscr{D}_S$)
6: Scan $\mathscr{D}_S$ to compute $\Sigma_{\mathscr{D}_S}$ // *Compute items to the right of S in $\mathscr{D}$*
7: **for all** $w' \in \Sigma_{\mathscr{D}_S}$ with $f(Sw, \mathscr{D}_S) \geq \sigma$ **do**
8: Output $(Sw, f(Sw, \mathscr{D}_S))$
9: Expand($Sw, \mathscr{D}_{Sw}$) // *Expand with frequent items*
10: **end for**

performing a series of *expansions*. In each expansion, a frequent length-l sequence is expanded to generate candidate length-$(l + 1)$ sequences, their projected databases and their supports. The main difference between the two lies in constructing projected databases. FreeSpan uses an item-based partitioning of the output search space and uses frequent items to recursively projected sequence databases, i.e., it expands a length-l sequence S to generate candidate length-$(l + 1)$ sequences that contain S. PrefixSpan, on the other hand employs a suffix-based partitioning of the output space and uses frequent prefixes to recursively project sequence databases, i.e., in each expansion it expands a frequent length-l sequence to generate candidate length-$(l + 1)$ sequences with prefix S.

In our work, we use the DFS approach of PrefixSpan, which is given as Algorithm 2.2. We start with an empty sequence ϵ and the full sequence database $\mathscr{D}$ (lines 1 and 2) and perform a series of expansions (lines 3 and 9). In each expansion (lines 6–8), when we expand S, we look for the set of items in input sequences $T \in \mathscr{D}_S$, which is given by $\Sigma_S(T) = \{ w \mid Sw \subseteq T \}$, i.e., we look for occurrences of S, and consider the items that occur to the right of S. For example, we have $\Sigma_b(T_1) = \{ b, c, a \}$. Expansion is performed by scanning $\mathscr{D}_S$ and computing the set $\Sigma_{\mathscr{D}_S} = \cup_{T \in \mathscr{D}} \{ \Sigma_S(T) \}$ of items and their frequencies (in $\mathscr{D}_S$). Continuing our example, we have $\Sigma_{\mathscr{D}_b} = \{ b, c, a, d \}$. For each frequent item $w \in \Sigma_S$, we output Sw and recursively grow Sw.

Figure 2.4 illustrates the DFS approach on our running example. As before, solid nodes represent frequent sequences and dotted nodes represent infrequent sequences. Each edge corresponds to an expansion and is labeled by the order in which expansions are performed. We start with the empty sequence ϵ perform the first expansion (E 1) to obtain all length-1 sequences a, b, c, and d. We then expand a (E 2) to obtain ac and aa from which only ac is turns out to be frequent. Thereafter, we expand ac to obtain aca (E 3), which is infrequent. At this point, no more expansions are performed and we return the sequence b. In a similar fashion, we

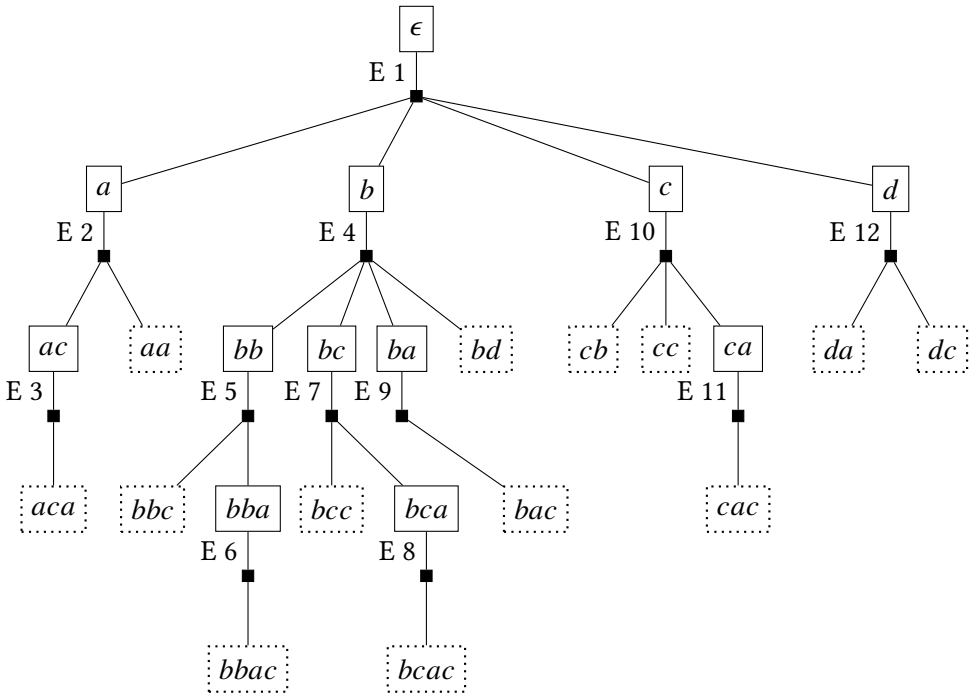


Figure 2.4: *Frequent sequence enumeration using a DFS approach.*

Expansion	Sequence	Projected Database	Support
E 1	a	$T_1\langle 4\rangle$, $T_2\langle 4\rangle$, $T_3\langle 4\rangle$, $T_4\langle 2\rangle$, $T_4\langle 4\rangle$	4
	b	$T_1\langle 1\rangle$, $T_1\langle 2\rangle$, $T_2\langle 1\rangle$, $T_3\langle 2\rangle$, $T_3\langle 3\rangle$	3
	c	$T_1\langle 5\rangle$, $T_2\langle 3\rangle$, $T_3\langle 1\rangle$, $T_4\langle 3\rangle$	4
	d	$T_2\langle 2\rangle$, $T_4\langle 1\rangle$	2
E 2	ac	$T_1\langle 5\rangle$, $T_4\langle 3\rangle$	2
	aa	$T_4\langle 4\rangle$	1
E 3	aca	$T_4\langle 4\rangle$	1
E 4	bb	$T_1\langle 2\rangle$, $T_3\langle 3\rangle$	2
	bc	$T_1\langle 3\rangle$, $T_1\langle 5\rangle$, $T_2\langle 3\rangle$	2
	ba	$T_1\langle 4\rangle$, $T_2\langle 4\rangle$ $T_3\langle 4\rangle$	3
	bd	$T_2\langle 2\rangle$	1
E 5	bbc	$T_1\langle 3\rangle, T_1\langle 5\rangle$	1
	bba	$T_1\langle 4\rangle$, $T_3\langle 4\rangle$	2
E 6	$bbac$	$T_1\langle 5\rangle$	1
...			

Figure 2.5: *Some expansions, projected databases and supports.*

recursively expand b (E 4–E 9), c (E 10, E 11), and finally d (E 12) to enumerate all frequent sequences.

The major cost in DFS arises from computing projected databases and determining frequent items in each expansion. To efficiently perform expansions, we model projected databases using posting lists. Our use of posting lists is reminiscent of the *pseudo-projection* technique of PrefixSpan, which is more efficient than constructing *physical* projected databases [Pei et al. (2001)]. In more detail, we initially scan the input sequence database and construct posting lists for all length-1 sequences. When expanding a sequence S, we use its posting list as follows. For each posting $T\langle pos\rangle$, we consider items in the input sequence T at positions $pos + 1, \ldots, |T|$ and add $T\langle pos'\rangle$ to the posting list of the child node $St_{pos'}$ for $pos < pos' \leq |T|$. For example, consider the sequence bb and its posting list shown in Figure 2.5. When expanding bb we add postings $T_1\langle 3\rangle$ to bba (a occurs at position 3 in T_1), $T_1\langle 4\rangle$ to bbc (c occurs at position 4 at T_1), $T_1\langle 5\rangle$ to bba (a occurs at position 5 at T_1), and $T_2\langle 4\rangle$ to bba (a occurs at position 4 in T_2). Figure 2.5 shows some expansions, projected databases, and supports for some sequences for our running example.

Part I

Traditional Subsequence Constraints: Scalability

Chapter 3

Length and Gap Constraints

In this chapter,[a] we develop scalable methods for frequent sequence mining (FSM) with length and gap constraints. Recall that the goal of FSM is to find all subsequences that appear in sufficiently many input sequences. In practice, it is often useful to focus on subsequences that are contiguous or "close" and/or have bounded length. For example, n-gram mining [Berberich and Bedathur (2013)] aims to find consecutive subsequences of length n in text where as word association mining [Church and Hanks (1990)] aims to find combinations of words that frequently appear in close proximity (but not necessarily consecutively). Similarly, when mining web usage data or any form of log files, sequences of items that are close may be more insightful than sequences of far-away items. This notion of closeness is addressed by gap-constrained frequent sequence mining [Srikant and Agrawal (1996)] in which FSM is parameterized with a *gap-constraint* $\gamma \geq 0$ and *length-constraint* $\lambda \geq 1$. Informally, for a given input sequence, we only consider subsequences that can be generated without skipping more than γ consecutive items and have at most λ items. We obtain n-gram mining for $\gamma = 0$ and $\lambda = n$, word association mining for (say) $\gamma = 5$ and $\lambda = 2$, and unconstrained FSM for $\gamma = \infty$ and $\lambda = \infty$.

There are several well-known approaches [Srikant and Agrawal (1996); Zaki (2000); Pei et al. (2002)] for length- and gap-constrained FSM. However, these methods typically operate on a single machine, and therefore, cannot deal with today's vast amount of data. Miliaraki et al. (2013) proposed a scalable, distributed algorithm for gap-constrained FSM called MG-FSM that can handle billions of sequences. MG-FSM is based on MapReduce [Dean and Ghemawat (2008)], which constitutes a natural environment of large-scale, distributed data processing. MG-FSM partitions the

[a] The material in this chapter is based on Beedkar et al. (2015) and Beedkar and Gemulla (2015).

input data into many smaller partitions in way that reduces the communication cost and that the partitions can be mined independently and in parallel. However, at times, it suffers from high computational cost of mining each partition. Moreover, the basic framework only supports positional gap-constraints and is inefficient for mining datasets with long input sequences.

In this work, we propose several extensions to the basic MG-FSM algorithm. First, we derive the pivot sequence mining algorithm for mining partitions created by the MG-FSM framework. Our pivot sequence miner substantially reduces the computation cost of mining partitions and is up to 5× faster than basic MG-FSM algorithm. Second, we discuss methods to support temporal sequence mining, in which items are annotated with timestamps. This allows MG-FSM to handle temporal gaps (such as "at most one minute" for session analysis). Finally, we develop indexing techniques that enables MG-FSM to efficiently handle datasets in which input sequences are very long.

The remainder of this chapter is organized as follows. In Section 3.1, we formally define the problem of gap-constraint frequent sequence mining and establish the notation used throughout the chapter. In Section 3.2, we give an overview of the basic MG-FSM algorithm. In Section 3.3, we detail the pivot sequence mining algorithm for mining partitions locally. Discussion on handling temporal gap-constraints and long input sequences are detailed in Sections 3.4 and 3.5 respectively. We discuss related work in Section 3.7 and present out experimental study in Section 3.6 before summarizing in Section 3.8.

3.1 Preliminaries

We start by formally defining the problem of length- and gap-constrained frequent sequence mining.

Gap-constrained subsequences

Denote by $\gamma \geq 0$ the *maximum-gap parameter*. We say that S is a γ-subsequence of T, denoted $S \subseteq_\gamma T$, when S is a subsequence of T and there is a gap of at most γ between consecutive items selected from T. Formally, $S \subseteq_\gamma T$ iff there exists integers $1 \leq i_1 < i_2 < \ldots < i_{|S|} \leq |T|$ such that $s_k = t_{i_k}$ (as before) and $i_{k+1} - i_k - 1 \geq \gamma$ for $1 \leq k \leq |S|$. For example, if $T = abcd$ then $acd \subseteq_1 T$, $ad \subseteq_2 T$, and $ad \not\subseteq_0 T$.

Problem Statement

Denote by

$$\mathrm{Sup}_\gamma(S, \mathscr{D}) = \{\, T \in \mathscr{D} : S \subseteq_\gamma T \,\},$$

the γ-*support* of sequence S in the database $\mathscr{D}$, i.e., the multiset of input sequences in which S occurs and denote by $f_\gamma(S, \mathscr{D}) = |\mathrm{Sup}_\gamma(S, \mathscr{D})|$ the γ-*frequency* of sequence S. For $\sigma > 0$, we say that sequence S is (σ, γ)-frequent if $f_\gamma(S, \mathscr{D}) \geq \sigma$.

The length- and gap-constrained frequent sequence mining problem considered in chapter is as follows:

> Given a *support threshold* $\sigma \geq 1$, a *maximum-gap parameter* $\gamma \geq 0$, and a *length threshold* $\lambda \geq 1$, find the set $F_{\sigma,\gamma,\lambda}(\mathscr{D})$ of all (σ, γ)-frequent sequences in $\mathscr{D}$ of length at most λ. For each such sequence, also compute its frequency $f_\gamma(S, \mathscr{D})$.

For example, for database $\mathscr{D}_{ex} = \{\, abcaaabc, abcbbabc, abcccabc \,\}$, we obtain

$$
\begin{aligned}
F_{3,0,2}(\mathscr{D}_{ex}) &= \{\, a, b, c, ab, bc \,\}, \\
F_{3,1,2}(\mathscr{D}_{ex}) &= \{\, a, b, c, ab, ac, bc \,\}, \text{ and} \\
F_{3,2,2}(\mathscr{D}_{ex}) &= \{\, a, b, c, ab, ac, bc, ca \,\}.
\end{aligned}
$$

3.2 A Primer on MG-FSM

MG-FSM provides a general-purpose distributed framework for frequent sequence mining based on the MapReduce. It partitions the input data into many smaller partitions that can be mined independently and in parallel. In this section, we give an overview of the MG-FSM framework and briefly discuss its partitioning techniques.

3.2.1 MapReduce

MapReduce, developed by Dean and Ghemawat (2008) at Google, is a popular framework for distributed data processing on clusters of commodity hardware. It operates on key-value pairs and allows programmers to express their problem in terms of a *map* and a *reduce* function. Key-value pairs emitted by the map function are partitioned by key, sorted, and fed into the reduce function. An additional *combine* function can be used to pre-aggregate the output of the map function and increase efficiency. The MapReduce runtime takes care of execution and transparently handles failures in the cluster. While originally proprietary, open-source implementations of MapReduce, most notably Apache Hadoop, are available and have gained widespread adoption.

3.2.2 Overview of the MG-FSM framework

The key idea of the MG-FSM algorithm is to partition the set of output sequences using *item-based partitioning*. Item-based partitioning is a well-known concept in frequent itemset mining; it is used, for example, in the FP-growth algorithm [Han et al. (2004)] as well as in the distributed frequent itemset miners of [Buehrer et al. (2007); Li et al. (2008)]. MG-FSM creates a partition $\mathscr{P}_w$ for every frequent item

Algorithm 3.1 The MG-FSM algorithm

Require: Sequence database $\mathscr{D}$, σ, γ, λ, f-list $F_{\sigma,0,1}(\mathscr{D})$

```
1: Map(T):
2: for all distinct w ∈ T s.t. w ∈ F_{σ,0,1}(𝒟) do
3:    Construct a sequence database 𝒫_w(T) that is (w, γ, λ)-equivalent to { T }
4:    For each S ∈ 𝒫_w(T), output (w, S)
5: end for
6:
7: Reduce(w, 𝒫_w):
8: F_{σ,γ,λ}(𝒫_w) ← FSM_{σ,γ,λ}(𝒫_w)
9: for all S ∈ F_{σ,γ,λ}(𝒫_w) do
10:   if p(S) = w and S ≠ w then
11:      Output (S, f_γ(S, 𝒫_w))
12:   end if
13: end for
```

$w \in \Sigma$ and then mines frequent length- and gap-constrained sequences in each partition independently. The item w is referred to as the *pivot item* of partition $\mathscr{P}_w$. The MG-FSM algorithm is divided into a preprocessing phase, a partitioning phase, and a mining phase; all of which are fully parallelized.

Preprocessing phase

In the preprocessing phase, we compute the frequency of each item $w \in \Sigma$ and construct the set $F_{\sigma,0,1}(\mathscr{D})$ of frequent items, commonly called *f-list*. This can be done efficiently in a single MapReduce job (by running a version of WordCount that ignores repeated occurrences of items within an input sequence). We use the f-list to establish a total order $<$ on Σ: Set $w < w'$ if $f_0(w, \mathscr{D}) > f_0(w', \mathscr{D})$; ties are broken arbitrarily. Thus items are ordered by decreasing frequency. Write $S \leq w$ if $w' \leq w$ for all $w' \in S$ and denote by $\Sigma^+_{\leq w} = \{ S \in \Sigma^+ : w \in S, S \leq w \}$ the set of all sequences that contain w but no items larger than w. Finally, denote by $p(S) = \min_{w \in S}(S \leq w)$ the *pivot item* of sequence S, i.e., the largest item in S. Note that $p(S) = w \iff w \in S \wedge S \leq w \iff S \in \Sigma^+_{\leq w}$. For example, when $S = abc$, then $S \leq c$ and $p(S) = c$; here, as well as in all subsequent examples, we assume order $a < b < c < d$.

Partitioning phase

The partitioning and mining phases of MG-FSM are performed in a single MapReduce job. In the partitioning phase, we construct partitions $\mathscr{P}_w$ in the map phase: For each distinct item w in each input sequence $T \in \mathscr{D}$, we compute a small sequence database $\mathscr{P}_w(T)$ and output each of its sequences with reduce key w. We require $\mathscr{P}_w(T)$ to be "(w, γ, λ)-equivalent" to T, see Section. 3.2.3. For now, assume

that $\mathscr{P}_w(T) = \{ T \}$; a key ingredient of MG-FSM is to use rewrites that make $\mathscr{P}_w(T)$ as small as possible.

Mining phase

The mining phase is carried out in the reduce function. The input to the mining phase is given by

$$\mathscr{P}_w = \biguplus_{T \in \mathscr{D}, w \in T} \mathscr{P}_w(T),$$

which is automatically constructed by the MapReduce framework. Each reduce function runs an arbitrary FSM algorithm with parameters σ, γ, and λ on $\mathscr{P}_w$—denoted $\mathrm{FSM}_{\sigma,\gamma,\lambda}(\mathscr{P}_w)$ in Alg. 3.1—to obtain the frequent sequences $F_{\sigma,\gamma,\lambda}(\mathscr{P}_w)$ as well as their frequencies. Since every frequent sequence may be generated at multiple partitions, MG-FSM performs a filtering step to produce each frequent sequence exactly once. In particular, we output sequence S at partition $\mathscr{P}_{p(S)}$, i.e., at the partition corresponding to its largest item.

3.2.3 Constructing Partitions

We now summarize the partition construction of MG-FSM and, in particular rewriting techniques for constructing $\mathscr{P}_w(T)$ for an input sequence T. These rewriting techniques aim to minimize partition size, and therefore reduce communication cost between Map and Reduce phase, computational cost at each partition, and partition skew while maintaining correctness.

w-equivalency.

w-equivalency is a necessary and sufficient condition for the correctness of MG-FSM. A sequence S is a *pivot sequence* w.r.t. $w \in \Sigma$ if $p(S) = w$ and $2 \leq |S| \leq \lambda$. Denote by

$$G_{w,\gamma,\lambda}(T) = [F_{1,\gamma,\lambda}(\{ T \}) \cap \Sigma^+_{\leq w}] \setminus \{ w \}$$

the *set* of pivot sequences that occur in T, i.e., are γ-subsequences of T with largest item w. If $S \in G_{w,\gamma,\lambda}(T)$, then T is said to (w, γ, λ)-generate (or simply w-generate) S. For example,

$$G_{c,1,2}(acbfdeacfc) = \{ ac, cb, cc \}.$$

Two sequences T and T' are said to be (w, γ, λ)-equivalent (or simply w-equivalent), if

$$G_{w,\gamma,\lambda}(T) = G_{w,\gamma,\lambda}(T'),$$

i.e., they both generate the same set of pivot sequences. Similarly, two sequence databases $\mathscr{D}$ and $\mathscr{P}_w$ are (w, γ, λ)-equivalent (or simply w-equivalent) iff

$$G_{w,\gamma,\lambda}(\mathscr{D}) = G_{w,\gamma,\lambda}(\mathscr{P}_w).$$

MG-FSM produces correct results if $\mathscr{P}_w(T)$ is w-equivalent to T for all $T \in \mathscr{D}$.

Constructing $P_w(T)$.

We now summarize rewriting techniques that aim to reduce to overall size of $\mathscr{P}_w(T)$. Let $T = t_1 \dots t_{|T|}$ be an input sequence and consider pivot w. An index $1 \leq i \leq |T|$ is *w-relevant* if t_i is w-relevant, i.e., if $t_i \leq w$; otherwise it is *w-irrelevant*. When $t_i = w$, we say that the index i is *pivot index*. Since irrelevant items do not contribute to a pivot sequence, MG-FSM replaces these items with "blanks". For example, sequence *abddc* is written as $ab␣c$ (for pivot c). Replacing irrelevant items with blanks enables effective compression (e.g., *abddc* can be written as $ab␣^2c$).

Perhaps the most important rewrite is *unreachability reduction* that removes unreachable items, i.e., items that are "far away" from any pivot item. For example, consider a input sequence $T = cadbabeadcddae$ and corresponding sequence $T' = ca␣bab␣a␣c␣␣a␣$ obtained after replacing c-irrelevant items with blanks. Here indexes 1 and 10 are pivot indexes. For removing unreachable items, we compute the left and the right distance to a pivot item. The left distance of an index i is the smallest number of items (number of "hops"+1) from a pivot index to index i; only relevant indexes are considered and subsequence indexes must satisfy the gap constraint (at most γ items in between). Similarly, right distance of an index i is the distance to the closest pivot to the right of i. For example, we obtain the following for $\gamma = 1$:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t_i	c	a	␣	b	a	b	␣	a	␣	c	␣	␣	a	␣
left	1	2	2	3	4	4	–	–	–	1	2	2	–	–
right	1	–	–	4	4	3	3	2	2	1	–	–	–	–

Here – correspond to infinite distance. The left distance for index 5 for example is 4, which is determined by indexes 1, 2, 4, 5 (indexes 1, 3, 5 is not allowed since index 3 is irrelevant). Indexes where the minimum distance min(left, right) $\leq \lambda$ are unreachable and can be safely removed. For example, for $\lambda = 3$ we obtain $T' = ca␣bb␣a␣c␣␣$.

Other important rewrites include *prefix/suffix reduction* where leading and trailing blanks are removed (e.g., $ca␣bb␣a␣c␣␣$ is reduced to $ca␣bb␣a␣c$) and *blank reduction* where any sequence of more than $\gamma + 1$ blanks are replaced with exactly $\gamma + 1$ (e.g., $ca␣␣␣␣cba$ can be reduced to $ca␣␣cba$ for $\gamma = 1$). MG-FSM also performs *blank separation*, where a sequence can be written in terms of multiple shorter sequences (e.g., $acb␣␣bca$ can be split into acb and bca for pivot c). Blank separation, however, is ineffective when items are not often repeated.

Rewrites in practice

In practice, the above rewrites are performed as follows. For each sequence T and each frequent item $w \in T$, MG-FSM performs a backward scan to obtain the right distances of all indexes. It then performs a forward scan of T in which it simultaneously (1) computes the left distances, (2) performs unreachability reduction, (3)

replaces irrelevant items by blanks, (4) performs prefix/suffix and blank reduction to obtain $\mathscr{P}_w(T)$.

3.3 Mining Partitions

In this section, we first briefly discuss how existing FSM approaches of Section 2.2 can be adapted to mine length- and gap-constrained sequences. These approaches mine all frequent sequences and must be combined with a filtering step to restrict output to pivot sequences (cf. line 7, Algorithm 3.1). We then propose a more efficient, special-purpose sequence miner that directly mines pivot sequences.

3.3.1 Sequential FSM algorithms

We first briefly describe how we extend BFS and DFS approaches to handle length and gap constraints and then discuss the overhead associated with them in context of MG-FSM.

BFS with length and gap constraints

Recall that BFS uses a level-wise approach to iteratively generate sequences of length-1, then length-2, and so on. To adapt BFS to handle the length constraint λ, we stop iterative process after λ^{th} iteration, i.e., we add the condition $k \leq \lambda$ in line 3 of Algorithm 2.1. To handle gap constraint γ, we modify the the posting list intersection in which we merge two postings $T\langle pos\rangle$ and $T'\langle pos'\rangle$ only when the conditions $T = T'$ and $pos < pos' - \gamma + 1$ satisfy. For example, consider two sequences a and d and their posting lists $L_a = T_1\langle 1\rangle, T_2\langle 2\rangle, T_3\langle 1\rangle$ and $L_d = T_1\langle 3\rangle, T_3\langle 2\rangle$. We obtain $L_{ad} = T_3\langle 2\rangle$ for $\gamma = 0$ and $L_{ad} = T_1\langle 1\rangle, T_3\langle 1\rangle$ for $\gamma = 1$.

DFS with length and gap constraints

We adapt the DFS approach (Algorithm 2.2) to handle length and gap constraints as follows. To handle length constraint, we only expand a sequence S if $|S| < \lambda$. To handle gap constraint, when we expand S, we only look for the set of right items in input sequences $T \in \mathscr{D}_S$, which is given by $\Sigma_S(T) = \{\, w \mid Sw \subseteq_\gamma T \,\}$, i.e., we look for occurrences of S, and then consider the items that occur to the right of S that are at most $\gamma + 1$ items apart. For example, if $T = cabda$, then we have $\Sigma_{ca}(T) = \{\, b \,\}$ for $\gamma = 0$ and $\Sigma_{ca}(T) = \{\, b, d \,\}$ for $\gamma = 1$.

Overhead

In the context of MG-FSM, the BFS and DFS approaches have substantial computational overhead: They compute and output all frequent sequences, whether or not

these sequences are pivot sequences (i.e., $p(S) = w$) and thus non-pivot sequences need to be pruned. To see this, consider pivot d and example partition

$$\mathscr{P}_d = \{\, adda, cabd, ca\_db, b\_aadbc \,\}, \tag{3.1}$$

for $\sigma = 2$, $\gamma = 1$ and $\lambda = 4$. Both BFS and DFS methods will produce sequences such as ca and ab, neither of which contain pivot d and thus need to be filtered out by MG-FSM. Unfortunately, neither BFS nor DFS can be readily extended to avoid enumerating non-pivot sequences. This is because short non-pivot sequence might contribute to longer pivot sequences. In BFS, we obtain frequent pivot sequence cad from ca (a non-pivot sequence) and ad (a pivot sequence). Similarly, DFS obtains cad by expanding the non-pivot sequence ca. This costly computation of non-pivot sequences cannot be avoided without sacrificing correctness. Note that both approaches also compute frequent sequences that do not contribute to a pivot sequence later on (e.g., sequence ab).

3.3.2 Pivot Sequence Miner

In what follows, we propose PSM, an effective and efficient algorithm that significantly reduces the computational cost of mining each partition. In contrast to the methods discussed above, PSM restricts its search space to only pivot sequences and is thus customized to MG-FSM. We also describe optimizations that further improve the performance of PSM.

Algorithm

The key goal of PSM is to only enumerate pivot sequences. PSM is based on DFS, but, in contrast, starts with the pivot w (instead of the empty sequence) and expands a sequence to the left and to the right (instead of just to the right). Since PSM starts with the pivot, every intermediate sequence will be a pivot sequence. The PSM algorithm is shown as Algorithm 3.2. We assume that for all $T \in \mathscr{P}_w$, $p(T) = w$; this property is ensured by MG-FSM's partitioning framework.

PSM starts with $S = w$ (pivot item) and determines the support set $\mathscr{D}_w$ (line 1); under our assumptions, $\mathscr{D}_w = \mathscr{P}_w$ so that nothing needs to be done. We then perform a series of right-expansions almost as expansions in DFS (lines 2 and 13); the only difference is that we do not right-expand with the pivot item (cf. line 11). After the right-expansions are completed, we have produced all frequent pivot sequences that start with the pivot item (and do not contain another occurrence of the pivot item).

Figure 3.1 illustrates PSM on the partition of Equation (3.1) with pivot d. Solid nodes represent frequent sequences; dotted nodes represent infrequent sequences that are explored by PSM. Each edge corresponds to an expansion and is labeled with its type (RE=right expansion, LE=left expansion) and order of expansion. We start

Algorithm 3.2 Mining pivot sequences

Require: $\mathscr{P}_w$, Σ, σ, γ, λ

1: $S \leftarrow w$, $\mathscr{D}_S \leftarrow \mathrm{Sup}_\gamma(S, \mathscr{P}_w)$
2: Expand(S, $\mathscr{D}_S$, right)
3: Expand(S, $\mathscr{D}_S$, left)
4:
5: Expand(S, $\mathscr{D}_S$, dir)
6: **if** $|S| = \lambda$ **then**
7: **return**
8: **else**
9: Scan $\mathscr{D}_S$ to compute Σ_S^{dir}
10: **if** dir = right **then**
11: **for all** $w' \in \Sigma_S^{\mathrm{dir}} \setminus \{ w \}$ with $f_\gamma(Sw', \mathscr{P}_w) \geq \sigma$ **do**
12: Output $(Sw', f_\gamma(Sw', \mathscr{P}_w))$
13: Expand(Sw', $\mathscr{D}_{Sw'}$, right)
14: **end for**
15: **end if**
16: **if** dir = left **then**
17: **for all** $w' \in \Sigma_S^{\mathrm{dir}}$ with $f_\gamma(w'S, \mathscr{P}_w) \geq \sigma$ **do**
18: Output $(w'S, f_\gamma(w'S, \mathscr{P}_w))$
19: Expand($w'S$, $\mathscr{D}_{w'S}$, right)
20: Expand($w'S$, $\mathscr{D}_{w'S}$, left)
21: **end for**
22: **end if**
23: **end if**

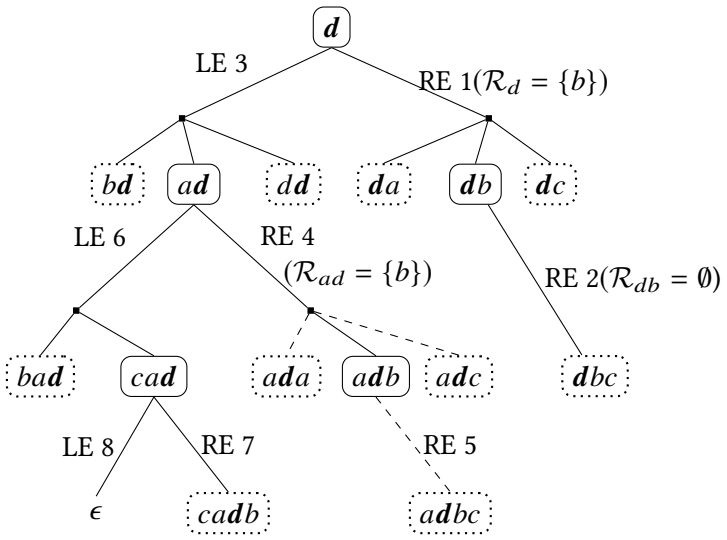


Figure 3.1: *Pivot sequence enumeration for partition $\mathscr{P}_d$ for $\sigma = 2$, $\gamma = 1$ and $\lambda = 4$.*

with sequence d and perform the first right-expansion (RE 1) to obtain sequences da, db, and dc from which only db turns out to be frequent. We then right-expand db to obtain dbc (RE 2), which is infrequent. At this point, no more right-expansions are performed and we return to the pivot.

After the pivot has been right-expanded, PSM performs a *left-expansion* of the pivot w (line 3), producing sequences of form $w'w$. Left-expansions are symmetrical to right-expansions, but we expand S to the left by computing the set of items $\Sigma_S^{\text{left}} = \bigcup_{T \in \mathscr{D}_S} \{\Sigma_S^{\text{left}}(T)\}$, where $\Sigma_S^{\text{left}}(T) = \{\, w' \mid w'S \subseteq_\gamma T \,\}$. In our example, we obtain frequent sequence ad and some infrequent sequences (LE 3). We now perform a sequence of right-expansions on ad (RE 4 and RE 5, line 19 of Algorithm3.2). Note that PSM never left-expands a sequence that is a result of a right-expansion. Once all right-expansions of ad have been computed, we left-expand it (LE 6, line 20) and proceed recursively as above.

Correctness

PSM enumerates each frequent pivot sequence exactly once; there are no duplicates and no missed sequences. To see this, consider an arbitrary pivot sequence S of length at least 2 (with pivot w). Then there is a unique decomposition

$$S = S_l w S_r$$

such that $w \notin S_r$. We refer to S_l as the *prefix* of S (i.e., the part of S occurring to the left of the (last) pivot) and S_r as the *suffix* (to the right). For example, sequence cad with pivot d has prefix $S_l = ca$ and suffix $S_r = \epsilon$. Note that the decomposition is unique because $w \notin S_r$; e.g., $S = adda$ uniquely decomposes into $S_l = ad$, $w = d$

and $S_r = a$. This is the reason why we do not right-expand with the pivot (line 11). PSM generates S from d by first performing left-expansions until $S_l w$ is obtained, and then a series of right-expansions to obtain $S_l w S_r$. Figure 3.1 shows a number of examples; e.g., sequence $cadb$ is obtained by expansions LE 3, LE 6, and RE 7. If PSM were to perform left-expansions after a right expansion, then $cadb$ would also be obtained from a left-expansion of adb (obtained from RE 4). PSM avoids such duplicates.

Indexing right-expansions

We now describe an optimization technique which further reduces the search space. The key idea is to store information of right-expansions to make future right-expansions more efficient. To see why this may help, consider RE 1 and RE 4 in Figure 3.1. From RE 1, we know that da is infrequent. Thus, when performing RE 4, we do not need to consider sequence ada since it must also be infrequent by the anti-monotonicity property of support [Lemma 2.1; (page 7)]. In general, if Sw' is an infrequent right-expansion of S, then $w''Sw'$ will be an infrequent right-expansion of $w''S$.

We make use of this observation as follows. Whenever we perform a right-expansion of some sequence S, we store in an index the set $\mathcal{R}_S$ of the resulting frequent expansion items. In our example, we have $\mathcal{R}_d = \{b\}$ from RE 1 since db is the only frequent right-expansion of d. We subsequently use the information about $\mathcal{R}_S$ as follows. Whenever we perform a right-expansion of some sequence $S_l S$, we restrict the set of expansion items to $\mathcal{R}_S$. In our example, when expanding ad in RE 4, we only consider expansion item b (since $\mathcal{R}_d = \{b\}$). For all other items, neither counting nor support set computation is performed; these items are shown in nodes connected with dashed lines in Figure 3.1. If $\mathcal{R}_S$ is empty, no right-expansions need to be performed and we do not scan the database. This happens for the sequence adb in our example; since we obtain $\mathcal{R}_{db} = \emptyset$ from RE 2, we do not perform RE 5.

Our choice of indexing only right-expansions is tailored to the order in which PSM explores pivot sequences. For example, consider LE 3 in Figure 3.1. Information about frequent left-expansions for $S = d$ (i.e., ad) will not be of any use, since during the traversal, we will never left-expand any sequence of the form SS_r (such as db; recall that PSM never left-expands a sequence that is a result of a right-expansion). Therefore, we only index right-expansions to prune search space. To save memory, our actual implementation unions the indexes of each level of each series of right expansions (i.e., we maintain one index for the frequent items that occur directly after S, one index for the items that occur two items after S, and so on).

Analysis

In what follows, we study the worst-case size of the search space of the PSM algorithm and compare it to the one of the BFS and DFS approaches. Let us assume a hypothetical database (or partition) that has k distinct items and that each sequence in the database has length λ. Further assume that all possible sequences of length up to λ are frequent in the database; there are $\sum_{l=1}^{\lambda} k^l$ such sequences. Both BFS and DFS will first produce all of these sequences, but in the context of MG-FSM, subsequently only output the ones that contain the pivot item. There are $\sum_{l=1}^{\lambda}(k-1)^l$ sequences that do not contain the pivot; these are produced unnecessarily. In contrast, PSM only explores pivot sequences, of which there are $\sum_{l=1}^{\lambda} k^l - \sum_{l=1}^{\lambda}(k-1)^l$. Thus PSM explores a fraction of

$$1 - \frac{\sum_{l=1}^{\lambda}(k-1)^l}{\sum_{l=1}^{\lambda} k^l} \ll 1$$

of the sequences explored by BFS or DFS methods. For example, if $k = 100{,}000$ and $\lambda = 5$, PSM explores 0.005% of the search space of BFS or DFS.

In practice, the worst-case rarely occurs, of course. To shed more light on the relationship between PSM and DFS, consider our running example and suppose that we used DFS. In a first step, DFS computes all (item, frequency)-pairs, of which there are four: $(a, 4)$, $(b, 3)$, $(c, 3)$ and $(d, 4)$. For each so-found frequent sequence, DFS recursively makes a right-expansions to compute longer frequent sequences. In our running example, DFS ultimately computes 12 length-2 sequences (but outputs only the frequent ones): $(ad, 4)$, $(ab, 2)$, $(aa, 1)$, $(bd, 1)$, $(ba, 1)$, $(bc, 1)$, $(ca, 2)$, $(cb, 1)$, $(dd, 1)$, $(da, 1)$, $(db, 2)$, $(dc, 1)$. Similarly, DFS computes 9 length-3 sequences and two length-4 sequences. The total size of the search space of DFS is thus 27. On the other hand, PSM only explores 11 sequential patterns; these are shown by the nodes connected with solid lines in Figure 3.1. Thus PSM explores only roughly half of the search space of DFS in our example.

3.4 Temporal Gap Constraints

We have restricted attention so far to frequent sequence mining with a *positional gap constraint*. In applications such as session analysis, however, input sequences are often built from time-annotated events instead of items; in such applications, *temporal gap constraints* are more suitable [Srikant and Agrawal (1996)]. This means that we want to treat a pair of events as sufficiently close if the in-between time span is small (e.g., events that occur within 1 hour), i.e., independently of the number of events that occurs in between. In this section, we describe how MG-FSM can be adapted to support such temporal gap constraints.

Definition 3.1. *A* temporal sequence *is an ordered list* $T = t_1(\zeta_1)\, t_2(\zeta_2) \cdots t_l(\zeta_l)$ *of* events*, i.e., item-timestamp pairs. For* $1 \leq i \leq l$*, event* $t_i(\zeta_i)$ *consists of item* t_i *taken*

from dictionary Σ and timestamp ζ_i taken from a discrete set of timestamps $\mathcal{T} \subseteq \mathbb{Z}$. The timestamps are distinct and ordered, i.e., $\zeta_i < \zeta_j$ for $1 \leq i < j \leq l$.

Note that the timestamps can be of any desired granularity (e.g., seconds, minutes, hours or days).

Denote by $\Delta_{ij} = \zeta_j - \zeta_i - 1$ the *temporal gap* between events $t_i(\zeta_i)$ and $t_j(\zeta_j)$, $i < j$. Observe that $\Delta_{ij} \geq 0$. To handle temporal gap constraints, we "convert" the temporal gap constraint to a positional gap constraint by mapping the event sequence into a sequence of items and gaps. In particular, we convert temporal sequence $T = t_1(\zeta_1)\, t_2(\zeta_2) \cdots t_l(\zeta_l)$ to regular sequence

$$T' = t_1 \textvisiblespace^{\Delta_{12}} t_2 \textvisiblespace^{\Delta_{23}} t_3 \cdots \textvisiblespace^{\Delta_{(l-1)l}} t_l.$$

Here, ␣ denotes the blank symbol as before; $\textvisiblespace^{\Delta_{i(i+1)}}$ represents as many gaps as time units (without an event occurring) passed between events $t_i(\zeta_i)$ and $t_{i+1}(\zeta_{i+1})$. If two events occur at adjacent timestamps (i.e., their temporal gap is 0), then no blanks appear between the corresponding items. As described in Section 3.2.3, we reduce the overhead of adding gaps to the input sequences by using a compression technique that encodes sequences of consecutive blanks with run-length encoding. With such compression, the conversion takes linear time and space.

To mine frequent sequences with a temporal gap constraint, we simply run MG-FSM with a positional gap constraint on the converted sequences. In particular, denote by τ a maximum temporal-gap parameter; at most τ time units are thus allowed to pass between two events to be considered close. We then set maximum-gap parameter $\gamma = \tau - 1$ when running MG-FSM. To see why this produces the desired result, observe that our rewrite ensures that two events $t_i(\zeta_i)$ and $t_j(\zeta_j)$, $i < j$, with temporal gap Δ_{ij} have exactly Δ_{ij} items or blanks in between them. The time passed between the occurrence of $t_i(\zeta_i)$ and $t_j(\zeta_j)$ is $\Delta_{ij} + 1$; for this reason, we set $\gamma = \tau - 1$ (instead of $\gamma = \tau$).

Consider, for example, the sequence database $\mathcal{D} = \{\, a(2)\, b(3)\, c(6)\, a(8) \,\}$, which consists of a single temporal sequence with four events. After conversion, we obtain sequence database $\mathcal{D}' = \{\, ab\textvisiblespace c\textvisiblespace a \,\}$ by dropping timestamps and adding the respective number of blanks. Set $\sigma = 1$ and $\lambda = 3$. For a temporal-gap constraint of one time unit ($\tau = 1$, and thus $\gamma = 0$), we obtain frequent sequences $F_{1,0,3}(\mathcal{D}') = \{\, a, b, c, ab \,\}$. For $\tau = 2$ ($\gamma = 1$), we obtain $F_{1,1,3}(\mathcal{D}') = \{\, a, b, c, ab, ca \,\}$. Finally, for $\tau = 3$ ($\gamma = 2$), we obtain $F_{1,2,3}(\mathcal{D}) = \{\, a, b, c, ab, bc, ca, abc, bca \,\}$.

As a final note, we remark that MG-FSM can also handle temporal sequences with repeated timestamps, provided that items with equal timestamps have meaningful order (but the granularity of the timestamp is too coarse-grained to capture this order). To do so, we modify both timestamps and the maximum-gap parameter in a way that makes timestamps unique, retains the original order of the events, and ensures correct results. In more detail, we conceptually multiply each timestamp by $2r-1$ before conversion, where $r > 1$ is an upper bound on the number of events that

can occur simultaneously. We then replace repeated timestamps by consecutive sequences of timestamps. For example, the sequence of events $a(1)\,b(1)\,c(2)\,d(2)\,e(2)$ is modified to $a(5)\,b(6)\,c(10)\,d(11)\,e(12)$ for $r = 3$. Now all timestamps are distinct; we convert the database as described above, obtaining $ab\_\_cde$ for our example. We run MG-FSM with maximum-gap parameter

$$\gamma = (\tau - 1)(2r - 1) + (3r - 3).$$

Here $3r - 3$ denotes the maximum positional gap between two events that originally occurred at consecutive timestamps. In our example, where $r = 3$, we set $\gamma = 1$ for $\tau = 0$ and $\gamma = 6$ for $\tau = 1$.

When $\gamma = 0$, a temporal rewrite can also be used to mine sequences of itemsets (as opposed to sequences of items) with MG-FSM. To see this, consider the sequence of itemsets $T = \langle ac\rangle \rightarrow \langle b\rangle$, where we enclose itemsets by $\langle$ and $\rangle$ and separate itemsets by $\rightarrow$. T has the following non-empty subsequences of itemsets

$$\langle a\rangle, \langle b\rangle, \langle c\rangle, \langle ac\rangle, \langle a\rangle \rightarrow \langle b\rangle, \langle c\rangle \rightarrow \langle b\rangle, \langle ac\rangle \rightarrow \langle b\rangle.$$

We can rewrite T to a temporal sequence T' such that every subsequence of itemsets of T corresponds to a subsequence of items of T' and vice versa. To do so, we first flatten the itemsets to an item sequence and put a special *itemset marker item* $\rightarrow$ between itemsets. We obtain the sequence of items $T_1 = ac \rightarrow b$, which consists of four items. We now associate a timestamp with each item, starting from 1 and incrementing by 1 at each occurrence of the itemset marker item as well as at its consecutive item. We then reorder same-timestamp events lexicographically. This gives us the event sequence $T' = a(1)\,c(1) \rightarrow (2)\,b(3)$. With a choice of $\tau = 1$, T' generates the sequences of items

$$a, b, c, ac, a{\rightarrow}b, c{\rightarrow}b, ac{\rightarrow}b,$$

as well as some additional sequences that start or end with $\rightarrow$; we ignore these additional sequences. Then T and T' have equivalent subsequences. Given a database of sequences of itemsets, we rewrite every sequence as just described, apply MG-FSM, and filter out the additional sequences. Although this technique correctly mines sequences of itemsets, it is limited to the case $\gamma = 0$ and care must be taken to support length constraints correctly.

3.5 Handling Long Input Sequences

MG-FSM's partitioning works well as long as the sequences in the sequence database are relatively short (e.g., sequences that correspond to sentences in text mining). This allows us to scan with low cost the entire input sequence repeatedly during partition construction. The assumption of short sequences does not generally

hold, i.e., in some applications sequences can be very long (e.g., sequences that correspond to entire documents in text mining). To handle long sequences, we need to ensure that the rewriting techniques of MG-FSM remain efficient as the length of the input sequences increases.

Recall that for an input sequence T, rewriting methods for constructing partitions perform a backward and a forward scan for each pivot item $w \in T$ (see Section 3.2.3). The total number of backward-forward scans depends on the number of distinct (frequent) items in T, but generally can be as high as $|T|$. Since the scans take linear time, the computational cost is $O(|T|^2)$. This quadratic overall cost is acceptable when T is short ($|T|$ small), but imposes severe overheads when T is long ($|T|$ large). To avoid this performance bottleneck, we propose to build for each input sequence T an *inverted index* structure that maps pivot items to their respective positions in T. By utilizing such an index, we avoid performing a full backward-forward scan and instead perform a focused scan "around" the occurrences of the pivot under consideration.

In more detail, we perform an initial pass over input sequence T to build an inverted index. The index stores for each distinct frequent item $w \in T$ the positions of w's occurrences in T.[b] Given such an index, we construct $\mathscr{P}_w(T)$ for each indexed pivot w as follows. Instead of scanning T, we only consider parts of T that are sufficiently close to w; we ensure that all omitted parts cannot contribute to a pivot sequence. In more detail, denote by $I = (i_1, \ldots, i_r)$ the positions at which w occurs in T. We then restrict the forward-backward scan to the union of the ranges $[i_j - (\gamma+1)(\lambda-1), i_j + (\gamma+1)(\lambda-1)]$ for $1 \leq j \leq r$. All items outside of these ranges are unreachable and therefore do not need to be considered.

There is a trade-off between the construction cost and the benefit of the inverted index. We have described above an inverted index that maintains all positions of each pivot item; we subsequently refer to this index as *full index*. A simple alternative is to use a *min-max index*, which is more efficient to construct. In particular, the *min-max index* maintains only the positions of the first and the last occurrence of each pivot. Given such an index, we perform for pivot w a forward-backward scan of range $[l_w - (\gamma+1)(\lambda-1), r_w + (\gamma+1)(\lambda-1)]$, where l_w and r_w are the positions of the first and last occurrences of w. Observe that if there is only one occurrence of w, we scan identical ranges when using either the min-max index or the full index. If there are multiple occurrences of a pivot, the full index can be more effective than the min-max-index (esp. when the left- and right-most occurrences are far apart).

We now illustrate the effect of the inverted index on the processing cost of a backward-forward scan using an example. Consider sequence

$$T = cadbaefebdaecdgadfae,$$

pivot c, $\gamma = 1$, and $\lambda = 3$. To construct partition $\mathscr{P}_c(T)$ without an index, we need

[b] We use the term "position" instead of "index" to avoid confusion with the index structure.

to scan T twice and thus process $2|T| = 40$ items. If we use the min-max index, we restrict our scans to the neighborhood of c's first occurrence ($l_c = 1$) and c's last occurrence ($r_c = 13$). We scan twice the range $[1 - 2 \cdot 2, 13 + 2 \cdot 2]$ (i.e., range $[1, 17]$) for a total of 34 processed items. Finally, if we use a full index, we scan twice ranges $[1, 5]$ (for the occurrence of c at position 1) and $[9, 17]$ (position 13). The total number of processed items is 28.

3.6 Experiments

We conducted an experimental study in the contexts of text mining and session analysis on large real-world datasets. In particular, we investigated the efficiency of our PSM algorithm for mining each partition, studied MG-FSM's performance for mining temporal sequences, and evaluated the effectiveness of our indexing techniques for handling long input sequences.

We found that our PSM algorithm increased MG-FSM's efficiency by up to 5×. We observed that MG-FSM successfully mined temporal sequences on the Netflix dataset [Bennett and Lanning (2007)]; temporal sequence mining can be expensive, however, when the data contains large bursts of events in small timespans. Finally, our use of inverted indexes for mining long input sequences was effective and significantly decreased partitioning costs.

3.6.1 Setup

Hadoop cluster

We ran our experiments on a local cluster consisting of eleven Dell PowerEdge R720 computers connected using a 10 GBit Ethernet connection. Each machine has 64GB of main memory, eight 2TB SAS 7200 RPM hard disks, and two Intel Xeon E5-2640 6-core CPUs. All machines ran Debian Linux (kernel version 3.2.48.1.amd64-smp), Oracle Java 1.7.0_25, and use the Cloudera cdh3u6 distribution of Hadoop 0.20.2. One machine acted as the Hadoop master node, the other ten machines acted as worker nodes. The maximum number of concurrent map or reduce tasks was set to 8 per worker node. All tasks launched with 4 GB heap space.

Datasets

We used two real-world datasets for our experiments, see Table 3.1. The first dataset is the The New York Times corpus (NYT), which consists of over 1.8 million newspaper articles published between 1987 and 2007. We created two sequence databases from this corpus, denoted NYT-sen and NYT-doc, in which we respectively treat each sentence or each document as an input sequence. The sequences in NYT-doc are substantially longer than the ones in NYT-sen; we thus use NYT-doc to evaluate the effectiveness of our indexing techniques for long sequences (see Section 3.5).

	NYT-sen	NYT-doc	Netflix
Average length	19	603	266
Maximum length	21,174	38,917	7,966
Total sequences	53,137,507	1,830,592	398,820
Total items	1,051,435,745	1,051,435,745	106,145,170
Distinct items	1,577,233	1,577,233	17,769
Total bytes	3,087,605,146	3,087,605,146	608,347,782

Table 3.1: *Dataset characteristics*

Our second dataset is the Netflix dataset [Bennett and Lanning (2007)], which we use to evaluate our approach for mining temporal sequences. The Netflix data contains more than 100M ratings from 480k users for around 18k movies; each rating is annotated with a timestamp. We constructed a temporal database from this data by creating a temporal sequence for each user; this sequence consists of (timestamp, movie)-pairs ordered by timestamp. Since the Netflix dataset contains a few heavy-raters, with up to 5,500 ratings on a single day, we exclude these users from our dataset to ensure a meaningful output and keep runtimes manageable.

Measures

In the following experiments, we report the performance measure as total time elapsed between launching a task and receiving the final result. For our experiments on temporal sequence mining and mining long input sequences, we break down this time into time taken by the map phase, shuffle phase, and reduce phase. Since these phases overlap in a MapReduce job, we report the time elapsed until finishing of each phase.

3.6.2 Results

A. Effectiveness of PSM algorithm

We evaluated the efficiency of our PSM algorithm by running MG-FSM on the NYT dataset with 3 different parameter settings of increasing difficulty w.r.t the output size. We report the total runtime in Figure 3.2. We observed an overall speedup of 2× to 5× which stems from using the PSM algorithm for mining partitions where as MG-FSM uses standard BFS approach for mining each partition. For more analysis on the PSM algorithm, we refer to Section 5.5.2 in which we also compare with DFS approach and additionally consider hierarchy constraints.

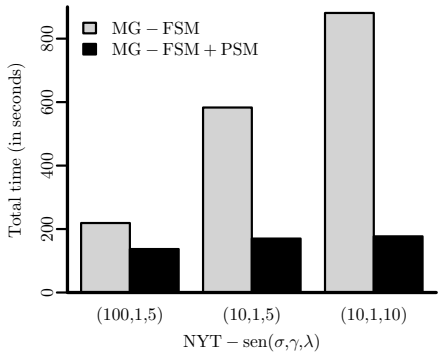
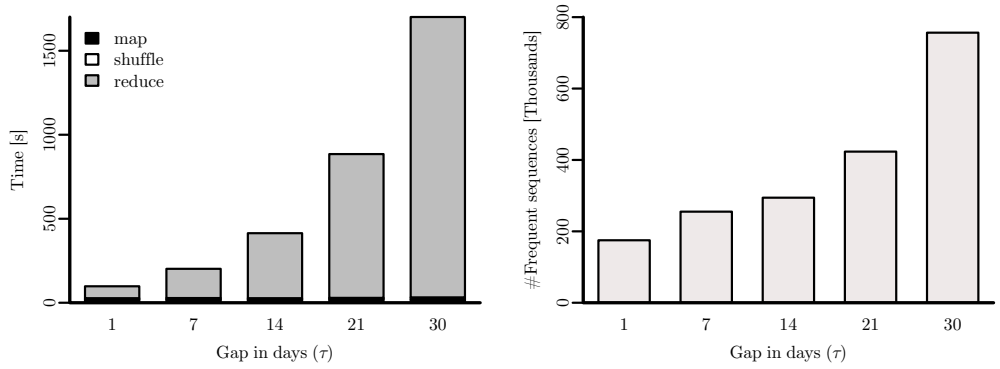


Figure 3.2: *Effectiveness of PSM*



(a) Netflix (runtimes)

(b) Netflix (sequences)

Figure 3.3: *Temporal sequences*

Sequence of movie titles (frequency)
"Men in Black II", "Independence Day", "I, Robot" (2,268)
"Pulp Fiction","Fight Club" (7,406)
"LOTR: The Fellowship of the Ring", "LOTR: The Two Towers" (19,303)
"The Patriot", "Men of Honor" (28,710)
"Con Air", "The Rock" (29,749)
"'Pretty Woman", "Miss Congeniality" (30,036)

Table 3.2: *Example frequent sequences from Netflix ($\sigma = 1000, \lambda = 5, \tau = 1$ day)*

B. Mining temporal sequences

We evaluated our approach for mining temporal sequences using the Netflix dataset Bennett and Lanning (2007). We extracted temporal sequences of movies capturing the order in which these movies were rated by users. Mining frequent sequences in this context yields sequences of movies reflecting the chronological order in which a user viewed or rated them.

We mined frequent sequences from this dataset for $\sigma = 1000, \lambda = 5$ and temporal gaps of 1, 7, 14, 21 and 30 days. The results are shown in Figure 3.3a and 3.3b. Figure 3.3a depicts the runtimes as we increase the temporal gap (τ) from 1 day (which corresponds to $\gamma = 297$) to 30 days (corresponds to $\gamma = 6068$). Figure 3.3b shows the total size of the result, i.e., how many frequent sequences were mined. Frequent sequences of user rating events within a 1-day time span were mined in 98s and were 175,003 in total. When the temporal gap was increased to 30 days (1-month time span), we mined 756,528 frequent sequences (a 4.32x increase) while total runtime had a significant 17x increase again due to the large number of candidate 2-sequences constructed by the mining algorithm.

Table 3.2 includes some example sequences of movies mined from the Netflix dataset. We can see that this includes movies from a trilogy in chronological order (see sequence (3), which consists of movies from the "Lord of the Rings" trilogy) and movies with the same actor (see sequence (1), which consists of movies starring actor Will Smith).

C. Mining long input sequences

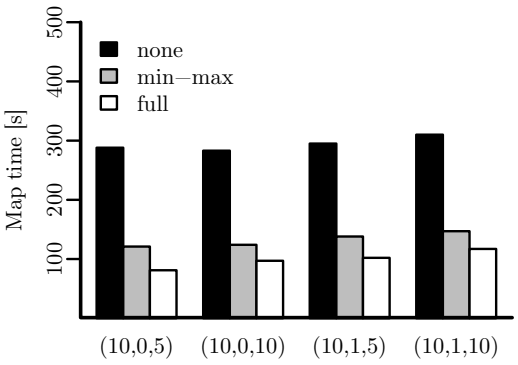
In this group of experiments, we studied the performance of the indexing techniques of Section 3.5 for long sequences. We used the NYT-doc dataset, in which each input sequence corresponds to an entire document. The average sequence length was 603 items; see Table 3.1. We evaluated MG-FSM without indexing (termed *none*), with

an index of the first and last position of each distinct item (*min-max*), as well as a full index of all positions (*full*). Recall that the goal of using indexes is to reduce cost of rewriting (map phase of MG-FSM).

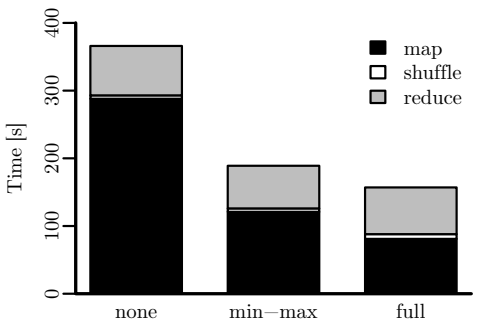
We first compared how the map time (i.e., the time until the last map tasks finished) was affected when the different kinds of indexes are used. We considered four configurations and the results are shown in Figure 3.4a. The benefit of the different indexes across different setups is similar: map time is mainly affected by the input, which remains the same, and is less sensitive to the parameters γ and λ. In all cases, the use of the min-max index reduced the total map time by more than half. When the full index was used, runtime was improved even more to 93s for setting ($\sigma = 10, \gamma = 0, \lambda = 10$) compared to 124s when the min-max index is used and 283s when no index is used. We also show the total runtimes, including reduce time, for two different settings, ($\sigma = 10, \gamma = 0, \lambda = 5$) and ($\sigma = 10, \gamma = 1, \lambda = 5$) in Figures 3.4b and 3.4c. In the easier setting, where $\gamma = 0$, the effect of using an index is large since map time corresponds to a large portion of the total time, which is reduced from 366s to 157s when the full index is used. Setting γ to 1 increases the reduce time, i.e., mining takes longer. The total runtime is reduced from 907s (with no index) to 638s (with full index).

Our experiments also show that different length distributions of input sequences affect both map and reduce times. Recall that NYT-sen and NYT-doc contain the same data, but differ in sequence length (and number of sequences). The runtimes for $\sigma = 10$, $\gamma = 1$ and $\lambda = 5$ for NYT-sen and NYT-doc are shown in Figures 3.2 and 3.4c, respectively. When no indexing was used, the overall runtime of the map phase was more than 2x larger on NYT-doc than on NYT-sen; i.e., longer sequences translate to larger map times. With the full index, however, the map time of NYT-sen and NYT-doc were almost identical (108s vs. 102s, respectively). Note that the total mining time (i.e., including the reduce phase) is not comparable because outputs are different.

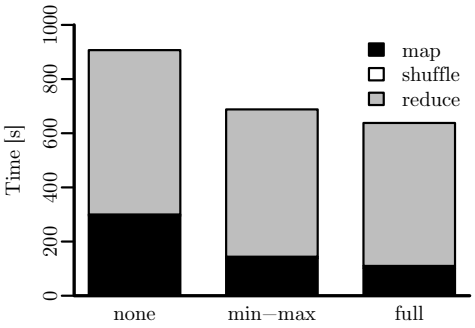
We also studied whether the use of indexing improves performance when the input sequences are short. Using the NYT-sen dataset with short sequences (each sentence corresponds to a different input sequence and the average length is 19), we observed that the construction and maintenance of the index was slower than just scanning repeatedly the input sequences. When sequences are short, we need to perform only a few, cheap scans so that index construction is not beneficial.



(a) NYT-doc (map time only)



(b) NYT-doc ($\sigma = 10, \gamma = 0, \lambda = 5$)



(c) NYT-doc ($\sigma = 10, \gamma = 1, \lambda = 5$)

Figure 3.4: *Effectiveness of indexing long sequences*

3.7 Related Work

We now relate the ideas put forward in this chapter to existing prior work. Prior approaches can be coarsely categorized with respect to the type of pattern being mined (frequent sequences or application-specific special cases such as n-grams) and according to their parallelization (sequential, shared-memory parallel, shared-nothing parallel, or MapReduce).

There are several extensions to basic sequential algorithms discussed in Chapter 2 to handle constraints. GSP [Srikant and Agrawal (1996)] introduced length, gap, temporal gaps as well as hierarchy constraints (we will discuss hierarchy constraints in Chapter 5). Zaki (2000) proposed the cSPADE algorithm, which extends SPADE to handle length, gap, and temporal constraints. Pei et al. (2002) investigated how constraints can be adapted in pattern-growth approach. Our adaption of BFS and DFS approaches to handle length- and gap constraints emulates these algorithms. Zaki (2000) and Giannotti et al. (2006) considered variations of the temporal sequence mining problem discussed in this chapter in which, the former aims to mine frequent sequences constrained to user specified time window and the later aims to discover frequent sequences of sufficiently close time-annotated events. For the specific case of Web access mining, Pei et al. (2000) proposed WAP-miner for mining Web logs, which uses a WAP-tree data structure to compactly represent the sequence database and employs suffix based tree-projection mechanism to grow sequences. Frequent episode mining [Mannila et al. (1997)], a related but different problem, determines sequences that occur frequently within a single long sequence.

Parallel approaches to frequent sequence mining have been proposed for different machine models. Zaki (2001a) proposed pSPADE, which extends the SPADE algorithm to shared memory parallel architecture. In his approach, Zaki investigated how posting list intersections can be computed in parallel either via performing a single intersection in parallel or performing local intersection in parallel. This approach however led to poor performance due high amount of synchronization required between processes. Zaki also proposed search space partitioning based on suffix-based equivalence classes in which each process work on separate classes. Guralnik et al. (2001) and Guralnik and Karypis (2004) examined how the projection-based pattern-growth approach from Agarwal et al. (2001), which is similar to PrefixSpan, can be parallelized by distributing data and/or work among machines.

Given the important role of n-grams in natural language processing and information retrieval, several solutions exist for this specific special case of frequent sequence mining. SRILM [Stolcke (2002)] is one of the best-known toolkits to compute and work with n-gram statistics for document collections of modest size. Brants et al. (2007) describe how large-scale statistical language models are trained at Google. To compute counts of n-grams having length five or less, they use a simple extension of WordCount in MapReduce. Huston et al. (2011) develop distributed methods to build an inverted index for n-grams that occur more than once in the document

collection. Berberich and Bedathur (2013) described Suffix-σ, which partitions the input data based on suffixes and runs in a single MapReduce job.

To the best of our knowledge, none of the existing work provides a satisfactory solution to general frequent sequence mining in MapReduce. Perhaps closest to this work is the work on parallel itemset mining of Buehrer et al. (2007); Guralnik and Karypis (2004), which also makes use of item-based partitioning of the output space. In contrast to MG-FSM, these methods use database projections tailored to itemset mining; these projections cannot be used for frequent sequence mining and are generally less flexible than MG-FSM's partition construction techniques.

3.8 Summary

MG-FSM is a scalable algorithm for length- and gap-constrained sequence mining and uses the BFS approach in the local mining phase. We showed that using a customized miner like PSM, which is aware of the MG-FSM framework, leads to higher overall efficiency. We also proposed various extensions to MG-FSM for special application scenarios. First we discussed methods to support temporal sequence mining, in which items are annotated with timestamps. This allows MG-FSM to handle temporal gaps (for session analysis). Second, we showed how MG-FSM can efficiently handle datasets in which input sequences are very long.

Chapter 4

Maximality and Closedness Constraints

Frequent sequence mining from large datasets can potentially generate a large number of sequences, especially when the support threshold is low and the length parameter is large. A standard approach [Yan et al. (2003); Fournier-Viger et al. (2013)] to reduce the number of mined sequences without losing information is to output only sequences that are maximal or closed. Such sequences compactly represent the set of all frequent sequences along with their exact frequency (closed sequences) or a lower bound thereof (maximal sequences). In this chapter,[a] we show how MG-FSM can be adapted to mine maximal or closed sequences in a scalable fashion.

4.1 Definitions

The key motivation behind maximal and closed sequence mining is that knowing that a sequence S' is frequent also provides us with information about whether or not certain subsequences of S' are frequent. In more detail, set

$$\gamma^- = \begin{cases} 0 & \text{if } \gamma < \infty \\ \infty & \text{if } \gamma = \infty. \end{cases}$$

The following lemma describes the relationship between the frequency of a sequence S' and the frequency of (some of) its subsequences.

[a] The material in this chapter is based on Beedkar et al. (2015).

Lemma 4.1 (Support monotonicity). *Let S and S' be two sequences such that $S \subseteq_{\gamma^-} S'$. For all sequence databases $\mathscr{D}$, we have*

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

Proof. Consider any input sequence $T = t_1 \dots t_n \in \mathscr{D}$ such that $S' \subseteq_\gamma T$. We need to show that this implies $S \subseteq_\gamma T$. Since $S' \subseteq_\gamma T$, there is a set of indexes $i'_1 < \dots < i'_{|S'|}$ such that (i) $s_k = t_{i'_k}$ for $1 \le k \le |S'|$, and (ii) $i'_{k+1} - i'_k - 1 \le \gamma$ for $1 \le k < |S'|$. Furthermore, since $S \subseteq_{\gamma^-} S'$, there is a set of indexes $j_1 < \dots < j_{|S|}$ such that (i) $s_k = s'_{j_k}$ for $1 \le k \le |S|$, and (ii) $j_{k+1} - j_k - 1 \le \gamma^-$ for $1 \le k < |S|$. Now consider the set of indexes $i_k = i'_{j_k}$. We have $S = t_{i_1} \cdots t_{i_{|S|}}$ by construction so that S is a ∞-subsequence of T; this proves the lemma for $\gamma = \infty$. To finish the proof, observe that when $\gamma < \infty$, we have $\gamma^- = 0$ so that $j_{k+1} = j_k + 1$ and therefore $i_{k+1} - i_k - 1 = i'_{j_{k+1}} - i'_{j_k} - 1 = i'_{j_k+1} - i'_{j_k} - 1 \le \gamma$ so that $S \subseteq_\gamma T$. □

It directly follows that $f_\gamma(S, \mathscr{D}) \ge f_\gamma(S', \mathscr{D})$. Thus, if S' is frequent and S is a γ^--subsequence of S', then S must also be frequent. In particular, if $\gamma = \infty$, every subsequence of S' is frequent. If $\gamma < \infty$, the consecutive subsequences of S' are frequent.

Note that we carefully distinguished the cases $\gamma = \infty$ and $\gamma < \infty$. The reason is, perhaps contrary to intuition, that not every subsequence of a frequent sequence is necessarily frequent as well. To see this, consider the database

$$\mathscr{D}_{\mathrm{ex}} = \{\, abc, abec, abcd, abcfd \,\}, \tag{4.1}$$

and its set of $(2, 1, 4)$-frequent sequences:

$$F_{2,1,4}(\mathscr{D}_{\mathrm{ex}}) = \{a(4), b(4), c(4), d(2), ab(4), ac(4), bc(4), cd(2), \\ abc(4), acd(2), bcd(2), abcd(2)\}.$$

Here we also provide frequencies (which are not formally part of $F_{2,1,4}(\mathscr{D}_{\mathrm{ex}})$). Observe that sequence $S = abcd$ is frequent whereas some of its subsequences are not. In particular, sequence $bd \subseteq_1 S$ is not frequent, even though bd is a γ-subsequence (but not a γ^--subsequence) of S. As asserted by Lemma 4.1, all *consecutive* subsequences of S are indeed frequent (a, ab, abc, …).

Consider a frequent sequence $S' \in F_{\sigma,\gamma,\lambda}(\mathscr{D})$ of length l. The above lemma implies that when $\gamma = \infty$, each of the $2^l - 1$ non-empty subsequences of S' are also frequent. Similarly, when $\gamma < \infty$, each of the $l(l+1)/2$ non-empty consecutive subsequences of S are frequent. The goal of mining maximal sequences is to avoid outputting these "redundant" sequences. In particular, a sequence is maximal if and only if it is not redundant:

Definition 4.1 (Maximality). *A sequence S is $(\sigma, \gamma, \lambda)$-*maximal *if S is $(\sigma, \gamma, \lambda)$-frequent and there is no sequence $S' \supset_{\gamma^-} S$ which is also $(\sigma, \gamma, \lambda)$-frequent. The set*

of $(\sigma, \gamma, \lambda)$*-maximal sequences is given by*

$$F^{max}_{\sigma,\gamma,\lambda}(\mathscr{D}) = \{\, S \in F_{\sigma,\gamma,\lambda}(\mathscr{D}) \mid \neg\exists S' \in F_{\sigma,\gamma,\lambda}(\mathscr{D}) : S \subset_{\gamma^-} S' \,\}.$$

For our running example, we obtain

$$\begin{aligned} F^{\max}_{2,\infty,4}(\mathscr{D}_{\text{ex}}) &= \{\, abcd(2) \,\}, \\ F^{\max}_{2,1,4}(\mathscr{D}_{\text{ex}}) &= \{\, acd(2), abcd(2) \,\}. \end{aligned} \tag{4.2}$$

As alluded to above, we can reconstruct the set of all frequent sequences from the set of maximal sequences:

$$F_{\sigma,\gamma,\lambda}(\mathscr{D}) = \{S \mid S \subseteq_{\gamma^-} S', S' \in F^{\max}_{\sigma,\gamma,\lambda}(\mathscr{D})\}.$$

This is true because (1) every γ^--subsequence of a maximal sequence must be $(\sigma, \gamma, \lambda)$-frequent and (2) every $(\sigma, \gamma, \lambda)$-frequent sequence must be a γ^--subsequence of some maximal sequence. (1) holds by Lemma 4.1, (2) holds by definition of $F^{\max}_{\sigma,\gamma,\lambda}(\mathscr{D})$.

A similar reasoning can be applied to closed sequences. Here we want to be able to reconstruct from the set of closed sequences the set of frequent sequences *along with their frequencies.* The following notion of closedness allows for such reconstruction.

Definition 4.2 (Closedness)**.** *A sequence* S *is* $(\sigma, \gamma, \lambda)$-closed *if* S *is* $(\sigma, \gamma, \lambda)$*-frequent and there is no* $(\sigma, \gamma, \lambda)$*-frequent sequence* $S' \supset_{\gamma^-} S$ *of the same frequency, i.e., with* $f_\gamma(S', \mathscr{D}) = f_\gamma(S, \mathscr{D})$*. The set of* $(\sigma, \gamma, \lambda)$*-closed sequences is given by*

$$F^{closed}_{\sigma,\gamma,\lambda}(\mathscr{D}) = \{\, S \in F_{\sigma,\gamma,\lambda}(\mathscr{D}) \mid \neg\exists S' \in F_{\sigma,\gamma,\lambda}(\mathscr{D}) : S \subset_{\gamma^-} S' \wedge f_\gamma(S, \mathscr{D}) = f_\gamma(S', \mathscr{D}) \,\}.$$

For our running example, we obtain

$$\begin{aligned} F^{\text{closed}}_{2,\infty,4}(\mathscr{D}_{\text{ex}}) &= \{\, abcd(2), abc(4) \,\}, \\ F^{\text{closed}}_{2,1,4}(\mathscr{D}_{\text{ex}}) &= \{\, ac(3), abc(4), acd(2), abcd(2) \,\}. \end{aligned} \tag{4.3}$$

Ignoring frequencies, observe that $F^{\max}_{\sigma,\gamma,\lambda}(\mathscr{D}) \subseteq F^{\text{closed}}_{\sigma,\gamma,\lambda}(\mathscr{D}) \subseteq F_{\sigma,\gamma,\lambda}(\mathscr{D})$ so that reconstruction of frequent sequences is still possible. The frequency of a reconstructed sequence is the maximum of the frequencies of its closed γ-supersequences. For example,

$$f_1(bc) = \max\{\, f_1(abc), f_1(abcd) \,\} = \max\{\, 4, 2 \,\} = 4.$$

4.2 Mining Maximal Sequences

One way to adapt MG-FSM to mine maximal sequences is to first compute the set of all frequent sequences $F_{\sigma,\gamma,\lambda}(\mathscr{D})$ and subsequently filter out sequences that are not maximal. This approach is not efficient, however: First, we mine too many sequences

from each partition (i.e., sequences that cannot possibly be maximal). Second, a naïve approach for the subsequent filtering step takes $O(|F_{\sigma,\gamma,\lambda}(\mathscr{D})|^2)$ time. In what follows, we propose a more suitable approach which integrates the maximality constraint directly into MG-FSM. We refer to this adaptation as MG-FSM$^+$.

Recall that MG-FSM creates one partition $\mathscr{P}_w$ for each item w. Let S be a pivot sequence for $\mathscr{P}_w$; i.e., $p(S) = w$ and $2 \leq |S| \leq \lambda$. Then MG-FSM guarantees that $f_\gamma(S, \mathscr{P}_w) = f_\gamma(S, \mathscr{D})$. Now suppose that S is frequent so that MG-FSM outputs it when mining $\mathscr{P}_w$. Ideally, we would like MG-FSM$^+$ to output S if and only if it is also maximal. Unfortunately, we cannot test for maximality locally in each partition because the frequencies $f_\gamma(S', \mathscr{P}_w)$ of supersequences $S' \sqsupset_\gamma S$ may not (and usually will not) coincide with the corpus frequency $f_\gamma(S', \mathscr{D})$ when $p(S') \neq w$. To see this, fix σ, γ, and λ and denote the output of MG-FSM at partition $\mathscr{P}_w$ by

$$F_w(\mathscr{P}_w) = \{\, S \in F_{\sigma,\gamma,\lambda}(\mathscr{D}) \mid p(S) = w \;\wedge\; S \neq w \,\}.$$

For our example database $\mathscr{D}_{\text{ex}}$ of Equation (4.1) and $\sigma = 2$, $\gamma = 1$ and $\lambda = 4$, we have

$$F_c(\mathscr{P}_c) = \{\, ac(4), bc(4), abc(4) \,\}.$$

Now consider frequent sequence $S = abc \in F_c(\mathscr{P}_c)$. Sequence S is not maximal since sequence $S' = abcd \sqsupset_0 S$ is frequent in $\mathscr{D}_{\text{ex}}$. However, $S' \notin F_c(\mathscr{P}_c)$ so that we cannot decide locally at $\mathscr{P}_c$ whether or not S is maximal.

A key ingredient to MG-FSM$^+$ is to test for *local maximality* and output in each partition only those frequent sequences that are locally maximal. Our local-maximality test exploits that whenever a sequence is not locally maximal, then it is also not (globally) maximal; we thus do not incorrectly filter out maximal sequences. The set of locally maximal sequences at partition $\mathscr{P}_w$ is given by:

Definition 4.3 (Local and global maximality)**.** *A sequence S with $p(S) = w$ is* locally maximal *if $S \in F_w^{max}(\mathscr{P}_w)$, where*

$$F_w^{max}(\mathscr{P}_w) = \{\, S \in F_w(\mathscr{P}_w) \mid \neg\exists S' \in F_w(\mathscr{P}_w) : S \sqsubset_\gamma S' \,\}.$$

Sequence S is globally maximal *if $S \in F_{\sigma,\gamma,\lambda}^{max}(\mathscr{D})$.*

Thus a sequence S with pivot $w = p(S)$ is locally maximal if it is maximal with respect to the output $F_w(\mathscr{P}_w)$ at the partition $\mathscr{P}_w$ that mines S. Stated differently, a sequence S is locally maximal if and only if S is frequent, $|S| \geq 2$, and there is no frequent sequence $S' \sqsupset_\gamma S$ with the same pivot item (i.e., $p(S') = p(S)$). For our running example, we obtain

$$F_c^{\max}(\mathscr{P}_c) = \{\, ac(4), abc(4) \,\}.$$

Observe that $bc \in F_c(\mathscr{P}_c)$ but, since $abc \in F_c(\mathscr{P}_c)$, $bc \notin F_c^{\max}(\mathscr{P}_c)$. Also observe that bc is indeed not maximal. Figure 4.1 shows the set of locally maximal sequences for each of the partitions obtained for our example database. The following lemma asserts that we can safely filter out sequences that are not locally maximal.

Lemma 4.2. *Every globally maximal sequence S with $|S| \geq 2$ is also locally maximal.*

Proof. Let S with $|S| \geq 2$ be globally maximal and set $w = p(S)$. We have $f(S, \mathscr{D}) \geq \sigma$ and, since $\mathscr{D}$ and $\mathscr{P}_w$ are w-equivalent, $S \in F_w(\mathscr{P}_w)$. We have to show that $S \in F_w^{\max}(\mathscr{P}_w)$ as well. Suppose to the contrary that $S \notin F_w^{\max}(\mathscr{P}_w)$. By the definition of local maximality, there must be a sequence $S' \supset_{\gamma^-} S$ with $S' \in F_w(\mathscr{P}_w)$. Since $S' \in F_w(\mathscr{P}_w)$, we have $p(S') = w$, $|S'| \geq 2$ and $f(S'; \mathscr{P}_w) \geq \sigma$. Since furthermore $\mathscr{D}$ and $\mathscr{P}_w$ are w-equivalent, it follows that $f(S'; \mathscr{P}_w) = f(S'; \mathscr{D})$ and thus $f(S'; \mathscr{D}) \geq \sigma$. But then $S' \in F_{\sigma,\gamma,\lambda}(\mathscr{D})$ so that S cannot be globally maximal, a contradiction. □

Note that the opposite does not necessarily hold, i.e., there can exist a sequence S that is locally maximal but not globally maximal. This happens when (1) there is no frequent sequence $S' \supset_{\gamma^-} S$ with $p(S') = w$ but (2) there is a frequent sequence $S'' \supset_{\gamma^-} S$ with $p(S'') > w$. Note that $p(S'') \geq w$ for all $S'' \supseteq S$. Here (1) implies that S is locally maximal and (2) implies that S is not globally maximal. We refer to such sequences as *spurious sequences*. In $F_c^{\max}(\mathscr{P}_c)$ shown above, all sequences are spurious (since both acd and $abcd$ are frequent).

MG-FSM$^+$, which is given as Algorithm 4.1, is divided into two steps, each corresponding to a MapReduce job.

1) Mine and output the set $F_w^{\max}(\mathscr{P}_w)$ of locally maximal sequences for each partition $\mathscr{P}_w$; this step is similar to MG-FSM. A straightforward approach to obtain $F_w^{\max}(\mathscr{P}_w)$ for each partition $\mathscr{P}_w$ is to first compute $F_w(\mathscr{P}_w)$ and then test whether each so-obtained sequence is locally maximal. A more efficient alternative, which we use in MG-FSM$^+$, is to directly mine locally maximal sequences instead. To do so, we can use any suitable maximal sequence miner; e.g, the algorithm of Fournier-Viger et al. (2013).[b]
2) Determine the set of globally maximal sequences by identifying and eliminating all spurious sequences.

The algorithm is illustrated in Figure 4.1.

In the remainder of this section, we discuss an efficient technique for pruning spurious sequences. Let S^+ be a locally maximal sequence with $p(S^+) = w^+$; i.e., $S^+ \in F_{w^+}^{\max}(\mathscr{P}_{w^+})$. Furthermore, let S be a γ^--subsequence of S^+ and set $w = p(S)$. The key idea of our approach is as follows: If S is locally maximal, then S^+ "proves" that S is spurious; we refer to such an S^+ as a *witness* for the spuriousness of S. In our running example, $S = ac$ is a spurious sequence at partition $\mathscr{P}_c$; sequence $S^+ = acd$ from partition $\mathscr{P}_d$ is its witness (see also Figure 4.1). Note that a spurious sequence can have more than one witness.

[b] This approach is valid if we ensure that $p(T) \leq w$ for all $T \in \mathscr{P}_w$. Since during rewriting (Map1, Section 3.2.3), we replace all irrelevant items (i.e., items $> w$) by blanks, this property holds.

Algorithm 4.1 The MG-FSM$^+$ algorithm

Require: Sequence database $\mathscr{D}$, σ, γ, λ, f-list $F_{\sigma,0,1}(\mathscr{D})$, type $\in$ { max, closed }

Map1(T):
Same as Map(T) in Algorithm 3.1; removal of irrelevant items required

Reduce1(w, $\mathscr{P}_w$):
$F^{\text{type}}_{\sigma,\gamma,\lambda}(\mathscr{P}_w) \leftarrow \text{FSM}^{\text{type}}_{\sigma,\gamma,\lambda}(\mathscr{P}_w)$
for all $S \in F^{\text{type}}_{\sigma,\gamma,\lambda}(\mathscr{P}_w)$ **do**
  if $p(S) = w$ and $S \neq w$ **then**
    Output $(S,\ f_\gamma(S, \mathscr{P}_w))$
  end if
end for

Map2(S^+, $f_\gamma(S^+, \mathscr{D})$): // *where* $S^+ \in F_{\sigma,0,1}(\mathscr{D}) \cup \bigcup_w F^{type}_w(\mathscr{P}_w)$
$f^+ \leftarrow f_\gamma(S^+, \mathscr{D})$
$l^+ \leftarrow |S^+|$
for all $S \in W_\gamma(S^+)$ **do**
  Output $(S, \langle l^+, f^+ \rangle)$
end for

Reduce2(S, { $\langle l, f \rangle$ }):
switch (type)
  case max**:**
    $\langle l^*, f^* \rangle \leftarrow$ pair in { $\langle l, f \rangle$ } having maximum length l
  case closed**:**
    $\langle l^*, f^* \rangle \leftarrow$ pair in { $\langle l, f \rangle$ } having highest frequency f;
    resolve ties by picking the pair with maximum length l
end switch
if $|S| = l^*$ **then**
  Output (S, f^*)
end if

To ensure the efficiency of the pruning step, we need to ensure that we find a witness for each spurious sequence *efficiently* and in parallel. MG-FSM$^+$ uses the following observation to restrict search to the set of *primary witnesses*.

Lemma 4.3. *Let S be a spurious sequence. Then there is a* primary-witness sequence *S^+, which satisfies*

1) *$S \subset_{\gamma^-} S^+$, $p(S^+) > p(S)$, and S^+ is frequent,*
2) *S^+ is locally maximal,*
3) *there is no intermediate sequence S^* with $p(S^*) < p(S^+)$ and $S \subset_{\gamma^-} S^* \subset_{\gamma^-} S^+$.*

Proof. Since S is spurious, there must be some globally maximal sequence $S' \supset_{\gamma^-} S$ satisfying (1); Lemma 4.2 implies that S' also satisfies (2). If S' satisfies (3), we take $S^+ = S'$ and are done.

Otherwise, let $w = p(S)$. Pick any sequence S' that satisfies (1) and (2); the discussion above shows that there is such a sequence. Suppose that S' contains only one distinct item w' that is larger than w. We show that S' then satisfies (3). Suppose to the contrary that S' contains a subsequence S^* satisfying $S \subset_{\gamma^-} S^* \subset_{\gamma^-} S'$ and $w^* = p(S^*) < p(S') = w'$. Since S' contains only one distinct item that is "larger" than w, we must have $w^* = w$. Since S' is frequent and by Lemma 4.1 any γ^--subsequence of a frequent sequence is itself frequent, we conclude that S^* is frequent. Putting both together, we have $S^* \in F_w(\mathscr{P}_w)$. But then S is not locally maximal and thus not a spurious sequence, a contradiction.

Now assume that S' satisfies (1) and (2) but not (3). We show that there must be a "smaller" sequence S^- such that $p(S) < p(S^-) < p(S')$ and S^- satisfies (1) and (2). If S^- also satisfies (3), we are done. If not, we iterate this process by taking the just-obtained sequence S^- for S'. Since after every iteration $p(S) < p(S^-) < p(S')$, S^- will eventually satisfy (3); by the discussion above, this happens at the latest when S^- contains only one item larger than w. The lemma thus follows.

It remains to show that S^- exists. Let $w' = p(S')$ and let S^* be any subsequence of S' violating (3), i.e., $w^* = p(S^*) < w'$ and $S \subset_{\gamma^-} S^* \subset_{\gamma^-} S'$. Using arguments as above, we find that S^* satisfies (1). If S^* also satisfies (2), set $S^- = S^*$. Otherwise, S^* is not locally maximal. But then there is a locally maximal sequence $S_2^* \in F_{w^*}(\mathscr{P}_{w^*})$ with $S^* \subset_{\gamma^-} S_2^*$. Clearly, $S \subset_{\gamma^-} S_2^*$ as well. Since additionally $p(S_2^*) = w^* > w$, we conclude that S_2^* satisfies (1) and (2), and set $S^- = S_2^*$. □

Again, there can be more than one primary witness for S. The key property exploited by MG-FSM$^+$ is (3). We provide some intuition on the assertion of the lemma here. First, in our example of Figure 4.1, *abc* (from $\mathscr{P}_c$) is a primary witness for spurious sequence *ab* (from $\mathscr{P}_b$). In contrast, even though sequence *abcd* (from $\mathscr{P}_d$) is a witness for *ab*, it is not a primary witness since it violates (3) with intermediate sequence *abc*. In general, the lemma tells us that if S is spurious, then there is primary witness sequence S^+ which contains S as a γ^--subsequence arranged in

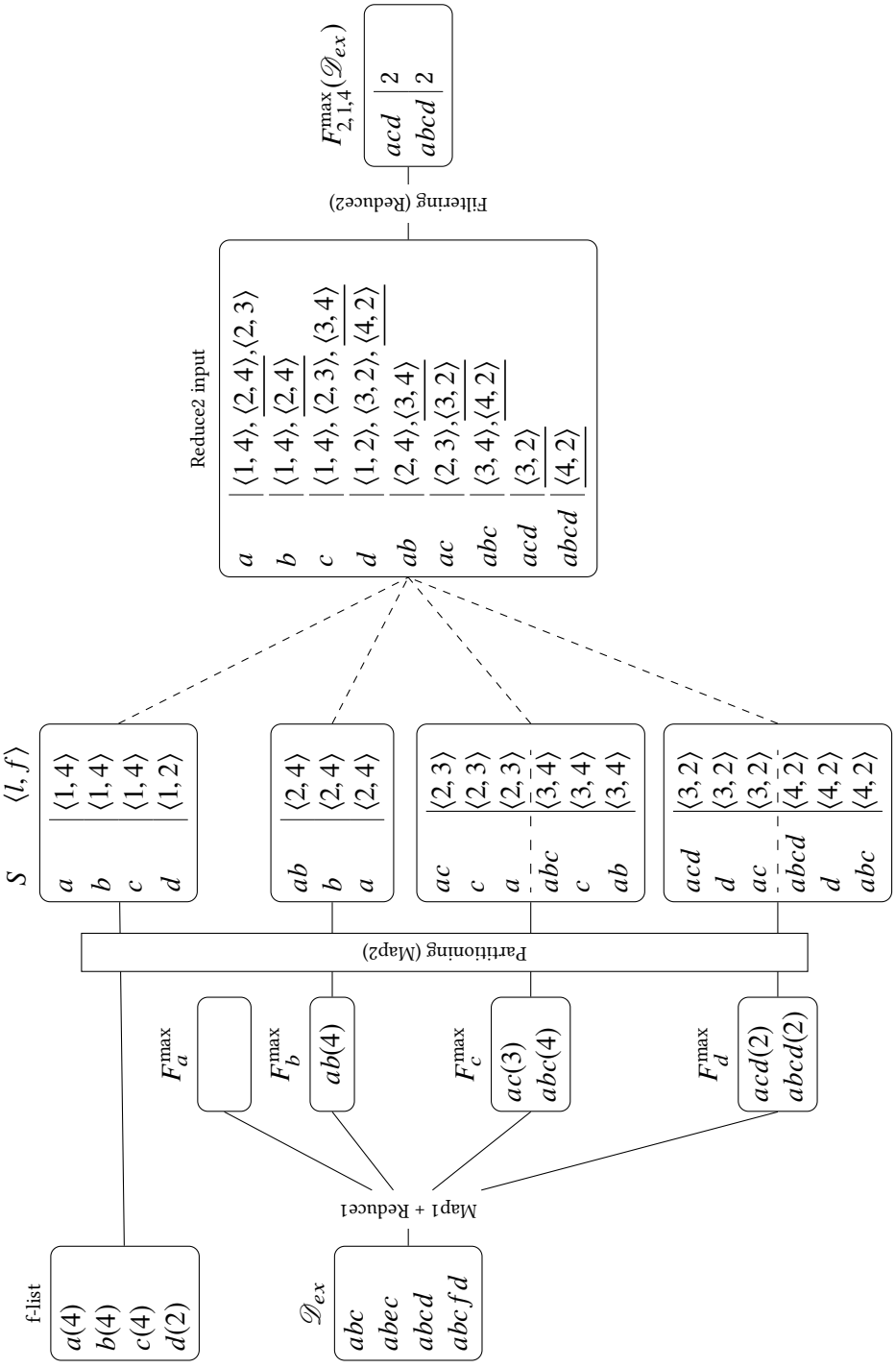


Figure 4.1: *Mining maximal sequences with MG-FSM$^+$ ($\sigma = 2$, $\gamma = 1$, $\lambda = 4$).*

a "certain way". To see how S^+ is arranged, let $w^+ = p(S^+)$. If $\gamma < \infty$, property (1) implies that S appears consecutively in S^+. The spuriousness of S along with property (3) imply that there is either pivot w^+ or the start/end of the sequence to the left and right of the occurrence of S. Continuing the above example, ab occurs consecutively in its primary witness abc and is enclosed by the start of the sequence to the left and pivot c to the right. Similarly, if $\gamma = \infty$, the spuriousness of S along with property (3) imply that S is obtained from S^+ by dropping all pivots w^+ from S^+.

We are now ready to describe the second step of MG-FSM$^+$ (Lines 12–28 of Algorithm 4.1), which removes spurious sequences. The step is divided into a partitioning phase, which matches primary witnesses with their spurious sequences, and a filtering phase, which produces the final output.

Partitioning phase (Map2).

We map over the locally maximal sequences obtained in the first step of MG-FSM$^+$ (sequences of length at least 2) as well as over the f-list (length 1). For each sequence S^+, we generate the set of sequences for which S^+ can potentially be a primary witness. By the arguments right below Lemma 4.3, there is only a small set of such sequences. In more detail, set $w^+ = p(S^+)$ and divide S^+ into non-empty *chunks* $S_1, \ldots, S_n$ by splitting at pivots, i.e.,

$$S^+ = (w^+)^* \, S_1 \, (w^+)^+ \, S_2 \, (w^+)^+ \ldots (w^+)^+ \, S_n \, (w^+)^*,$$

such that $p(S_i) < w^+$; here $(w^+)^*$ ($(w^+)^+$) denotes 0 or more (1 or more) occurrences of pivot w^+. Denote by

$$W_\gamma(S^+) = \{ S^+ \} \cup \{ w^+ \} \cup \begin{cases} \{ S_1 S_2 S_3 \ldots S_n \} & \text{if } \gamma = \infty \\ \{ S_1, S_2, S_3, \ldots, S_n \} & \text{if } \gamma < \infty, \end{cases}$$

the set of sequences for which S^+ can be a primary witness, as well as w^+ and S^+ itself. Note that we include pivot w^+ because, if $|S^+| \geq 2$, S^+ proves 1-sequence w^+ from the f-list to be spurious. For example, for sequence $S^+ = abcdbb$ with $p(S^+) = d$, we have $W_\infty(S^+) = \{ abcdbb, d, abcbb \}$ and $W_1(S^+) = \{ abcdbb, d, abc, bb \}$.

We emit a key-value pair for every sequence $S \in W_\gamma(S^+)$: the key is S, the value is fixed to the pair of length and frequency of S^+, i.e., we output $(S, \langle |S^+|, f_\gamma(S^+, \mathscr{D}) \rangle)$. Figure 4.1 shows the output of Map2 for our example database (by partition and by key). Note that only key-value pair $(S^+, \langle |S^+|, f_\gamma(S^+, \mathscr{D}) \rangle)$ has the length of the key equal to the length recorded in the value; for all other key-value pairs $(S, \langle l, f \rangle)$, we have $S \neq S^+$ and $l = |S^+| > |S|$. The total length of all emitted key-value pairs is linear in the total length of the set of locally maximal sequences.

Filtering phase (Reduce2).

The reduce phase processes independently each sequence output as a key in Map2. These sequences consist of all frequent 1-sequences (from the f-list), all locally maximal sequences, and some additional sequences contained in the set $W_\gamma(S^+)$ of some frequent sequence S^+. For each sequence S, we are given the corresponding *evidence set* $E(S) = \{ \langle l, f \rangle \}$ of (length, frequency)-pairs as input. We first determine whether or not S is a frequent 1-sequence or a locally maximal sequence. In particular, if there is no pair $\langle l_S, f_S \rangle \in E(S)$ such that $l_S = |S|$, then S is a member of some set $W_\gamma(S^+)$ but it is itself neither a frequent 1-sequence nor locally maximal. Thus we do not output S. Otherwise, there is a pair $\langle l_S, f_S \rangle \in E(S)$ such that $l_S = |S|$; this pair was produced when processing $S^+ = S$ in Map1 so that $f_s = f_\gamma(S^+, \mathscr{D}) = f_\gamma(S, \mathscr{D})$. We now need to determine whether S is globally maximal. If there is any additional pair $\langle l^+, f^+ \rangle$ in $E(S)$, then this pair must have been generated from a primary witness S^+ for S (with $S^+ \neq S$ and $S \in W_\gamma(S^+)$) and we have $l^+ > |S|$. We conclude that S is spurious. Finally, if there is no such pair, then S does not have a primary witness. Lemma 4.3 then implies that S is globally maximal so that we output (S, f_S). All of the above steps can be performed jointly as follows: Select any pair $\langle l^*, f^* \rangle$ of maximum length from $E(S)$ and output (S, f^*) if and only if $|S| = l^*$. Figure 4.1 shows $\langle l^*, f^* \rangle$ (underlined) and the output of Reduce2 for our example database. Sequences *acd* and *abcd* are the only globally maximal sequences in this example; these sequences are correctly identified by our approach.

In our implementation of the second step of MG-FSM$^+$, we further improve efficiency by making use of the combine functionality of MapReduce. In particular, our combine function mirrors Reduce2; the key difference is that we only and always output the key-value pair $(S, \langle l^*, f^* \rangle)$. Combiners thus prune length-frequency pairs that are not needed in Reduce2 so that correctness is maintained. Our use of combiners reduces the communication costs between the map and reduce phases as well as the computational cost in the reduce phase itself.

4.3 Mining Closed Sequences

MG-FSM$^+$ can also be used to mine closed sequences using a similar approach as described above; see Algorithm 4.1. Figure 4.2 shown the corresponding illustration. We outline the key differences in this section.

In the first step, we mine and output the set $F_w^{\text{closed}}(\mathscr{P}_w)$ of *locally closed* sequences in each partition $\mathscr{P}_w$ where,

$$F_w^{\text{closed}}(\mathscr{P}_w) = \{S \in F_w(\mathscr{P}_w) \mid \neg\exists S' \in F_w(\mathscr{P}_w) : S \subset_{\gamma^-} S' \\ \wedge f_\gamma(S, \mathscr{P}_w) = f_\gamma(S', \mathscr{P}_w)\}.$$

As before, we can use any closed sequence miner to obtain this set (Line 5 of Al-

gorithm 4.1). For our example database, we obtain

$$
\begin{aligned}
F_a^{\text{closed}}(\mathscr{P}_a) &= \emptyset, \\
F_b^{\text{closed}}(\mathscr{P}_b) &= \{\, ab(4) \,\}, \\
F_c^{\text{closed}}(\mathscr{P}_c) &= \{\, ac(3), abc(4) \,\} \text{ and} \\
F_d^{\text{closed}}(\mathscr{P}_d) &= \{\, acd(2), abcd(2) \,\}.
\end{aligned}
$$

These sets (coincidentally) agree with the corresponding sets of maximally closed sequences shown in Figure 4.1. (The set of globally closed sequences for our example database is given in Equation (4.3).)

In the second step, we determine *globally closed* sequences by identifying and eliminating all spurious sequences, i.e., sequences that are locally but not globally closed (ab is spurious in our example). We use a slightly different notion of witness: a locally closed sequence S^+ is a *potential witness* of the spuriousness of S if as before $S \subset_{\gamma^-} S^+$; it is a *witness* if additionally $f_\gamma(S, \mathscr{D}) = f_\gamma(S^+, \mathscr{D})$. As in the case of maximality, we can show that there must exist a primary witness for each spurious sequence; the proof is similar to Lemma 4.3 and omitted here. For example, the sequence $ab \in F_b^{closed}(\mathscr{P}_b)$ is spurious; its witness (and primary witness) is sequence $abc \in F_c^{closed}(\mathscr{P}_c)$ with $f_1(ab, \mathscr{D}_{ex}) = f_1(abc, \mathscr{D}_{ex}) = 4$. As another example, the sequence $ac \in F_c^{closed}(\mathscr{P}_c)$ is not spurious; the only potential witness is $acd \in F_d^{closed}(\mathscr{P}_d)$, but acd has incorrect frequency ($f_1(ac, \mathscr{D}_{ex}) = 3 \neq 2 = f_1(acd, \mathscr{D}_{ex})$) so that it is not a witness.

To eliminate spurious sequences, we map over the locally closed sequences as well as over the f-list exactly as in the case of maximality (Map1); i.e., we output for each sequence S^+ the set $W_\gamma(S^+)$ of sequences for which S^+ can be a primary witness. We then check in the filtering phase for each sequence S whether (1) there is a potential witness (as before) that (2) agrees in frequency with S and thus is also a witness (new). Consider a sequence S and its corresponding evidence set $E(S) = \{\, \langle l, f \rangle \,\}$. We select from $E(S)$ the pair $\langle l^*, f^* \rangle$ having highest frequency; we break ties by selecting the pair of maximum length. We then output (S, f^*) if and only if $|S| = l^*$. To see that this approach correctly determines globally closed sequences, assume that S is locally closed and spurious. Observe that each potential witness S^+ of S satisfies $f_\gamma(S^+, \mathscr{D}) \leq f_\gamma(S, \mathscr{D})$ by Lemma 4.1; equality holds if and only if S^+ is also a witness. Thus, if there is no witness for S, then S is the unique sequence of highest frequency in $E(S)$; we have $\langle l^*, f^* \rangle = \langle |S|, f_\gamma(S, \mathscr{D}) \rangle$ and thus output $(S, f_\gamma(S, \mathscr{D}))$. Otherwise, if there is a witness S^+, then $f_\gamma(S^+, \mathscr{D}) = f_\gamma(S, \mathscr{D})$ and our tie-breaking strategy applies. Since $|S^+| > |S|$, we select $\langle l^*, f^* \rangle = \langle |S^+|, f_\gamma(S^+, \mathscr{D}) \rangle$ and thus do not output S.

The input to the filtering step in our running example is shown in Figure 4.2 under "Reduce2 input" in which we also underline $\langle l^*, f^* \rangle$. First consider spurious sequence $S = ab$. We have $E(S) = \{\, \langle 2, 4 \rangle, \langle 3, 4 \rangle \,\}$ and thus select $\langle l^*, f^* \rangle = \langle 3, 4 \rangle$ (produced from $W_1(abc)$). Since $l^* = 3 \neq 2 = |S|$, we conclude that S is

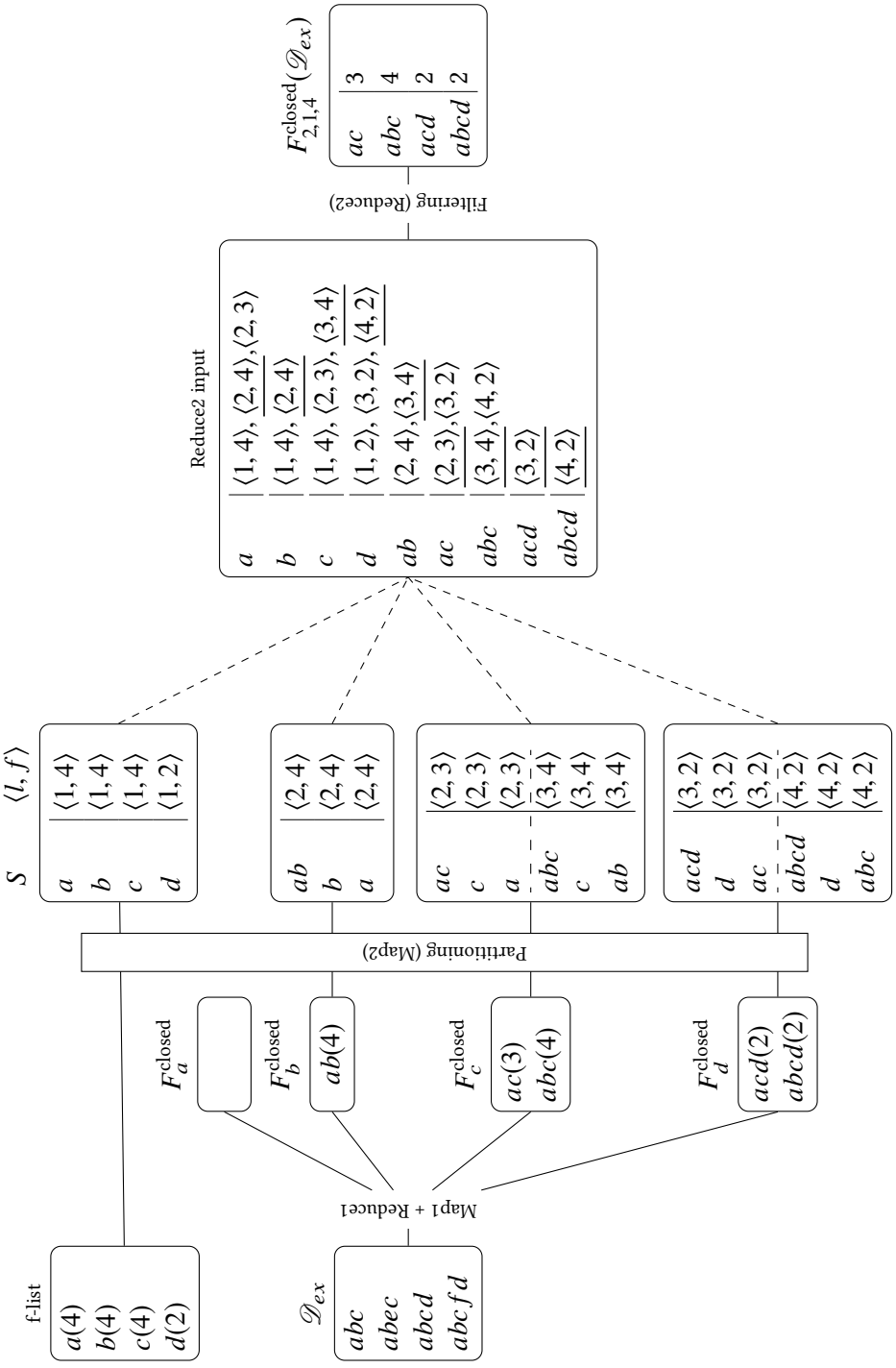


Figure 4.2: *Mining closed sequences with MG-FSM$^+$ ($\sigma = 2, \gamma = 1, \lambda = 4$).*

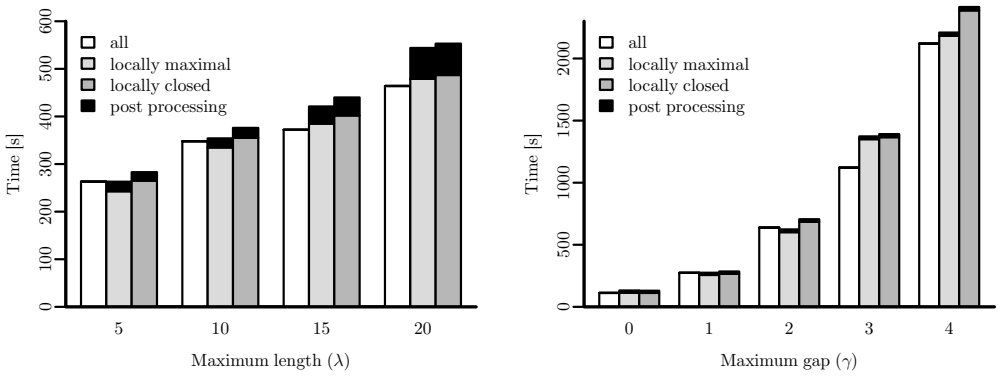
spurious. As another example, consider the globally closed sequence $S = ac$ with $E(S) = \{ \langle 2, 3\rangle, \langle 3, 2\rangle \}$. Here we select pair $\langle l^*, f^*\rangle = \langle 2, 3\rangle$ (from $W_1(S)$), which is the unique pair of highest frequency. Since $l^+ = 2 = |S|$, we conclude that S is not spurious and output S. In fact, pair $\langle 3, 2\rangle \in E(S)$ has been generated from $W_1(acd)$; we correctly identify that acd is not a witness (even though it is a potential witness). The output of Reduce2 shows the set of globally closed sequences obtained by our approach.

4.4 Experiments

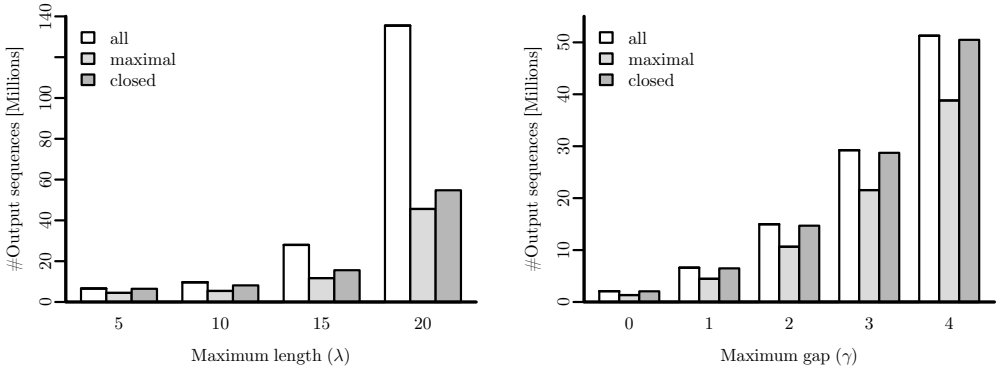
We evaluated the performance of MG-FSM$^+$ for mining maximal and closed sequences. Recall that MG-FSM$^+$, in contrast to MG-FSM, makes use of a post-processing step to filter out spurious sequences. We report separately the time required to mine locally maximal or closed sequences (first MapReduce job) and the time required for post-processing (second job). In all experiments, we used the NYT-sen dataset (see Table 3.1) and set the default values to $\sigma = 100$, $\gamma = 1$ and $\lambda = 5$.

We first studied the performance of MG-FSM$^+$ for various choices of the maximum-length parameter λ, which we vary from 5 to 20. We set $\sigma = 100$ and $\gamma = 1$. Figure 4.3a shows the total runtime for mining all sequences, maximal sequences, as well as closed sequences. Figure 4.3c shows the corresponding number of output sequences. First, observe that for large values of λ, the decrease in output size is significant (up to 3x for maximality and 2.5x for closedness); this shows that mining only maximal or closed sequences can be beneficial. Second, observe that the time required to mine maximal or closed sequences is close to the time required to mine all frequent sequences, i.e., the overhead of mining locally maximal or closed sequences as well as filtering spurious sequences in the post-processing step is low. Finally, observe that the time required for post-processing increases as we increase λ. As can be seen in Figure 4.3c, large values of λ lead to a larger output sizes in all cases. This increase in output size translates to more work in the post-processing step, which thus takes more time.

We also studied the impact of the maximum-gap parameter γ by varying its value from 0 to 4. We set $\sigma = 100$ and $\lambda = 5$. The results are shown in Figures 4.3b and 4.3d. As before, the overhead of maximal or closed sequence mining (second job) was small. For large values of γ, we observed that the time required to mine locally maximal or closed sequences (first job) was slightly larger than the time required to mine all sequences. This increase in runtime stems from our additional test for local maximality or closedness; we mine all sequences but only output the maximal and closed ones. This test took more time (up to 200s) as γ, and thus the number of sequences being processed and tested, increased. This problem is not inherent to MG-FSM$^+$: using a state-of-the-art maximal or closed sequence miner in the local mining step may reduce running time. Note that maximal and closed



(a) NYT-sen ($\sigma = 100, \gamma = 1$) **(b)** NYT-sen ($\sigma = 100, \lambda = 5$)



(c) NYT-sen ($\sigma = 100, \gamma = 1$) **(d)** NYT-sen ($\sigma = 100, \lambda = 5$)

Figure 4.3: *Performance of mining maximal and closed sequences*

sequence mining was not particularly effective in reducing the output size for our choice of $\lambda = 5$. This happens because λ was comparably small and all sequences of length λ are maximal and closed.

4.5 Related Work

To reduce the size of the frequent sequences, many studies have focused on mining closed sequences since they concisely represent the set of all frequent sequences. Adapting pattern growth approaches like PrefixSpan, Yan et al. (2003) describe the CloSpan algorithm to mine closed sequences. It maintains the set of already mined closed sequence candidates which are used to prune the search space and checks if a newly found sequence is a candidate closed sequence. This method requires post processing to prune non-closed sequences. A potential limitation of CloSpan is that it requires to maintain the set of all closed sequence candidates in memory. To this end, Wang and Han (2004) proposed the BIDE algorithm, which does not require to keep a candidate set of closed sequences. Instead, it uses a bi-directional closure checking scheme to determine if a newly generated candidate sequence is closed or not. Cong et al. (2005) describe a parallel distributed-memory variant of BIDE. Their algorithm assigns each process frequent length-1 sequence and its pseudo-projected database; each process then mines closed sequences using BIDE. Li and Wang (2008) extend the framework of BIDE to mine closed sequences with gap-constraints. Differing from the pattern growth approaches, the ClaSP algorithm Gomariz et al. (2013) exploits the vertical database format of SPADE to discover closed sequences. It traverses the lattice of all sequences in a depth-first order and uses ideas from CloSpan to generate candidate closed sequences and to prune non-closed sequences. Ideas for mining closed sequences can well be carried over to mine maximal sequences. In this direction, Fournier-Viger et al. describe the MaxSP algorithm Fournier-Viger et al. (2013) which is based on BIDE to mine maximal sequences. A later algorithm called VMSP Fournier-Viger et al. (2014), which is along the lines of ClaSP, mines maximal sequential patterns using the vertical database format. Luo and Chung (2005) describe the MSPX method which uses database samples for mining maximal sequential patterns. However, MSPX is an approximate algorithm and thus may not mine the complete set of maximal patterns.

4.6 Summary

In this chapter, we proposed the MG-FSM$^+$ algorithm, which extends MG-FSM to support mining maximal and closed gap-constrained sequences. In particular, we showed that using arbitrary maximal or closed sequence miner to mine partition results in sequences that may not be globally maximal or closed across partitions. We developed efficient pruning techniques to identify and prune such sequences.

Our experimental study indicates that the overhead of our pruning techniques is low.

Chapter 5

Hierarchy Constraints

In many applications of frequent sequence mining, the individual items of the input sequences are naturally arranged in a hierarchy. For example, the individual words in a text document can be arranged in a *syntactic hierarchy*: words (e.g., "lives") generalize to their lemmas ("live"), which in turn generalize to their respective part-of-speech tags ("verb"). Products in sequences of customer transactions also form a natural *product hierarchy*, e.g. "Canon EOS 70D" may generalize to "digital camera", which generalizes to "photography", which in turn generalizes to "electronics". As a final example, entities such as persons can be arranged in *semantic hierarchies*; e.g., "Angela Merkel" may generalize to "politician," "person," "entity." Depending on the application, the hierarchy may exhibit different properties; it may be flat or deep, or it may have low or high fan-out. Hierarchies are sometimes inherent to the application (e.g., hierarchies of directories or web pages) or they are constructed in a manual or automatic way (e.g., product hierarchies).

In this chapter,[a] we consider a generalized form of frequent sequence mining—which we refer to as *generalized sequence mining* (GSM)—in which the item hierarchies are specifically taken into account. In particular, the items in both input sequences and sequential patterns may belong to different levels of the item hierarchy. This generalization allows us to find sequences that would otherwise be hidden. For example, in the context of text mining, such patterns include generalized n-grams (the ADJ house) or typed relational patterns (PERSON lives in CITY). In both cases, the patterns do not actually occur in various non-generalized form, but are useful for language modeling [Jang and Mostow (2012); Lin et al. (2012); Wang and Vergyri (2006)] or information extraction tasks [Anh and Gertz (2012); Nakashole et al. (2011,

[a] The material in this chapter is based on Beedkar and Gemulla (2015).

2012)]. Hierarchies can also be exploited when mining market-basket data [Srikant and Agrawal (1996)]—e.g., users may first buy some camera, then some photography book, and finally some flash—or in the context of web-usage mining [Hollink et al. (2013); Liao et al. (2011)].

The problem of mining frequent sequences with hierarchies was introduced by Srikant and Agrawal (1996)], in which they extended the GSP algorithm to deal with hierarchies. The extended algorithm takes as input sequences of *itemsets* (as opposed to sequences of items). The hierarchy is then encoded into itemsets by replacing each item ("lives") by an itemset consisting of the item and its parents ({"lives", "live", "VERB"}); pruning or post-processing techniques are used to output consistent generalized patterns. This approach is not efficient for large databases because it blows up the input data by a factor maximum depth of the hierarchy, and suffers from the repeated scans of the input data that GSP needs to make to count sequences. In fact, the problem of how to scale frequent sequence mining with hierarchies to large databases has not been studied in the literature.

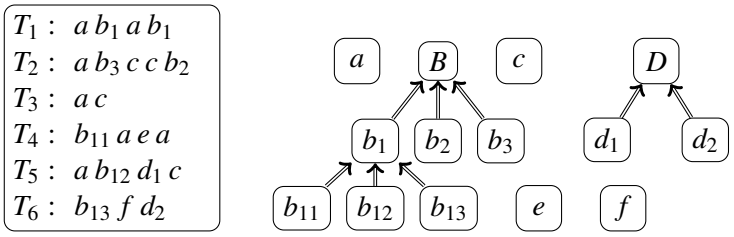
We propose LASH,[b] the first scalable, general-purpose algorithm for mining frequent sequences with hierarchies. LASH is inspired by MG-FSM in that it first partitions the data and subsequently mines each partition independently and in parallel. Key ingredients to the scalability of LASH are (i) a novel, hierarchy-aware variant of item-based partitioning, (ii) optimized partition construction techniques, and (iii) efficient sequential GSM algorithms to mine each partition. We implemented LASH using MapReduce and performed an experimental study on large real-world datasets including natural-language text and product sequences. Our results suggest that LASH has good scalability and run-time efficiency.

The remainder of this chapter is organized as follows. In Section 5.1, we formally define the problem of generalized sequence mining. In Section 5.2, we give an overview of LASH and alternative baseline algorithms. In Section 5.3, we describe the partition construction step of LASH in more detail. Algorithms for mining each partition are discussed in Section 5.4. Section 5.5 describes our experimental study and results. We discuss related work in Section 5.6 and summarize the chapter in Section 5.7.

5.1 Preliminaries

We start with a formal definition of the GSM problem and related concepts; our notation and terminology from Chapter 3 is extended accordingly.

[b] LArge-scale Sequence mining with Hierarchies



(a) Example database **(b)** Vocabulary and hierarchy

Figure 5.1: *A sequence database and its vocabulary*

Hierarchies

In GSM, the vocabulary is arranged in a hierarchy, i.e., each item has zero or more children and at most one parent.[c] Figure 5.1a shows a example sequence database $\mathscr{D}_{ex}$ and a corresponding hierarchy in Figure 5.1b that we will use as an example throughout this chapter. If an item v is an ancestor of some item u, we say that v is "more general" than u; e.g., in our example hierarchy, B is more general than b_1. We distinguish *leaf items* (most specific, no descendants), *root items* (most general, no ancestors), and *intermediate items*. In a hierarchy about music, for example, the song "Blue Monday" may be a leaf item, its parent "pop song" an intermediate item, which in turn may have as parent the root item "music". For two items $u, v \in \Sigma$, we say that u *directly generalizes* to v if u is more specialized than v; i.e., if u is a child of v in the hierarchy, which we denote by $u \Rightarrow v$. Denote by $\Rightarrow^*$ the reflexive transitive closure of $\Rightarrow$. In our example, we have $b_{11} \Rightarrow b_1$, $b_1 \Rightarrow B$ and consequently $b_{11} \Rightarrow^* B$. 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 $\text{desc}(b_1) = \{ b_1, b_{11}, b_{12}, b_{13} \}$.

Generalized sequences

We extend relation $\Rightarrow$ to sequences in a natural way. In particular, we say that sequence $T = t_1 \ldots t_n$ directly generalizes to sequence $S = s_1 \cdots s_{n'}$, denoted $T \Rightarrow S$, if $n = n'$ and there exists an index $j \in [n]$ such that $t_j \Rightarrow s_j$ and $t_i = s_i$ for $i \neq j$. In our example, sequence $T_1 = ab_1ab_1$ satisfies $T_1 \Rightarrow aBab_1$, $T_1 \Rightarrow ab_1aB$, and $T_1 \Rightarrow^* aBaB$ (the most general form of T_1). Note that we do not place any limitation on the set of items that occur in database $\mathscr{D}$; each input sequence may be composed of items from arbitrary levels of the hierarchy.

[c] In this work, we assume that the item hierarchy forms a forest. In some applications, this assumption may be violated and the hierarchy may instead form a directed acyclic graph; our methods can be extended to deal with such hierarchies as well.

Generalized subsequences

Combining generalizations and gap-constrained subsequences, we say that $S = s_1 s_2 \ldots s_{|S|}$ is a *generalized subsequence* of $T = t_1 t_2 \ldots t_{|T|}$, denoted $S \sqsubseteq_\gamma T$, if there exists integers $1 \leq i_1 < \ldots < i_{|S|} \leq |T|$ such that $t_{i_k} \Rightarrow^* s_k$ (recall that $t_{i_k} \Rightarrow^* s_k$ includes the case $t_{i_k} = s_k$) and $0 \leq i_{k+1} - i_k - 1 \leq \gamma$ for $1 \leq k \leq |S|$. For example, we have $ad_1 \sqsubseteq_1 T_5$ and $aD \sqsubseteq_1 T_5$ (even though D does not occur in T_5). Note that if S is a subsequence of T, then S is also a generalized subsequence of T; the opposite may or may not hold.

Support

Denote by

$$\mathrm{Sup}_\gamma(S, \mathscr{D}) = \left\{ T \in \mathscr{D} : S \sqsubseteq_\gamma T \right\},$$

the *support set* of sequence S in the database $\mathscr{D}$, i.e., the multiset of input sequences in which S occurs directly or in specialized form. In our example database, we have $\mathrm{Sup}_0(aBc, \mathscr{D}_{ex}) = \{ T_2 \}$ and $\mathrm{Sup}_1(aBc, \mathscr{D}_{ex}) = \{ T_2, T_5 \}$. Denote by $f_\gamma(S, \mathscr{D}) = |\mathrm{Sup}_\gamma(S, \mathscr{D})|$ the *frequency* (or *support*) of S; e.g., $f_0(aBc, \mathscr{D}_{ex}) = 1$ and $f_1(aBc, \mathscr{D}_{ex}) = 2$. We say that sequence S is *frequent* in $\mathscr{D}$ if its frequency passes a *support threshold* $\sigma > 0$, i.e., $f_\gamma(S, \mathscr{D}) \geq \sigma$.

Problem statement

> Denote by $\sigma > 0$ a minimum support threshold, by $\gamma \geq 0$ a maximum-gap constraint, and by $\lambda \geq 2$ a maximum-length constraint. The GSM problem is to find all frequent generalized sequences S, $2 \leq |S| \leq \lambda$, along with their frequencies $f_\gamma(S, \mathscr{D})$ ($\geq \sigma$).

Note that we exclude frequent items in our problem statement; these items can easily be determined (and are, in fact, also determined by our LASH algorithm). In our ongoing example and for $\sigma = 2$, $\gamma = 1$ and $\lambda = 3$, we obtain (sequence, frequency)-pairs: $(aa, 2)$, $(ab_1, 2)$, $(b_1 a, 2)$, $(aB, 3)$, $(Ba, 2)$, $(aBc, 2)$, $(Bc, 2)$, $(ac, 2)$, $(b_1 D, 2)$, and $(BD, 2)$. Observe that $b_1 D$ is frequent even though it does not occur in the database and none of its specializations are frequent. Thus GSM can detect non-obvious patterns in the data.

Discussion

The GSM problem as stated above asks for *all* sequences that frequently occur (directly or in specialized form) in the database. Depending on the dataset, the set of frequent sequences can be very large and partly redundant. In the example above, for instance, the fact that $b_1 D$ is frequent implies that BD must also be frequent. In this case, the frequencies match; in general, they can be different (e.g., aB has

Algorithm 5.1 Naïve GSM approach

Require: $\mathscr{D}$, Σ, $\Rightarrow$, σ, γ, λ

```
1: Map(T)
2: for all S ∈ G_{γ,λ}(T) do
3:     emit (S, 1)
4: end for
5:
6: Reduce(S, F = (f_1, ..., f_n))
7: f_γ(S, 𝒟) ← Σ_i f_i
8: if f_γ(S, 𝒟) ≥ σ then
9:     emit(S, f_γ(S, 𝒟))
10: end if
```

higher frequency ab_1). The potentially large number of output sequences is acceptable for applications that focus on exploration (like the Google n-Grams viewer or Netspeak) or use frequent sequences as input to further automated tasks (e.g., as features in a learning system). In some applications, the set of output sequences needs to be further restricted (e.g., using maximality or closedness constraints of the previous chapter); we do not consider such restrictions in this work.

5.2 Distributed Generalized Sequence Mining

In what follows, we first discuss a set of baseline algorithms for solving the GSM problem in a distributed fashion and describe their advantages and drawbacks. We then propose LASH, a scalable distributed algorithm that alleviates the drawbacks of the baseline approaches. All algorithms are described in terms of the MapReduce framework.

5.2.1 Naïve Approach

A naïve approach to GSM is to first generate each generalized subsequence of each input sequence and to subsequently count the global frequency of each such subsequence. This approach can be implemented in MapReduce in a way similar to "word counting" and is shown as Algorithm 5.1. In more detail, denote by

$$G_{\gamma,\lambda}(T) = \left\{ S \mid S \sqsubseteq_{\gamma} T, 1 < |S| \leq \lambda \right\}$$

the set of generalized subsequences of T that match the length and gap constraints. For example, for transaction $T_4 = b_{11}aea$ and $\gamma = 1$ and $\lambda = 3$, we obtain

$$\begin{aligned} G_{1,3}(T_4) = \{ & b_{11}a, b_{11}e, ae, aa, ea, b_{11}ae, b_{11}aa, b_{11}ea, aea, \\ & b_1a, b_1e, b_1ae, b_1aa, b_1ea, Ba, Be, Bae, Baa, Bea \}, \end{aligned}$$

Algorithm 5.2 Computing generalized f-list

Require: $\mathscr{D}, \Sigma, \Rightarrow$
1: Map(T)
2: **for all** $w \in G_1(T)$ **do**
3: emit $(w, 1)$
4: **end for**
5:
6: Reduce$(w, F = (f_1, \ldots, f_n))$
7: $f_0(w, \mathscr{D}) \leftarrow \sum_i f_i$
8: emit$(w, f_0(w, \mathscr{D}))$

where the first line lists subsequences and the second line their generalizations. To implement the naïve approach in MapReduce, we map over input sequences and, for each input sequence T, we output each element $S \in G_{\gamma,\lambda}(T)$ (as key). In the reduce function, we count for each generalized subsequence S how often it occurred in the data and output S if $f_\gamma(S, \mathscr{D}) \geq \sigma$.

The key advantage of the naïve algorithm is its simplicity. The key disadvantage, however, is that it creates excessive amounts of intermediate data and is thus generally inefficient (cf. $G_{1,3}(T_4)$ above). Denote by δ the maximum depth of the item hierarchy and set $l = |T|$. For $\gamma = 0$, naïve outputs $O(l\delta^\lambda)$ generalized subsequences per input sequence, i.e., it is exponential in λ and polynomial in δ; this number is infeasibly large in all but the most simple cases. When $\gamma, \lambda \geq l$, the situation becomes even more severe and naïve outputs $O((\delta + 1)^l)$ generalized subsequences per input sequence.

5.2.2 Semi-Naïve Approach

To reduce the number of subsequences generated by the naïve approach, we can make use of item frequencies to prune the set $G_{\gamma,\lambda}(T)$ of generated subsequences. We refer to this improvement as the semi-naïve approach.

The semi-naïve approach makes use of a *generalized f-list*, which contains each frequent item w along with its frequency $f_0(w, \mathscr{D})$. Note that the generalized f-list is hierarchy-aware, i.e., the frequency of each item $w \in \Sigma$ is given by the number of input sequences that contain w or any of its descendants. In other words, item w is *frequent* if $f_0(w, \mathscr{D}) \geq \sigma$; otherwise w is *infrequent*. For our example database and $\sigma = 2$, the generalized f-list is shown in the top-left corner of Figure 5.2 on page 68; it is also used by our LASH algorithm.

The generalized f-list can be computed efficiently in a single MapReduce job as shown in Algorithm 5.2. Denote by

$$G_1(T) = \{ w' \mid w \in T, w \Rightarrow^* w' \}$$

the set of items that appear in T along with their generalizations. For example,

$$G_1(T_4) = \{ b_{11}, a, e, a, b_1, B \} .$$

Note that $G_1(T)$ has size $O(l\delta)$, where as before $l = |T|$, and is thus linear in l and δ. To obtain the generalized f-list, we map over each $T \in \mathscr{D}$ and output each item in $G_1(T)$ along with an associated frequency of 1. The reducer sums up the frequencies for each item w to obtain $f_0(w, \mathscr{D})$.

The semi-naïve algorithm computes the set of frequent generalized sequences in a second MapReduce job. It uses the generalized f-list to reduce the number of generalized subsequences emitted by the map function of the naïve algorithm; the reduce function remains unmodified and counts frequencies. The semi-naïve algorithm outputs only the subsequences $S \in G_{\gamma,\lambda}(T)$ of input sequence T that do not contain any infrequent item (see below). For example, the semi-naïve algorithms emits for transaction $T_4 = b_{11}aea$, $\gamma = 1$, and $\lambda = 3$ the generalized subsequences

$$aa, b_1a, b_1aa, Ba, Baa.$$

Compared to the set $G_{1,3}(T_4)$ output by the naïve algorithm, the output size is reduced by a factor of more than 3.

The correctness of the semi-naïve algorithm stems from the following lemma, which implies that frequent sequences cannot contain infrequent items:

Lemma 5.1 (Support monotonicity)**.** *For any pair of generalized sequences S_1 and S_2 such that $S_1 \sqsubseteq_\gamma S_2$, we have $\mathrm{Sup}_\gamma(S_1, \mathscr{D}) \supseteq \mathrm{Sup}_\gamma(S_2)$ and consequently $f_\gamma(S_1, \mathscr{D}) \geq f_\gamma(S_2, \mathscr{D})$.*

The map phase (of the second job) can be implemented efficiently by first generalizing each item of T to its closest frequent ancestor (if existent). If an item has no frequent ancestor, we replace it by a special *blank symbol*, denoted by "␣". For example, for $T_4 = b_{11}aea$ and $\sigma = 2$, we obtain $T_4' = b_1a␣a$; here a is frequent, b_{11} is infrequent but has frequent parent b_1, and e is infrequent and has no frequent ancestor. We then enumerate and emit all sequences in $G_{\gamma,\lambda}(T_4')$ that do not contain a blank symbol. As will become evident later, the generalization of infrequent items is a concept that we also make use of in LASH (although in a slightly different way).

The semi-naïve algorithm is more efficient than the naïve algorithm if many items are infrequent; i.e., when σ is set to a high value. In the worst case, however, all items are frequent and the semi-naïve algorithm reduces to the naïve algorithm (with the additional overhead of computing of the generalized f-list).

5.2.3 Overview of LASH

The key idea of our LASH algorithm is to partition the set of sequential patterns using a *hierarchy-aware variant* of item-based partitioning. LASH is inspired by the

Algorithm 5.3 Partitioning and mining phase of LASH

Require: $\mathscr{D}$, Σ, $\Rightarrow$, σ, γ, λ

```
Map(T)
for all w ∈ G_1(T) with Sup(w, D) ≥ σ do
    Construct P_w(T)
    Emit (w, P_w(T))
end for

Reduce(w, P_w)
Compute the set G_{σ,γ,λ}(w, P_w) of the locally-frequent pivot sequences
for all S ∈ G_{w,σ,γ,λ}(P_w) do
    Emit (S, f_γ(S, P_w))
end for
```

MG-FSM algorithm (Chapter 3), which uses item-based partitioning to obtain a scalable sequence mining algorithm. In contrast to MG-FSM, LASH supports hierarchies and exploits them whenever possible.

LASH creates a partition $\mathscr{P}_w$ for every frequent item $w \in \Sigma$ and then mines frequent sequences in each partition independently. We subsequently refer to item w as the *pivot item* of partition $\mathscr{P}_w$. LASH is divided into a preprocessing phase, a partitioning phase, and a mining phase.

Preprocessing

In the preprocessing phase, LASH computes the item frequencies to obtain a generalized f-list (as in Section 5.2.2) and a total order $<$ on Σ. Like MG-FSM, we use the total order determines the partitioning used in the later phases; frequent items will be "small". The key difference to MG-FSM is that we define a total order on items that is consistent with the partial order of the hierarchy. In particular, for any pair of items $w_1, w_2 \in \Sigma$, we set $w_1 < w_2$ if $f_0(w_1, \mathscr{D}) > f_0(w_2, \mathscr{D})$. Ties are handled in a *hierarchy-aware* form: if $f_0(w_1, \mathscr{D}) = f_0(w_2, \mathscr{D})$ and w_1 occurs at a higher level of the item hierarchy, we set $w_1 < w_2$; the remaining ties are broken arbitrarily. This particular order ensures that $w_2 \Rightarrow w_1$ implies $w_1 < w_2$. Figure 5.2 shows the generalized f-list of our example database for $\sigma = 2$. Here items are ordered from small to large; i.e., we have $a < B < b_1 < c < D$. Note that item frequencies and total order can be reused when LASH is run with different parameters.

Partitioning and mining phase

The partitioning and mining phases of LASH are similar to MG-FSM and are outlined in Algorithm 5.3. LASH generates a partition $\mathscr{P}_w$ for each frequent item w (note that in LASH we create partitions for items in the input and their generaliza-

tions, i.e., also for non-leaf items); in our running example, the five partitions $\mathscr{P}_a$, $\mathscr{P}_B$, $\mathscr{P}_{b_1}$, $\mathscr{P}_c$, and $\mathscr{P}_D$ are created.

The partitioning phase is carried out in the map function, which as before maps over each input sequence T. For each frequent item $w \in G_1(T)$, we construct a "rewritten" sequence $\mathscr{P}_w(T)$ and output it with reduce key w. Note that if w is frequent and one of its descendants occurs in T, we create $\mathscr{P}_w(T)$ even if $w \notin T$. A simple and correct approach to compute $\mathscr{P}_w(T)$ is to set $\mathscr{P}_w(T) = T$. A key ingredient of LASH is to use rewrites that compress T as much as possible while maintaining correctness; we discuss such rewrites in Section 5.3.

The mining phase is carried out in the reduce function. The MapReduce framework automatically constructs partitions

$$\mathscr{P}_w = \biguplus_{T \in \mathscr{D}} \{ \mathscr{P}_w(T) \} .$$

Each reduce function then runs a customized GSM algorithm on its partition $\mathscr{P}_w$; partitions are processed independently and in parallel. The GSM algorithm is provided with the parameters w, σ, γ, and λ and produces the set $G_{\sigma,\gamma,\lambda}(w, \mathscr{P}_w)$ of *locally-frequent pivot sequences* such that, for each $S \in G_{\sigma,\gamma,\lambda}(w, \mathscr{P}_w)$, S is frequent, $p(S) = w$ and $2 \leq |S| \leq \lambda$. This local mining step can be performed using an arbitrary GSM algorithm (which produces a superset of $G_{\sigma,\gamma,\lambda}(w, \mathscr{P}_w)$) followed by a filtering step. In LASH, we proceed with the more efficient hierarchy aware version of the pivot sequence miner of Section 3.3.2 that directly produces $G_{\sigma,\gamma,\lambda}(w, \mathscr{P}_w)$.

Discussion

The key difference between LASH and the naïve and semi-naïve algorithm is the use of item-based partitioning (LASH) versus the use of sequence partitioning (naïve and semi-naïve). The advantage of item-based partitioning is that the amount of data communicated from map to the reduce phase can be significantly lowered by the use of good rewrite techniques. Moreover, the reduce functions can directly leverage state-of-the-art sequential GSM algorithms; we discuss such algorithms in Section 5.4.

5.3 Partition Construction

We now discuss partition construction and, in particular, our rewrite techniques in more detail. As stated above, a simple way to construct $\mathscr{P}_w(T)$ is to set $\mathscr{P}_w(T) = T$. For our example database ($\sigma = 2$), we obtain for pivot B the partition

$$\mathscr{P}_B = \{ a\, b_1\, a\, b_1,\ a\, b_3\, c\, c\, b_2,\ b_{11}\, a\, e\, a,\ a\, b_{12}\, d_1\, c,\ b_{13}\, f\, d_2 \} \tag{5.1}$$

Using such a partitioning strategy is inefficient due to the following reasons: (1) *skew*: partitions of highly frequent items will contain many more sequences than

partitions of less frequent items, (2) *redundant computation*: a large number of duplicate sequences are mined at multiple partitions (e.g., sequence aBc will be mined in partitions $\mathscr{P}_a$, $\mathscr{P}_B$, $\mathscr{P}_{b_1}$ and $\mathscr{P}_c$ but output only in partition $\mathscr{P}_c$), and (3) *high communication cost*: each input sequence T is replicated $|G_1(T)|$ times, which results in substantial communication cost.

In what follows, we propose rewrite techniques for constructing $\mathscr{P}_w(T)$ with the aim to overcome the above mentioned shortcomings. We refer to these rewrites as *reductions* (since they ultimately reduce the length of T).

5.3.1 Generalized w-Equivalency

We first establish the notion of *generalized w-equivalency*, which is an important criterion for the correctness of LASH. In particular, LASH is guaranteed to produce correct results if for all frequent items w, partition $\mathscr{P}_w$ and database $\mathscr{D}$ are w-equivalent.

Extending our running notation, denote by

$$G_{w,\lambda}(T) = \left\{ S \mid S \sqsubseteq_\gamma T, 2 \leq |S| \leq \lambda, p(S) = w \right\} \tag{5.2}$$

the set of generalized subsequences S of T that (1) satisfy the length and gap constraints and (2) have pivot item w. Note that we often suppress the dependence of $G_{w,\lambda}(T)$ on γ for brevity. We refer to each sequence in $G_{w,\lambda}(T)$ as *pivot sequence*. For our example and for $\sigma = 2$ and $\gamma = 1$ (which we use from now on), we obtain

$$G_{b_1,2}(T_1) = \{ ab_1, b_1a, b_1b_1, b_1B, Bb_1 \}. \tag{5.3}$$

Note that $BB \notin G_{b_1,2}(T_1)$ since each pivot sequence must contain at least one pivot (and $p(BB) = B \neq b_1$).

We say that two sequences T and T' are w-equivalent, if

$$G_{w,\lambda}(T) = G_{w,\lambda}(T'),$$

i.e., they both generate the same set of pivot sequences. For example,

$$G_{B,2}(T_2) = G_{B,2}(a\, b_3\, c\, c\, b_1) = \{ aB \} = G_{B,2}(aB).$$

LASH produces correct results if $\mathscr{P}_w(T)$ is w-equivalent to T. To see this, denote by

$$G_{w,\lambda}(\mathscr{D}) = \biguplus_{T \in \mathscr{D}} G_{w,\lambda}(T)$$

the multiset of pivot sequences generated from $\mathscr{D}$. Now observe that if $\mathscr{P}_w(T)$ is w-equivalent to T, then $G_{w,\lambda}(\mathscr{D}) = G_{w,\lambda}(\mathscr{P}_w)$; we then say that databases $\mathscr{D}$ and $\mathscr{P}_w$ are w-equivalent. Both databases then agree on the multiset of pivot sequences and, consequently, on their frequencies. Thus for every S with $p(S) = w$ and $2 \leq |S| \leq \lambda$, we have $f_\gamma(S, \mathscr{D}) = f_\gamma(S, \mathscr{P}_w)$. Since these are precisely the sequences that

LASH mines and retains from $\mathscr{P}_w$ in the mining phase, correctness follows. Note that two databases can be w-equivalent but disagree on a frequency of any non-pivot sequence; e.g., $\mathscr{D}$ and $\mathscr{P}_B$ may be B-equivalent but disagree on the frequency of B itself (5 versus 4 in our example). In particular, the frequency of any non-pivot sequence can be equal, lower, or higher in $\mathscr{D}$ than in $\mathscr{P}_B$ without affecting correctness. The above discussion leads to the following lemma:

Lemma 5.2. *If $\mathscr{D}$ and $\mathscr{P}_w$ are w-equivalent w.r.t. λ and γ, then $f_\gamma(S, \mathscr{D}) = f_\gamma(S, \mathscr{P}_w)$ for all S satisfying $p(S) = w$ and $2 \leq |S| \leq \lambda$.*

Our notion of w-equivalency is a generalization of the corresponding notion of MG-FSM; we refer to Miliaraki et al. (2013) for a more formal treatment and proof of correctness. The key difference in LASH is the correct treatment of hierarchies.

5.3.2 w-Generalization

From the discussion above, we conclude that we can rewrite each input sequence T into any w-equivalent sequence $T' = \mathscr{P}_w(T)$ in Line 3 of Algorithm 5.3. Our main goal is to make T' as small as possible; this decreases both communication cost, computational cost, and (as will become evident later) skew.

Fix some pivot w and let $T = t_1 t_2 \dots t_l$. The first and perhaps most important of our rewrites is called w-generalization, which tries to rewrite T such that only "relevant" items remain. Recall from Section 3.2.3 that an item is *w-relevant* if $w' \leq w$; otherwise it is w-irrelevant. Similarly, index i is w-relevant if and only if t_i is w-relevant. For example, in sequence $T_2 = ab_3ccb_2$ only index 1 is B-relevant.

The key insight of w-generalization is that any generalized subsequence of T that contains an irrelevant item cannot be a pivot sequence (since the pivot is smaller than any irrelevant item by definition). Ideally, we would like to simply drop all irrelevant items from T; unfortunately, such an approach may lead to incorrect results since (1) if we drop irrelevant items, we cannot guarantee that the gap constraint remains satisfied and (2) generalizations of irrelevant items may be relevant and thus be part of a pivot sequence. To illustrate the violation of the gap constraint, suppose that we dropped cc from $T_2 = ab_3ccb_2$ to obtain $T_2' = ab_3b_2$. Then $BB \in G_{B,2}(T_2')$ but $BB \notin G_{B,2}(T_2')$ for $\gamma = 1$. To illustrate the second point, suppose that we drop all irrelevant items from T_2 to obtain a. We then miss pivot sequence $aB \sqsubseteq_1 T_2$ since $aB \not\sqsubseteq_1 a$.

Instead of dropping irrelevant items, w-generalization replaces irrelevant items by a carefully chosen set of items in the partition $\mathscr{P}_w$ to ensure correctness (MG-FSM would have replaced these items by blanks; see Section 3.2.3). There are two cases: (1) If index i is irrelevant and item t_i does not have an ancestor $w' < w$, we replace t_i by the special blank symbol $\_$, where $w < \_$ for all $w \in \Sigma$. The blank symbol acts as a placeholder and is needed to handle gap constraints. (2) Index i is irrelevant but has an ancestor that is smaller than the pivot. Let w' be the largest such ancestor.

We then replace t_i by w'. This step is similar to the generalization performed by the semi-naïve algorithm. It is more effective, however, since it generalizes all items that are less frequent than the pivot, whereas the semi-naïve algorithm only generalizes infrequent items before applying the naïve algorithm. Continuing our example with $T_2 = ab_3ccb_2$ with pivot B, indexes 3 and 4 are irrelevant and replaced by blanks (since c does not have an ancestor that is more frequent than B), whereas indexes 2 and 5 are irrelevant and replaced by B (the largest sufficiently frequent ancestor of both b_3 and b_2). We thus obtain $T_2' = aB␣B$.

At first glance, it seems as if w-generalization does not help: T_2 and T_2' have exactly the same length. However, we argue that the use of T_2' leads to substantially lower cost. First, we can represent blanks more compactly than irrelevant items; e.g., by using run-length encoding ($T_2' = aB␣^2B$) and/or variable-length encoding (few bits for blanks). Second, for similar reasons, we can represent smaller, generalized items more compactly than large items. Third, w-generalization enables the use of other effective rewrite techniques; see Section 5.3.3. Finally, w-generalization (as well as some of the other rewrites) makes sequences more uniform. If two sequences agree on their w-generalization, they can be "aggregated"; see the discussion in Section 5.3.4.

The correctness of w-generalization is captured in the following lemma.

Lemma 5.3. *Let $T = t_1t_2 \cdots t_n$ and denote by $T' = t_1't_2' \cdots t_n'$ the w-generalization of T. Then T and T' are w-equivalent.*

Proof. We have to show that $G_{w,\lambda}(T) = G_{w,\lambda}(T')$. Let $S = s_1 \ldots s_k \in G_{w,\lambda}(T)$. By definition, sequence S is a generalized subsequence of T, $p(S) = w$, and $s_j \leq w$ for $1 \leq j \leq k$. Thus there exists a set of indexes $1 \leq i_1 < \ldots < i_k \leq n$ such that $t_{i_j} \Rightarrow^* s_j$ and $i_{j+1} - i_j - 1 \leq \gamma$. We claim that $t'_{i_j} \Rightarrow^* s_j$ so that $S \in G_{w,\lambda}(T')$. There are two cases. If $t_{i_j} \leq w$, w-generalization does not modify index t_j so that $t'_{i_j} = t_j \Rightarrow^* t_j$. Otherwise, if $t_{i_j} > w$, w-generalization replaces t_{i_j} by the largest ancestor t'_{i_j} that is smaller then the pivot. Since $t_{i_j} \Rightarrow^* s_j$ and $s_j \leq w$, we conclude that $t_{i_j} \Rightarrow^* t'_{i_j} \Rightarrow^* s_j$ holds as well. Putting everything together, we obtain $G_{w,\lambda}(T') \subseteq G_{w,\lambda}(T')$.

It remains to show that $G_{w,\lambda}(T') \supseteq G_{w,\lambda}(T)$. This can be shown using the property that whenever $S \in G_{w,\lambda}(T')$, then $␣ \notin S$. The proof is similar to the one above and omitted here. □

5.3.3 Other Rewrites

LASH performs a number of additional rewrites, all of which aim to reduce the length of the sequence. In contrast to w-generalization, these rewrites closely resemble the rewrites of MG-FSM; we summarize them here and point out minor differences.

The first rewrite removes items that are unreachable in that they are "far away" from a *pivot index*. Let $T = t_1t_2 \ldots t_l$. In what follows, we assume that T has already

been w-generalized; then t_i is a pivot index if and only if $t_i = w$. In our implementation, w-generalization and unreachability reduction are performed jointly and are thus slightly more complex. Consider for example the sequence $T = ab_1acd_1ad_2cfb_2c$, pivot D, the hierarchy of Figure 7.1b, and the item order of Figure 5.2. We obtain $T' = ab_1acDaDc␣Bc$ by D-generalization; thus indexes 5 and 7 are pivot indexes. We then compute the left and right distances to a pivot, as well as the minimum distance. The left/right distance of an index is the size of the minimum set of increasing/decreasing indexes from a pivot index to the target index; only indexes that do not correspond to a blank as well as the target index are allowed and subsequent indexes must satisfy the gap constraint (at most γ items in between). For $\gamma = 1$, we obtain:

i	1	2	3	4	**5**	6	**7**	8	9	10	11
t'_i	a	b_1	a	c	D	a	D	c	␣	B	c
left	-	-	-	-	1	2	1	2	2	3	4
right	3	3	2	2	1	2	1	-	-	-	-
minimum	3	3	2	2	1	2	1	2	2	3	4

Here "-" corresponds to infinite distance. The left pivot distance of index 11, for example, is determined by the index sequence 7, 8, 10, 11 (length 4); the sequence 7, 9, 11 is not allowed since index 9 corresponds to a blank. As argued in MG-FSM, indexes that have distance larger than λ are unreachable and the corresponding items can be removed safely. For $\lambda = 2$, we obtain the reduced sequence $acDaDc␣$; for $\lambda = 3$, we obtain $ab_1acDaDc␣B$.

We also make use of a few other reductions of MG-FSM, which also apply to our generalized setting. First, we remove isolated pivot items, i.e., pivot items that do not have a non-blank item close by (within distance γ). We also remove leading and trailing blanks and replace any sequence of more than $\gamma + 1$ blanks by exactly $\gamma + 1$ blanks.

5.3.4 Putting Everything Together

We perform the above mentioned rewrites efficiently as follows. We first scan the sequence from right to left and, for each index, perform w-generalization and compute its left distance. We then scan the sequence from left to right, compute the right and pivot distance of each index, remove unreachable indexes, and remove blanks as described above. For a fixed hierarchy, the computational complexity for rewriting an input sequence of length l given a pivot is $O(l)$. Since an input sequence has at most δl pivot items, the overall computational complexity is $O(\delta l^2)$. Moreover, we output $O(\delta l)$ rewritten sequences of length at most l for all choices of γ and λ. Thus the communication complexity of LASH is polynomial, whereas the communication complexity of the naïve and semi-naïve approaches can be exponential

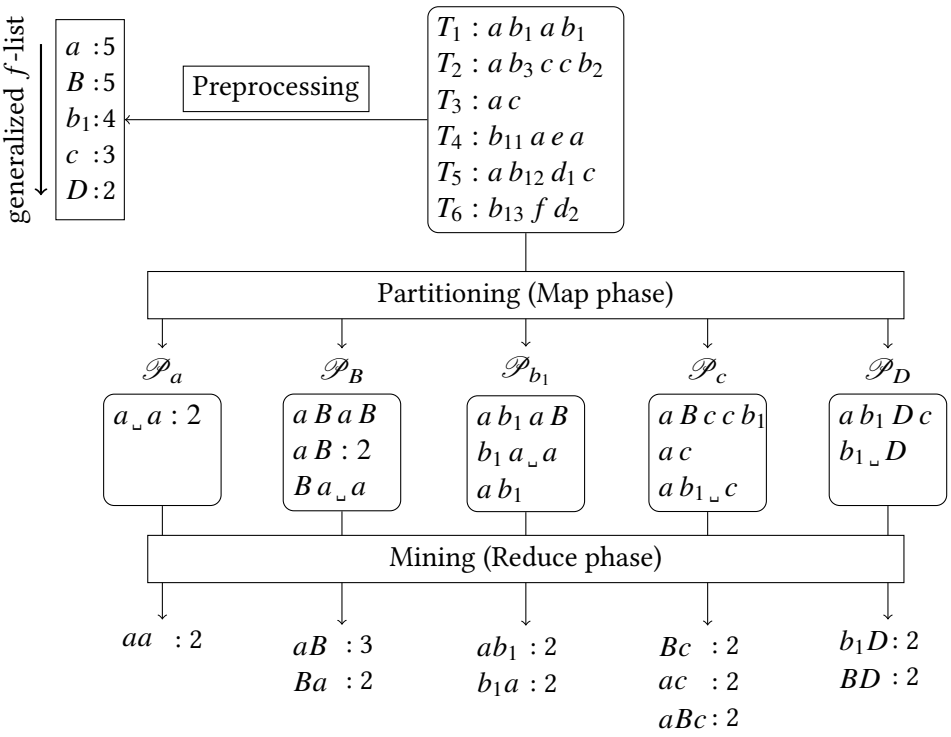


Figure 5.2: *Preprocessing, partitioning and mining phases of LASH for* $\sigma = 2$, $\gamma = 1$ *and* $\lambda = 3$.

($O((\delta + 1)^l)$). Moreover, our experiments suggest that LASH often performs much better than what could be expected from the above worst-case analysis.

The partitions generated by LASH for our example database are given in Figure 5.2. Recall the partition $\mathscr{P}_B$ from Equation (5.1). Using our rewrites, we obtain

$$\mathscr{P}_B = \{\, a\,B\,a\,B,\ a\,B,\ B\,a\,␣\,a,\ a\,B \,\}$$

which is significantly smaller. Observe that sequence aB occurs twice. We use combine functionality of Hadoop to aggregate such duplicated sequences. We also perform aggregation in the reduce function before starting the actual mining phase. Continuing the example, the final partition $\mathscr{P}_B$ is given by

$$\mathscr{P}_B = \{\, a\,B\,a\,B : 1,\ a\,B : 2,\ B\,a\,␣\,a : 1 \,\}\,.$$

Aggregation of duplicated sequences saves communication cost and reduces the computational cost of the GSM algorithm run in the mining phase.

5.4 Sequential GSM Algorithms

We now discuss methods to mine generalized sequences in each partition. In particular, we describe how sequential FSM algorithms of Section 3.3 can be adapted to handle hierarchies efficiently.

BFS with hierarchies

To adapt BFS approach to handle hierarchies, we first scan each sequence $T \in \mathscr{P}_w$ to create a posting list for each frequent length-2 generalized sequence. In particular, we add sequence T to the posting list of each element of $S \in G_2(T)$. Note that $G_2(T)$ consists of the 2-sequences that occur in S as well as all of their generalizations; this makes BFS approach hierarchy-aware. Consider for example input sequence $T = cab_1D$, the hierarchy of Figure 7.1b, and $\gamma = 1$. Then $G_2(T) = \{\, ca, cb_1, cB, ab_1, aB, aD, b_1D, BD \,\}$ so that we add T to 8 posting lists. This initial construction of the 2-sequence index is the only difference; i.e., we now proceed with level-wise approach unmodified. For example, when sequence ca and aD are frequent, we generate candidate sequence caD and obtain its frequency by intersecting the posting lists of ca and aD.

DFS with hierarchies

To adapt DFS to mine generalized sequences, we replace the projected database by the support set $\mathscr{D}_S$, which consists all input sequences in which S or a specialization of S occurs. When we right-expand S, the set of right items for transaction $T \in \mathscr{D}_s$ is given by $\Sigma_S^{\text{right}}(T) = \{\, w' \mid Sw' \sqsubseteq_\gamma T \,\}$, i.e., we look for occurrences of S or a *specialization* of S, and then consider the $\gamma + 1$ items to the right along with

their generalizations. For our example sequence $T = cab_1D$ with $\gamma = 1$, we have $\Sigma^{\text{right}}_{ca}(T) = \{ b_1, B, D \}$ and $\Sigma^{\text{right}}_{cB} = \{ D \}$. Like in DFS, right-expansion is performed by scanning $\mathscr{D}_S$ and computing the set $\Sigma^{\text{right}}_S = \bigcup_{T \in \mathscr{D}_S} \{\Sigma^{\text{right}}_S(T)\}$ of right items along with their frequencies. For each frequent item $w' \in \Sigma^{\text{right}}_S$, we output Sw' and recursively grow Sw'.

PSM with hierarchies

Adaption of PSM to mine generalized sequences is similar to that of the DFS approach described above. Recall from Section 3.3.2, that PSM starts with the pivot item w and recursively computes pivot sequences of form wS by making a series of right-expansions. Adaptation of right-expansions with hierarchies is similar to the one above (like the DFS) but we never right-expand with the pivot item w. After all the right-expansions are made, PSM then makes a series of left-expansions. To adapt left-expansions to hierarchies, we compute the set of items $\Sigma^{\text{left}}_S = \bigcup_{T \in \mathscr{D}_S} \{\Sigma^{\text{left}}_S(T)\}$, where $\Sigma^{\text{left}}_S(T) = \{ w' \mid w'S \sqsubseteq_\gamma T \}$, i.e., we compute the set of $\gamma + 1$ items to the left of pivot w along with their generalizations. For each left-expanded sequence, we compute a series of right-expansions and proceed as PSM.

5.5 Experiments

We now present results of our experimental study using two large real-world datasets in the contexts of generalized n-gram mining from textual data and customer behavior mining from product sequences. In particular, we compared LASH to the naïve and the semi-naïve algorithms, evaluated the efficiency of sequential GSM algorithms for mining each partition, and studied the scalability of LASH. We also studied the effect of different parameters—i.e., support (σ), gap (γ) and length (λ)—and how different types of hierarchies affect the performance of LASH.

We found that LASH outperformed the naïve and semi-naïve algorithms by multiple orders of magnitude. For mining partitions locally, the PSM algorithm was more efficient and faster than the BFS and DFS algorithms. Our scalability experiments suggest that LASH scales linearly as we add more compute nodes and/or increase input dataset size.

5.5.1 Setup

Implementation and cluster

We implemented LASH, the semi-naïve and naïve methods in Java (JDK 1.7). We represent items by assigning integers item ids according to the order $<$ obtained from the generalized f-list. Thus, highly frequent items are assigned smaller integer ids. We represent sequences as arrays of item ids and compress the data transmitted between the map and reduce phase using variable-length integer encod-

Dataset	Sequences	Avg length	Max length	Total items	Unique items
NYT	49,605,960	21.1	15,199	1,047,419,137	2,763,301
AMZN	6,643,666	4.5	25,630	29,667,966	2,374,096

Table 5.1: *Dataset characteristics*

Dataset	Hierarchy	Total items	Leaf items	Root items	Intermediate items	Levels	Avg.fan-out	Max.fan-out
NYT	L	2,910,327	407,806	2,502,521	0	2	2.7	36
	P	2,617,581	2,617,559	22	0	2	124,645.6	1,828,130
	LP	2,910,347	2,763,300	22	147,025	3	19.8	1,822,454
	CLP	2,970,092	2,763,300	22	206,770	4	14.4	1,822,454
AMZN	h2	2,374,147	2,371,524	2,623	0	2	48,398.4	904,162
	h3	2,374,509	2,371,536	2,630	343	3	6,050.7	332,723
	h4	2,376,539	2,371,670	2,633	2,236	4	1,038.9	332,723
	h8	2,387,422	2,373,158	2,634	11,630	8	204.2	332,723

Table 5.2: *Hierarchy characteristics*

ing. All experiments were run on a local Hadoop cluster consisting of eleven Dell PowerEdge R720 computers, each with 64GB of main memory, eight 2TB SAS 7200 RPM hard disks and two Intel Xeon E5-2640 6-core CPUs. Debian Linux (kernel version 3.2.48.1.amd64-smp) was used as an operating system. The machines in the cluster are connected via 10 GBit Ethernet. We use the Cloudera cdh3u6 distribution of Hadoop 0.20.2 running on Oracle Java 1.7.0_25. One machine acted as a Hadoop master node; the other ten machines acted as worker nodes. The maximum number of concurrent map or reduce tasks was set to 8 per worker node. All tasks are launched with 4 GB heap space.

Datasets

Statistics of the datasets and hierarchies used in our experiments are summarized in Table 5.1 and Table 5.2 respectively. We used two real-world datasets: The New York Times corpus (NYT) for mining generalized *n*-grams and Web data: Amazon reviews (AMZN) for mining generalized product sequences.

The NYT dataset consists of roughly 50M sentences from 1.8 million articles published during 1987 and 2007. We treat each sentence as an input sequence with each word (token) as an item. We generated a syntactic hierarchy by annotating the each word with its lemma and part-of-speech tag using the Stanford CoreNLP parser

and also annotated each word with its lowercase form (if different that its surface form). In our syntactic hierarchy, a word appearing in a sentence can generalize to its lowercase form, which generalizes to its lemma, which in turn generalizes to its part-of-speech tag. For example, the word "Changing" ⇒ "changing" ⇒ "change" ⇒ "VERB". We generated four variants of this hierarchy: NYT-L (word ⇒ lemma), NYT-P (word ⇒ pos), NYT-LP (word ⇒ lemma ⇒ pos) and NYT-CLP (word ⇒ case ⇒ lemma ⇒ pos). Note that the surface form of many words appearing the input sequences is same as their lowercase or lemma; this naturally creates a hierarchy in which items appearing in the input sequences come from different levels.

The AMZN dataset consists over 35 million reviews from over 6 million users spanning from 1995 to 2013. To generate product sequences, we identified a user sessions by grouping the reviews by user and sorting each so-obtained sequence by timestamp. We used the Amazon product hierarchy, in which, for example, the book "For Whom the Bell Tolls" ⇒ "Classics" ⇒ "Literature & Fiction" ⇒ "Books". We also considered different hierarchy types of varying depths (2–8) by varying the number of intermediate categories a product is assigned to.

Measures

In the following experiments, we report the performance measure as total time elapsed between launching a task and receiving the final result. We break down this time into time taken by the map phase, shuffle phase and the reduce phase. Since these phases overlap in a MapReduce job, we report the time elapsed until finishing of each phase. We also report the bytes transferred as the total data transferred between map and reduce task as obtained from Hadoop's `MAP_OUTPUT_BYTES` counter. All measurements reported are based on average of three independent runs and were performed with exclusive access to the machines.

5.5.2 Results

A. Overall Runtime

We initially evaluated the performance of LASH generalized n-gram mining (i.e., $\gamma = 0$) and compared it with the naïve and semi-naïve methods (discussed in Section 5.2) using the NYT-P dataset having two levels of hierarchy. For this dataset, we mined generalized n-grams with three different parameter settings of increasing difficulty w.r.t. to output size. The results are plotted using a log-scale in Figure 5.3a. With $\sigma = 1000, \lambda = 3$ and $\sigma = 100, \lambda = 3$, LASH obtained a speedup of around 10×. Further, LASH achieves a speed up of more than 50× for the setting with $\sigma = 100$, $\lambda = 5$.

For the entire NYT-CLP dataset having four levels of hierarchy, the naïve and semi-naïve algorithms were unable to handle the combinatorial blowup of the search space and were aborted after 12 hours. On the other hand, LASH required a little

over 600 seconds only. Also, as shown in Figure 5.3b, the total bytes transferred between the map and reduce phase is significantly less for LASH.

B. Local Mining

In our next set of experiments, we studied the efficiency of the sequential GSM algorithms of Sec. 5.4. We used the NYT dataset and performed runs with different settings of increasing difficulty w.r.t. output size. The results are shown in Figure 5.4a using log-scale. Since our choice of sequential mining approaches only affect the reduce phase, we report the mining time as time elapsed between the end of last reduce task and end of first sort task.

Compared to BFS, with the LP hierarchy (three levels) and the parameters $\sigma = 1000$, $\lambda = 5$, PSM was 9× faster. As we decreased the value of σ to 100, PSM was 15× faster and up to 22× faster with the full CLP hierarchy. For the setting CLP ($\sigma = 100$, $\lambda = 7$), BFS reported insufficient memory and terminated. On comparison to DFS, PSM was 2.5× to 3.5× faster for these settings. The efficiency of PSM stems from an optimized search space exploration of pivot sequences w.r.t. the partitions being mined. Since PSM uses a customized depth-first search, we also compared the number of candidate sequences generated per output sequence by DFS and PSM. As observed from Figure 5.4b, PSM explores a much smaller fraction of the search space.

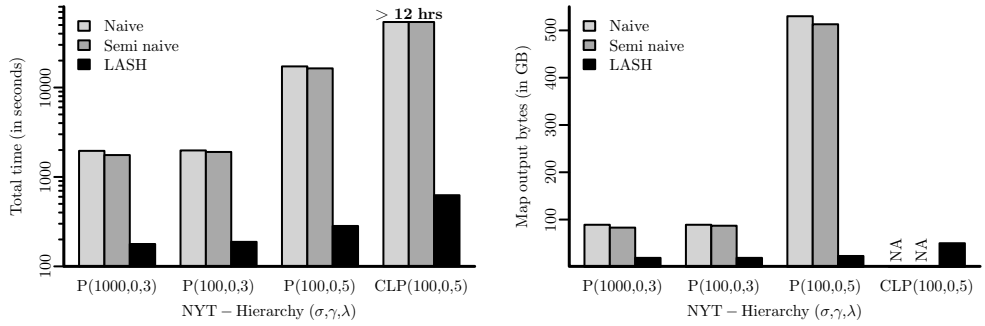
We also studied PSM's performance with indexing right-expansions (PSM+Index). We observed a trade-off between the construction cost and the benefit of indexing. The runtimes improved by 100s with increase in the values of λ and levels of hierarchy. We also observed that in all the cases, our indexing significantly pruned a lot of search space up to 2× (see Figure 5.4b).

C. Effect of Parameters

In this group of experiments, we studied how the performance is affected by different parameters σ, γ and λ. We used the AMZN-h8 dataset with full 8 levels of the hierarchy and fixed the parameters to $\sigma = 100$, $\gamma = 1$ and $\lambda = 5$.

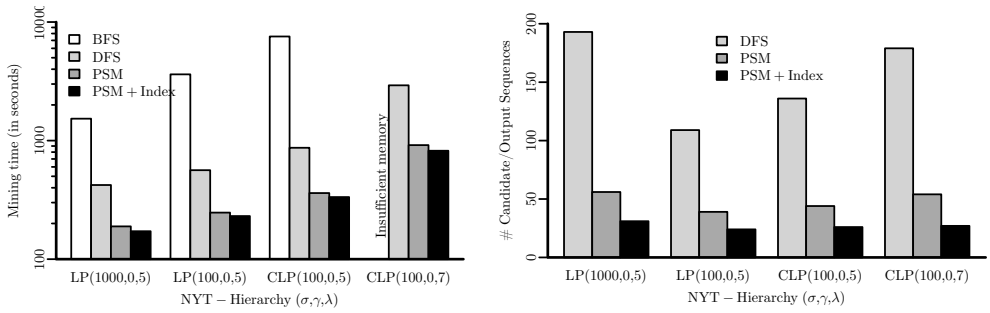
We first studied how the minimum support σ affects the performance by varying its value from 10 to 10,000. The results are shown in Figure 5.5a. The time taken by the map phase which consist of rewriting input sequences for each partition decreases as we increase the support. Recall that, our rewrites are independent of σ (see discussion in Section 5.3.4); however, σ has an indirect effect. At higher supports, fewer items from the lower levels of the hierarchy are frequent so that the effective depth of the hierarchy is reduced. Since our rewrites depends on this depth, the time per rewrite decreases as the support threshold is increased. The reduce time decreases as well since mining becomes cheaper at higher supports.

Second, we varied the value of maximum gap γ from 0 to 3. As we can see in Figure 5.5b, the impact on map times was not significant as the cost of rewriting is



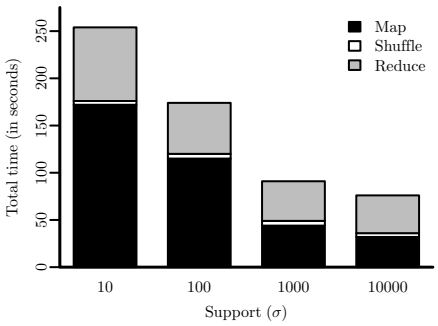
(a) Total time **(b)** Total bytes

Figure 5.3: *Performance of distributed algorithms.*

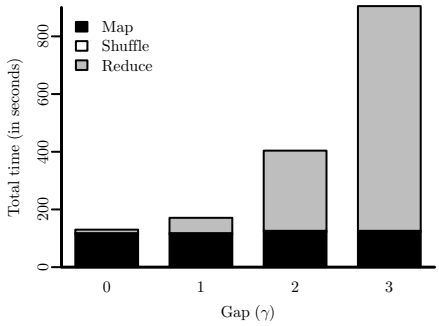


(a) Mining time **(b)** Search space

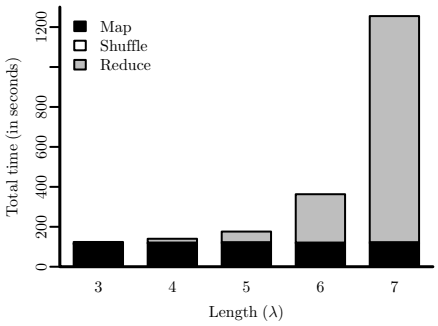
Figure 5.4: *Performance of sequential algorithms.*



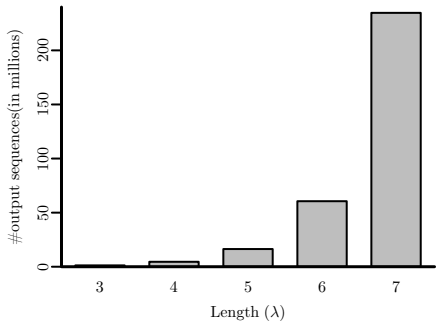
(a) AMZN-h8 ($\gamma = 1$, $\lambda = 5$)



(b) AMZN-h8 ($\sigma = 100$, $\lambda = 5$)

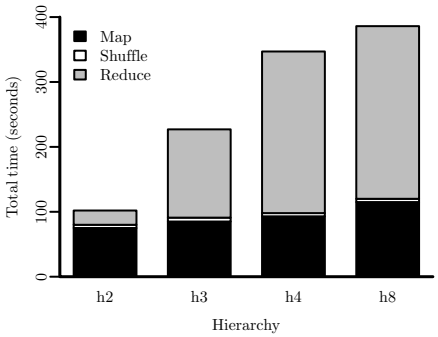


(c) AMZN-h8 ($\sigma = 100$, $\gamma = 1$)

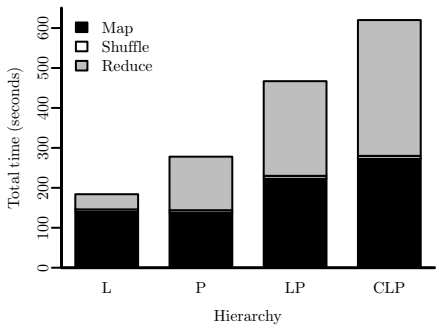


(d) AMZN-h8 ($\sigma = 100$, $\gamma = 1$)

Figure 5.5: *Effect of different parameters*



(a) AMZN ($\sigma = 100$, $\gamma = 2$, $\lambda = 5$)



(b) NYT ($\sigma = 100$, $\lambda = 5$)

Figure 5.6: *Effect of different hierarchies.*

largely independent of γ. However, it had a significant impact on the reduce times as as the search space during mining significantly increases with γ.

Lastly, we evaluated how maximum length λ effects the performance of LASH by varying its value from 3 to 7. The results are shown in Figure 5.5c. We observed that λ had very little impact on the map time. The reduce time increases significantly as we increase λ since mining becomes more expensive. In fact, the output size increases as we increase λ. Figure 5.5d shows that output size and reduce times are correlated.

D. Effect of Hierarchies

In this group of experiments, we studied how different types of hierarchies affect the performance of LASH. We used the AMZN and NYT datasets.

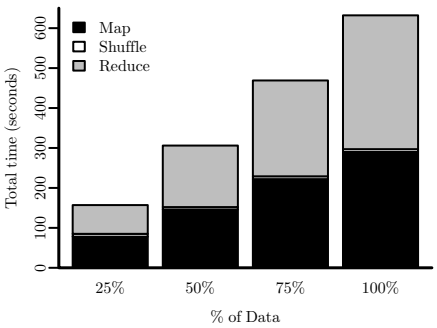
For the AMZN dataset (Figure 5.6a), we fixed the parameters $\sigma = 100$, $\gamma = 2$, $\lambda = 5$ and varied the hierarchy levels from 2 to 8. The map times slightly increases with an increase in levels, even though the support is fixed. This is because rewriting each sequence depends on the hierarchy depth. The reduce times increase significantly since an increase in hierarchy levels in turn increases the number of intermediate items (see Table 5.2). This makes mining more expensive: a partition needs to be created and mined for each intermediate item and the mining time of a partition also depends on the depth of the hierarchy. The effect in reduce times is less pronounced when the using the full hierarchy (8 levels) compared to 4 levels because most products in the Amazon product hierarchy have no more than 4 parent categories.

For the NYT dataset (Figure 5.6b), we set $\sigma = 100$, $\lambda = 5$, and considered four variants of the syntactic hierarchy (see Section 5.5.1). NYT-L and NYT-P both have two levels but show a significant difference in reduce times. This is because the NYT-L hierarchy has many roots with low fan-out, whereas the NYT-P hierarchy has few roots with high fan-out. Mining the latter hierarchy is more expensive, partly due to the high frequency of the root items, partly due to larger output size. We also observed that adding more levels to the hierarchy (NYT-LP and NYT-CLP) significantly increases both the map and the reduce times.

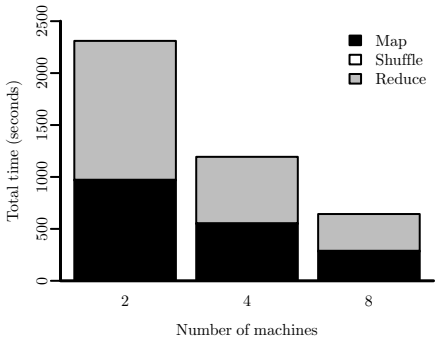
E. Scalability

In our final group of experiments, we studied the scalability of LASH as we add more compute nodes and/or increase the input data size. We used the NYT dataset with full CLP hierarchy and set the parameters $\sigma = 100$ and $\lambda = 5$.

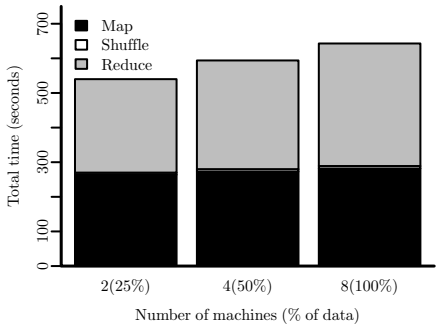
We first investigated the performance of LASH as we vary the input data size. To this end, from the NYT dataset, we extracted datasets that contain a random 25%-, 50%- and 75%-sample of the input sequences. The results are shown in Figure 5.7a. We observed that LASH is robust in handling increasing amounts of data with both map and reduce times increasing linearly as we add more data.



(a) NYT-CLP (σ = 100, λ = 5)



(b) NYT-CLP (σ = 100, λ = 5)



(c) NYT-CLP (σ = 100, λ = 5)

Figure 5.7: *Scalability results*

We evaluated strong scalability by running LASH on a fixed dataset (100% NYT-CLP) and varying the amount of parallel work by using 2, 4 and 8 compute nodes. Figure 5.7b shows that LASH exhibits good linear scalability with both map and reduce times decreasing equally as we increase the number of compute nodes.

We also evaluated weak scalability for LASH, in which we increase the input data size as we add more compute nodes. In particular, we simultaneously increased the size of the input data (25%, 50% and 100% of NYT-CLP) and number of compute nodes (2, 4 and 8). As observed from Figure 5.7c, LASH exhibits good weak scalability. Note that the total time ideally remains constant as we double both computational resources and input dataset. In practice, however, the number of output sequences increases by a factor of more than 2 when doubling the input data. We thus observe a slight increase in the runtimes. In this particular case, the number of output sequences increased from 43M (25% of input) to 99M (50% of input) to 220M (100% of input), which is a factor of 2.2×.

F. Output Statistics

We computed a number of statistics of the set of generalized subsequences that we mined from our datasets; the results are shown in Table 5.3.

First, we computed the percentage of *non-trivial* output sequences to judge whether generalized sequence mining is beneficial. We say that an output sequence is *trivial*, if it can be generated from the output of a standard sequence miner (which ignores hierarchies) by generalizing items. For example, non-trivial sequences on the NYT-CLP dataset ($\sigma = 100$) include: "NOUN lives in NOUN", "NOUN works at NOUN" and "the ADJ Book"; no specializations of these patterns were frequent in the input data. For the NYT and AMZN datasets, we observed that more than 70% and 95%, resp., of the sequences were non-trivial.

Recall the discussion at the end of Section 5.1, in which we argue that GSM may produce "redundant" (but nevertheless potentially useful) sequences. To see how many redundant sequence are mined, we computed the number of closed and maximal subsequences. In the context of GSM, a frequent sequence S is *maximal* if every supersequence $S' \sqsupseteq_0 S$ is infrequent, and closed if every supersequence has a different frequency. In Table 5.3, we observe that adding more levels to the hierarchy or lowering the support increases the fraction of redundant patterns, but that nevertheless a large number of patterns is non-redundant.

Dataset		Non-trivial (%)	Closed (%)	Maximal (%)
	Hierarchy			
NYT ($\sigma = 100, \lambda = 5$)	P	75.47	89.08	31.92
	LP	73.47	50.38	10.11
	CLP	70.26	35.42	6.06
	Min sup (σ)			
AMZN-h8 ($\gamma = 1, \lambda = 5$)	10,000	100	100	21.56
	1,000	99.78	85.79	14.50
	100	97.38	64.86	10.06

Table 5.3: *Output Statistics*

5.6 Related Work

Srikant and Agrawal (1996) proposed the use of extended sequences to incorporate hierarchies into the mining process. In this approach, each item in a sequence is replaced by an itemset containing the item and all its ancestors. As mentioned before, generalized sequence mining using extended sequences is inefficient as it increases the blows up sequence database by a factor of roughly the depth of the hierarchy, which makes repeated database scans by GSP much more expensive. Hierarchies have also been explored in context of multi-dimensional sequential pattern mining. To this end, Plantevit et al. (2006, 2010) proposed the HYPE algorithm and the M^3SP algorithm as its successor. In their approach, they prune hierarchies by only considering maf-sequences, which are pairs of items (each belonging to a dimension) that are maximal (i.e., each specialization is infrequent). Subsequently, they use SPADE to generate frequent sequences. A known limitation of their approach is that they do not mine all frequent sequences. Chen and Huang (2008) sketched the idea of fuzzy multi-level sequential patterns. They consider hierarchies in which an item can have more than one parent with different degrees of confidence and use a GSP-like approach to mine such patterns. Huang (2009) later presented a divide-and-conquer strategy based on the pattern-growth approach to mine such fuzzy multi-level patterns. Both approaches encode hierarchy information in each item, which is reminiscent of extended sequences and are outperformed by GSP [Huang (2009)].

5.7 Summary

We proposed LASH, an algorithm for mining frequent sequences in presence of hierarchies. To the best of our knowledge, LASH is the first distributed, scalable algorithm for mining such generalized sequences. LASH uses a novel, hierarchy-aware form of item-based partitioning and optimized partition construction tech-

niques that are specifically designed to handle hierarchies. Our experimental study indicates that LASH is efficient, scales to large real-world datasets, and is multiple orders of magnitude faster than existing baseline methods.

Part II

Non-traditional Subsequence Constraints: Expressibility

Chapter 6

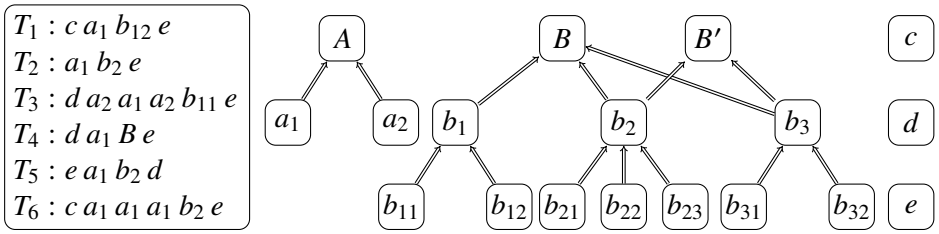
Expressing Subsequence Constraints

In this part of the thesis, we turn our attention to a general purpose framework for frequent sequence mining with subsequence constraints. We show that many subsequence constraints—including and beyond those discussed earlier—can be unified in a single framework. A unified treatment allows researchers to study subsequence constraints in general instead of focusing on certain combinations individually. It also helps to improve usability of pattern mining systems because it avoids the need to develop customized mining algorithms for the particular subsequence constraint of interest that arise in applications (e.g., verbal phrases between entities in information extraction applications or product sequences of a certain type in customer behavior mining applications).

Our focus in this chapter[a] lies on modeling and expressing subsequence constraints in a suitable way. We introduce *subsequence predicates* to model subsequence constraints in a general way, and propose a *pattern expression language* to concisely express subsequence predicates. We subsequently suggest a computational model based on finite state transducers, and describe the formal semantics of our language.

In Section 6.1, we formally define subsequence predicates. In Section 6.2, we propose our pattern expression language, which is based on regular expressions, but support capture groups and item hierarchies. Capture groups are the key ingredient for expressing most prior subsequence constraints in a unified way; see Table 6.2 on page 90 for examples. Direct support for item hierarchies allows us both to express subsequence constraints concisely and to mine generalized subsequences. Some example pattern expressions for expressing subsequence constraints in information

[a] The material in this chapter is based on Beedkar and Gemulla (2016).



(a) Example database **(b)** Example hierarchy

Figure 6.1: *A sequence database and its vocabulary*

extraction, natural language processing, and customer behavior mining applications are given in Tables 6.3 and 6.4 on page 90. In Section 6.3 we propose finite state transducers (FST) as a computational model for our pattern expressions. We show that FSTs are sufficiently powerful to express many subsequence constraints and provide translation rules for our pattern expressions to FSTs. Finally, in Section 6.4, we propose an *advanced pattern expression language*, which include features useful for expressing highly-customized subsequence constraints that may arise in applications and for mining generalized sequences in a more controlled fashion.

6.1 Subsequence Predicates

A subsequence constraint describes which subsequences of a given input sequence should be considered for frequent sequence mining. Our goal is to provide a general purpose framework to express subsequence constraints, including and beyond previously proposed constraints. Consider the following (admittedly contrived) subsequence constraint as an example.

Example 6.1. *Consider the example database $\mathscr{D}_{ex}$ shown in Figure 6.1. Suppose that we are interested in mining sequences of B's and/or descendants of A's. We restrict attention to sequences that occur consecutively in input sequences starting with c or d and ending with e. We also allow to generalize occurrences of descendants of A and B. Then $a_1 B \sqsubseteq T_1$ and $AB \sqsubseteq T_1$ satisfies this subsequence constraint, whereas $a_1 b_{12} \sqsubseteq T_1$, $a_1 b_1 \sqsubseteq T_1$, $a_1 B \sqsubseteq T_2$ and $AB \sqsubseteq T_2$ do not.*

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

We propose subsequence predicates as a general, natural model for subsequence constraints. A *subsequence predicate* P is a predicate on pairs (S, T), where $T \in \Sigma^+$ is any input sequence and $S \sqsubseteq T$ is a subsequence. Subsequence $S \sqsubseteq T$ satisfies the constraint when $P(S, T)$ holds. Note that P is not a predicate on (only) subsequence

S; it also involves input sequence T. We denote by $G_P(T) = \{ S \sqsubseteq T \mid P(S,T) \}$ the set of *P-subsequences* in T. For each $S \in G_P(T)$, we say that S is *P-generated* by T. For example, let P_{ex} be the subsequence predicate that expresses subsequence constraint of Example 6.1, then $G_{P_{ex}}(T_1) = \{ a_1B, AB \}$ and $G_{P_{ex}}(T_2) = \emptyset$.

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

FSM and subsequence predicates

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

$$\mathrm{Sup}_P(S, \mathscr{D}) = \{ T \in \mathscr{D} \mid S \in G_P(T) \}\,. \tag{6.1}$$

The *P-frequency* of S in $\mathscr{D}$ is given by $f_P(S, \mathscr{D}) = |\mathrm{Sup}_P(S, \mathscr{D})|$. In our example database, we have $\mathrm{Sup}_{P_{ex}}(Aa_1AB, \mathscr{D}_{ex}) = \{ T_3, T_6 \}$ and thus $f_{P_{ex}}(Aa_1AB, \mathscr{D}_{ex}) = 2$. Given a *support threshold* $\sigma > 0$, we say that a sequence S is *P-frequent* if $f_P(S, \mathscr{D}) \geq \sigma$. We are interested in mining all P-frequent sequences; these sequences capture common, relevant patterns in the data.

> 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 frequencies.

For example, reconsider the example sequence database and its vocabulary of Figure 6.1 and Example 6.1. Let P_{ex} be the subsequence predicate that expresses the subsequence constraint of Example 6.1. The set of all P_{ex}-frequent sequences for $\sigma = 2$ in is given by

$$\{ AAAB : 2, AB : 2, Aa_1AB : 2, a_1B : 2 \},$$

where we also give P-frequencies.

The above definitions are generalizations of the notions of frequency and support used in traditional frequent sequence mining. 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. Nevertheless, our mining algorithms, which we will describe in the next chapter, make use of suitable adapted notions of antimonotonicity for subsequence predicates (Lemma 7.1; page 110) and pattern expressions (Lemma 7.2; page 112).

6.2 Pattern Expression Language

In this section, we propose a pattern expression language for expressing subsequence predicates. Our language is based on regular expressions and designed to be simple and intuitive. Although there are subsequence predicates that cannot be expressed, our language is sufficiently expressive for many practical purposes: it covers and goes beyond all notions of subsequence constraints discussed earlier.

6.2.1 Pattern Expressions

Our language consists of the following set of *pattern expressions*, defined inductively:

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

Pattern expressions are based on regular expressions, but additionally include capture groups (in parentheses), hierarchies (by omitting $_=$), and generalizations (using $^\uparrow$ and $_=^\uparrow$). We make use of the usual precedence of rules for regular expressions to suppress square brackets (but not parentheses); operators that appear earlier in the above definition have higher precedence. We refer to expressions of form (1) or (2) as *item expressions*. We write $G_E(T)$ to refer to the set of subsequences "generated" by expression E on input T (see Section 6.3 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 make use of parentheses to distinguish these two cases; the semantics is similar to the use of capture groups in regular expressions [PCRE]. Given an expression E, only subexpressions that are enclosed in or contain a capture group will contribute to output; all other subexpressions serve to describe context information. For example, the pattern expression

$$E_{ex} = [c|d]([A^\uparrow \mid B_=^\uparrow]^+)e. \tag{6.2}$$

describes precisely the subsequence constraint of Example 6.1. The captured subexpression $[A^\uparrow \mid B_=^\uparrow]^+$ specifies the output, i.e., consecutive sequences of A's or its descendants and B's, and the uncaptured subexpressions $[c|d]$ and e specifies the context, i.e., the input sequences in which consecutive sequences of A's or its descendants and B's occur should start with a c or d and end with e.

Item expressions

Item expressions are the elementary form of pattern expressions and apply to one input item. If the item expression "matches" the input item, it can "produce" an output item; Table 6.1 provides an overview of basic item expressions. 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 7.1b, 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 $w' \in \text{desc}(w)$ (which includes w) and, when captured, produces the item that has been matched. For example, we have $G_{(A)}(A) = \{ A \}$, $G_{(A)}(a_1) = \{ a_1 \}$, and $G_{(A)}(b_1) = \emptyset$. Our language also provides *wild card* symbol "." to match any item; again, the matched item is produced when the wild card is captured. For example, $G_{(.)}(A) = \{ A \}$, $G_{(.)}(a_1) = \{ a_1 \}$ and, $G_{(.)}(b_1) = \{ b_1 \}$.

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 $w' \in \text{desc}(w)$—as expression w does—, and it produces either the matched input item or any of its ancestors that is also a descendant of w. For example, $G_{(B^{\uparrow})}(b_{12}) = \{ b_{12}, b_1, B \}$ and $G_{(b_1^{\uparrow})}(b_{12}) = \{ b_{12}, b_1 \}$. We also allow the use of a wild card with generalization operator: expression ".$^{\uparrow}$" matches any item and produces each of its generalizations. For example, $G_{(.^{\uparrow})}(b_1) = \{ b_1, B \}$. Our final item expression is used to enforce a generalization: $w_{=}^{\uparrow}$ matches any descendant of w and produces w, independently of which descendant has been matched. For example $G_{(B_{=}^{\uparrow})}(b_{12}) = \{ B \}$.

Composite expressions.

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

6.2.2 Examples

As mentioned earlier, our pattern expressions allow us to express many existing subsequence constraints in a unified way. We show in Table 6.2 how most prior subsequence constraints can be expressed using our pattern expressions. Note that the use of capture groups enables many of these pattern expressions.

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

Table 6.1: *Translation rules for basic item expressions (where $w, w', w'' \in \Sigma$)*

Pattern expressions can additionally express many customized subsequence constraints that arise in applications and cannot be handled by existing FSM frameworks. For example, expressions N_1–N_{11} shown in Table 6.3 express subsequence constraints useful for information extraction (IE) and natural language processing applications (NLP). These constraints were inspired from the work of Del Corro and Gemulla (2013); Del Corro et al. (2015); Fader et al. (2011); Lin et al. (2012); Manning and Schütze (1999); Nakashole et al. (2012); Trummer et al. (2015). Here we considered an item hierarchy in which words generalize to their lemmas, which in turn generalize to their part of speech tags and named entities generalize to their types (PERSON, ORGANIZATION, LOCATION, MISC) and then to ENTITY. For example, "lives"⇒"live"⇒"VERB" and "Barack Obama"⇒PERSON⇒ENTITY.

In Table 6.4, expressions A_1–A_4 express subsequence constraints use for market-basket analysis or for customer behavior mining applications. These expressions apply to product sequences obtained from Web data: Amazon reviews. Here we considered the Amazon product hierarchy in which for example, "Cannon5D"⇒"Digital Camera"⇒"Camera&Photo"⇒"Electronics".

6.3 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 [Mohri (1997)]. FSTs are similar to finite state automaton but additionally label transitions with output strings. Conceptually, an FST reads an input string and translates it to an output string in a nondeterministic fashion. We will use FSTs to specify subsequence predicate $P(S, T)$: the predicate holds if the FST can output subsequence S when reading input T.

6.3.1 Finite state transducers.

More formally, we consider a restricted form of FSTs defined as follows. An FST $\mathcal{A}$ is a 5-tuple $(Q, q_S, Q_F, \Sigma, \Delta)$, where:

- Q is a set of states,
- $q_S \in Q$ is the initial state,
- $Q_F \subseteq Q$ is the set of final states,
- Σ is an input and output alphabet, and
- $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Sigma \cup \{\epsilon\}) \times Q$ is a transition relation. For every transition $(q_{from}, in, out, q_{to}) \in \Delta$, we require that $out \in \text{anc}(in) \cup \{\epsilon\}$ and that whenever $in = \epsilon$ then $out = \epsilon$.

Our notion of FSTs differs from traditional FSTs in that we use a common input and output alphabet and in that we restrict output labels. The latter restriction ensures that our FSTs output generalized subsequences of their input (see Lemma 6.1 below).

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

Table 6.2: *Pattern expressions for traditional subsequence constraints.*

Pattern expression	Description
N_1: ENTITY (VERB$^+$ NOUN$^+$? PREP?) ENTITY	*Relational phrase between entities*
N_2: (ENTITY$^\uparrow$ VERB$^+$ NOUN$^+$? PREP? ENTITY$^\uparrow$)	*Typed relational phrases*
N_3: (ENTITY$^\uparrow$ be$^\uparrow_=$) DET? (ADV? ADJ? NOUN)	*Copular relation for an entity*
N_4: (.$^\uparrow$){3} NOUN	*Generalized 3-grams before a noun*
N_5: ([.$^\uparrow$ . .]|[. .$^\uparrow$.]|[. . .$^\uparrow$])	*Generalized 3-grams, where at most one item is generalized*
N_6: ([ADJ|NOUN] NOUN)	*Noun modified by adjective or noun*
N_7: (VERB PREP? NOUN$^+$)	*Verb, preposition and object*
N_8: (ADV? ADJ)	*Adjective with optional adverbial modifiers*
N_9: (NOUN PREP DET? NOUN)	*Noun phrases with preposition*
N_{10}: ENTITY (.*) ENTITY	*Phrases between entities*
N_{11}: (ENTITY) .* (ENTITY)	*Co-occurring entities*

Table 6.3: *Pattern expressions for subsequence constraints in information extraction and natural language processing applications.*

Pattern expression	Description
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*
A_2: (Book)[.{0,2}(Book)]{1,4}	*Sequences of books*
A_3: DigitalCamera[.{0,3}(.$^\uparrow$)]{1,4}	*Type of products bought after a digital camera*
A_4: (MInstr$^\uparrow$)[.{0,2}(MInstr$^\uparrow$)]{1,4}	*Generalized sequences of musical instruments*

Table 6.4: *Pattern expressions for subsequence constraints in customer behavior mining applications.*

FSTs can be viewed as a directed graph in which each state corresponds to a vertex and each transition $(q_{from}, in, out, q_{to}) \in \Delta$ to an edge from vertex q_{from} to vertex q_{to} with label *in*:*out*. We use shortcut notation $q_{from} \xrightarrow{in:out} q_{to}$ to denote transitions and refer to *in* as the *input label* and to *out* as the *output label*. We refer to transitions with input label ϵ (and thus output label ϵ) as *ϵ-transitions*. Figure 6.2 shows an example FST, where we mark the initial state $q_S = q_0$, final states $Q_F = \{ q_{11} \}$ with double circle, transitions with *in*:*out* labels and ϵ-transitions with ϵ.

Runs and outputs.

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

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

Example 6.2. *Consider the FST $\mathcal{A}_{F_{ex}}$ of Figure 6.2 and the example sequence database of Figure 6.1. $\mathcal{A}_{F_{ex}}$ has two accepting runs for sequence $T_1 = ca_1b_{12}e$, which are given by $p_1 = q_0 \xrightarrow{\epsilon} q_1 \xrightarrow{c:\epsilon} q_3 \xrightarrow{\epsilon} q_5 \xrightarrow{\epsilon} q_6 \xrightarrow{a_1:a_1} q_8 \xrightarrow{\epsilon} q_{10} \xrightarrow{\epsilon} q_5 \xrightarrow{\epsilon} q_7 \xrightarrow{b_{12}:B} q_9 \xrightarrow{\epsilon} q_{10} \xrightarrow{e:\epsilon} q_{11}$ with output $O(p_1) = a_1 B$, and $p_2 = q_0 \xrightarrow{\epsilon} q_1 \xrightarrow{c:\epsilon} q_3 \xrightarrow{\epsilon} q_5 \xrightarrow{\epsilon} q_6 \xrightarrow{a_1:A} q_8 \xrightarrow{\epsilon} q_{10} \xrightarrow{\epsilon} q_5 \xrightarrow{\epsilon} q_7 \xrightarrow{b_{12}:B} q_9 \xrightarrow{\epsilon} q_{10} \xrightarrow{e:\epsilon} q_{11}$ with output $O(p_2) = AB$. Thus, $G_{\mathcal{A}_{F_{ex}}}(T_1) = \{ a_1 B, AB \}$, as desired in Example 6.1. There is no accepting run for T_2 so that $G_{\mathcal{A}_{F_{ex}}}(T_2) = \emptyset$. Observe that $\mathcal{A}_{F_{ex}}$ precisely outputs the P-sequences as desired in Example 6.1.*

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

Lemma 6.1. *Let $T \in \Sigma^*$ be an input sequence and $\mathcal{A}$ be an FST. For any run p of $\mathcal{A}$ for T, it holds $O(p) \sqsubseteq T$.*

Proof. The proof is by induction. For $T = \epsilon$, the assertion holds because every path for T must consist of only ϵ-transitions so that $G(p) = \epsilon \sqsubseteq T$. Now suppose that the assertion holds for some sequence $T' \in \Sigma^*$. We show that it then also holds for $T = T'w$, $w \in \Sigma$. Let p be any path for T and set $S = O(p)$. We decompose p into two sequences of transitions: a path p' for T' with output S' and a remainder p_w with output s_w. This decomposition is always possible. We have $S = S's_w$. Since p' is a path for T', $S' \sqsubseteq T'$ by the induction hypothesis. Now observe that p_w must contain exactly one transition with input label w and that all other transitions must

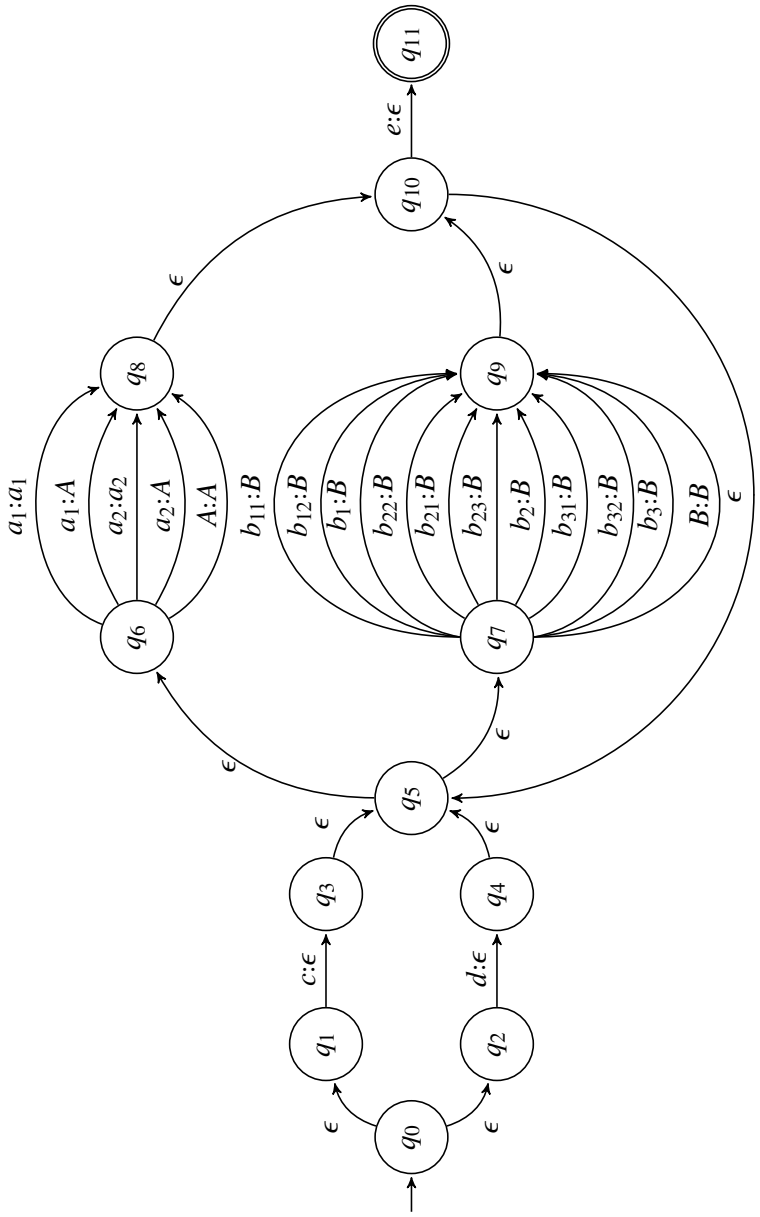


Figure 6.2: *FST for the pattern expression* $[c|d]([A^{\uparrow} \mid B^{\uparrow}_{=}]^{+})e$.

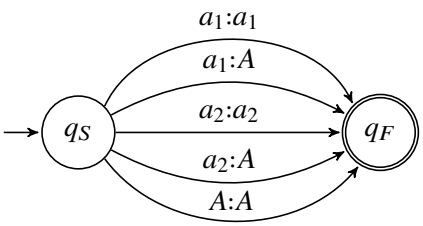


Figure 6.3: *FST for basic item expression* $(A^{\uparrow})$

be ϵ-transitions; otherwise p would not be a path 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 $w' \in \text{anc}(w) \cup \{\epsilon\}$, 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; e.g., there is no FST for predicate "all subsequences of form a^*b^* with an equal number of a's and b's". FST are a good trade-off between expressiveness and computational complexity, however: they can express all pattern expressions that occur in practice and they lend themselves to efficient mining algorithms that we will discuss in Chapter 7.

6.3.2 Translating pattern expressions

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 say that an expression E *matches* an input sequence T if there is an accepting path for T in $\mathcal{A}(E)$. Similarly, E *produces* (or, more specifically, can produce) sequence S for input T when there is an accepting path p for T with $O(p) = S$. Finally, E *generates* for T the set of subsequences $G_E(T) \stackrel{\text{def}}{=} G_{\mathcal{A}(E)}(T)$.

Each item expression is translated into a two-state FST with $Q = \{q_S, q_F\}$, where q_S is the initial and q_F the final state. Figure 6.3 shows an example translation for item expression $(A^{\uparrow})$. The transitions of the FST depend on the item expression and are summarized in Table 6.1 (page 88), column "FST".

The translation rules for composite expressions mirror the rules for translating regular expressions to nondeterministic finite state automaton [Thompson (1968)] for operators ? (optionality), $*$ (Kleene star), + (Kleene plus), | (union) and concatenation. We briefly discuss these translation rules; we maintain the invariant that each translated FST has exactly one final state.

- $\mathcal{A}([E]) = \mathcal{A}(E)$.
- $\mathcal{A}([E]?)$: Take the FST $\mathcal{A}(E)$ and add an ϵ-transition from the initial state to the final state.

- $\mathcal{A}([E]^*)$: Take the FST $\mathcal{A}(E)$ and add an ϵ-transition from the initial state to the final state and vice versa.
- $\mathcal{A}([E]^+)$: Take the FST $\mathcal{A}(E)$ and add an ϵ-transition from the final state to the initial state.
- $\mathcal{A}([E_1 E_2])$: Take the FSTs $\mathcal{A}(E_1)$ and $\mathcal{A}(E_2)$ and connect the final state of $\mathcal{A}(E_1)$ to the initial state of $\mathcal{A}(E_2)$ via an ϵ-transition. Take as initial state the initial state of $\mathcal{A}(E_1)$ and take as final state the final state of $\mathcal{A}(E_2)$.
- $\mathcal{A}([E_1|E_2])$: Take the FSTs $\mathcal{A}(E_1)$ and $\mathcal{A}(E_2)$. Create a new initial state q_S and a new final state q_F. Add ϵ-transitions from q_S to the initial states of $\mathcal{A}(E_1)$ and $\mathcal{A}(E_2)$. Add ϵ-transitions to q_F from final states of $\mathcal{A}(E_1)$ and $\mathcal{A}(E_2)$. Take q_F as the only final state.
- $\mathcal{A}([E]\{n,m\}) \stackrel{\text{def}}{=} \mathcal{A}(\underbrace{[E][E]\ldots[E]}_{n \text{ times}}\underbrace{[[E][[E][\ldots]?]?}_{m-n \text{ times}})$. Thus E is "repeated" between n and m times.
- $\mathcal{A}([E]\{n\}) \stackrel{\text{def}}{=} \mathcal{A}([E]\{n,n\})$.
- $\mathcal{A}([E]\{n,\}) \stackrel{\text{def}}{=} \mathcal{A}([[E]\{n\}[E]^*])$.

For example, if we translate expression E_{ex} of Equation (6.2) using above rules we obtain the FST show in Figure 6.2. 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.

6.4 Advanced Pattern Expression Language

Our pattern expressions are expressive enough to express several subsequence constraints. Their are limitations, however. For example, the expression N_1 in Table 6.3 (page 90) matches and produces verbal phrases between entities. In some applications, it might be more desirable to produce verbal phrases where verbs are generalized to their lemmas. As another example, consider the expression N_2 that produces typed relational phrases. Here the subexpression ENTITY$^{\uparrow}$ matches all entities and produces all ancestors of the matched entity up to ENTITY. N_2 along with typed relational phrases (e.g., PERSON lives in LOCATION) will also produce "over generalized" patterns (e.g., ENTITY lives in ENTITY) or "under generalized" patterns (e.g., Barack Obama lives in Washington) which might not be of interest to applications. As a final example, consider expression A_1 in Table 6.4, which produces sequences of electronic products. For a specific application, we might be interested in only producing sequences of electronic products that have price more than say 500. Such highly customized application specific requirements cannot be addressed by pattern expressions because:

- There is a limited way to express which items an item expression can match. For example, item expressions can either match only one item ($w_=$), or all descendants of an item (w), or any item using wild card (.). And,

- There is a limited way to express which items an item expression should produce. For a matched input item, basic item expressions produce either all ancestors ($.^{\uparrow}$), or ancestors up to the item described in the input label ($w^{\uparrow}$), or exactly the item described by the input label ($w^{\uparrow}_{=}$).

One approach to address above requirements using pattern expressions is to create a custom hierarchy for each application/pattern expression. Such an approach is generally cumbersome and inconvenient in practice.

6.4.1 Advanced Pattern Expressions

We now propose advanced pattern expressions which allows us to exercise more control and convenience in expressing which items should be matched by an item expression and which items should be produced.

Annotated hierarchy

To increase expressibility of item expressions, we leverage an *annotated hierarchy* in which each item $w \in \Sigma$ is annotated with zero or more attributes. For example, in a syntactic hierarchy—where a word can generalize to its lemma, which in turn can generalize to its POS-tag (e.g., lives⇒live⇒VERB)—attributes could be item name and level. In a product hierarchy, attributes could be product name, price, manufacturing company, etc. Figure 6.4 shows a variant of the example hierarchy of Figure 7.1b annotated with some attributes. We will use this as an example through out this section.

More formally, denote by $\mathscr{A} = \{\alpha_1, \alpha_2, \ldots, \alpha_{|\mathscr{A}|}\}$ set of attributes and by $\mathcal{D}_\alpha$ the domain of attribute $\alpha \in \mathscr{A}$. For our example hierarchy, we have $\mathscr{A}_{ex} = \{\text{name}, \text{level}, \text{type}\}$ and $\mathcal{D}_{\text{level}} = \{\text{R}, \text{M}, \text{L}\}$. For an item $w \in \Sigma$, we denote by $\alpha(w) \in \mathcal{D}_\alpha$ the value of attribute α for item w. For example, we have level(A) = R.

Item Descriptor

We introduce the notion of an *item descriptor*, which allows for flexibility in describing items. Item descriptors are defined inductively as follows:

- For all items $w \in \Sigma$, w is an item descriptor.
- For all attributes $\alpha \in \mathscr{A}$, $\alpha = v$ is an item descriptor where $v \in \mathcal{D}_\alpha$.
- P is an item descriptor where $P : \Sigma \to \{\text{true}, \text{false}\}$ is a user-defined predicate on items in Σ.
- If $\mathscr{I}$ is an item descriptor, then $(\mathscr{I})$ and $\neg(\mathscr{I})$ are also item descriptors.
- If $\mathscr{I}_1$ and $\mathscr{I}_2$ are item descriptors, then $(\mathscr{I}_1 \wedge \mathscr{I}_2)$ and $(\mathscr{I}_1 \vee \mathscr{I}_2)$ are also item descriptors.

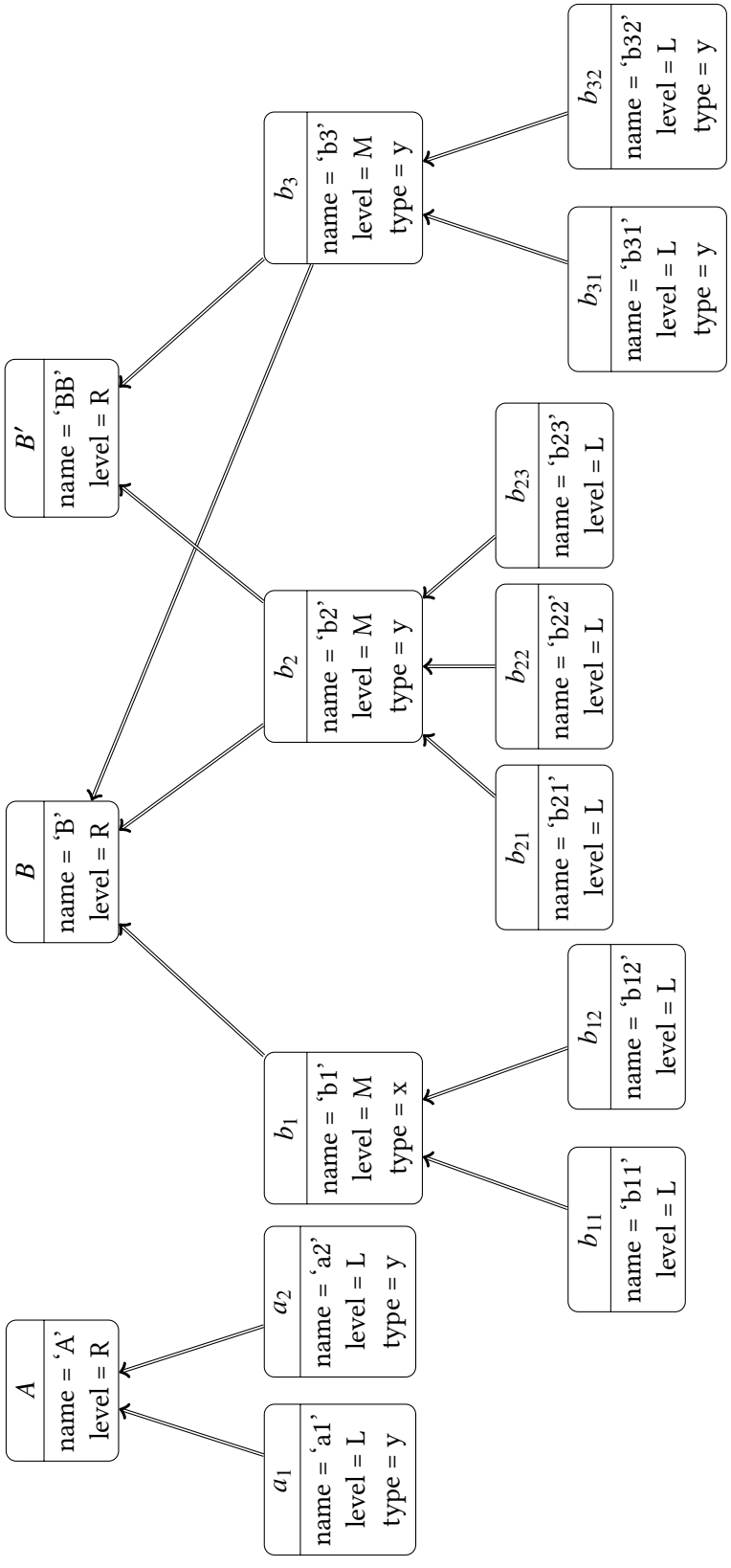


Figure 6.4: *Example annotated hierarchy.*

$\mathscr{I}$	$\Sigma_{\mathscr{I}}$	$\mathscr{I}_{ex}$	$\Sigma_{\mathscr{I}_{ex}}$
w	$\{ w \}$	B	$\{ B \}$
$\alpha = \nu$	$\{ w \mid w(\alpha) = \nu \}$	level = R	$\{ A, B, B' \}$
P	$\{ w \mid P(w) \}$	regex.match(name,'a*')	$\{ a_1, a_2 \}$
$\neg(\mathscr{I}_1)$	$\{ w \mid w \in \Sigma \setminus \Sigma_{\mathscr{I}_1} \}$	$\neg$(level = L)	$\{A, B, B', b_1, b_2, b_3, c, d, e\}$
$(\mathscr{I}_1 \wedge \mathscr{I}_2)$	$\{ w \mid w \in \Sigma_{\mathscr{I}_1} \cap \Sigma_{\mathscr{I}_2} \}$	level = M $\wedge$ type = y	$\{ b_2, b_3 \}$
$(\mathscr{I}_1 \vee \mathscr{I}_2)$	$\{ w \mid w \in \Sigma_{\mathscr{I}_1} \cup \Sigma_{\mathscr{I}_2} \}$	level = M $\vee$ type = y	$\{ a_1, a_2, b_1, b_2, b_3 \}$

Table 6.5: *Item descriptors and their corresponding examples.*

Denote by $\Sigma_{\mathscr{I}} \subseteq \Sigma$ the set of items described by item descriptor $\mathscr{I}$. Table 6.5 gives an overview of item descriptors and their examples using our example annotated hierarchy of Figure 6.4. The most basic item descriptor is w, which describes the item w. Sometimes we may want to describe all items that have a specific attribute-value in common (e.g., all items that are at a certain level in the hierarchy). Descriptor $\alpha = \nu$ serves this purpose; it describes all items w for which $\alpha(w) = \nu$. We also support user-defined predicate on Σ. For example, we may want to describe words in natural language text with certain inflection (e.g., words that end with "er" or "ier", or describe products with price greater than a certain value). Item descriptor P describes all items $w \in \Sigma$ for which $P(w)$ holds. Item descriptors can also be combined using logical not ($\neg$), logical and ($\wedge$), and logical or ($\vee$) operators to describe certain items (see examples in Table 6.5).

Advanced item expressions

Like item expressions, advanced item expressions also apply to one item. They match one item and produce zero or one item. The advanced language consists of the following set of item expressions, defined inductively:

- If $\mathscr{I}$ is an item descriptor, $[\mathscr{I}]$ and $[\mathscr{I}]_=$ are item expressions.
- $[\,]$ is an item expression.
- If I_1 and I_2 are item expressions, then $[I_1 \cap I_2]$ is an item expression.
- If I and O are item expressions, then $I^{\uparrow}O$ is an item expression.

Table 6.6 provides an overview our advanced item expressions. We denote by I an item expression and by Σ_I the set of items that the expression can match. For now assume that all item expressions are "captured", i.e., they produce one or more output items. The item expression $[\mathscr{I}]_=$ matches items $w \in \Sigma_{\mathscr{I}}$ described by the item descriptor $\mathscr{I}$ and produces the matched item w. For example, if $I = [B]_=$, then $G_I(B) = \{ B \}$ and $G_I(b_2) = \emptyset$. If $I = [\text{level} = \text{M} \wedge \text{type} = \text{y}]_=$, then $G_I(B) = \emptyset$ and $G_I(b_2) = \{ b_2 \}$. To match all descendants of items described an item descriptor $\mathscr{I}$,

Expr. (I)	Transl. type	matches (Σ_I)	produces	FST
$[\mathscr{I}]_=$	Uncaptured	$w \in \Sigma_{\mathscr{I}}$	ϵ	$\left\{ q_S \xrightarrow{w:\epsilon} q_F \mid w \in \Sigma_{\mathscr{I}} \right\}$
	Captured	$w \in \Sigma_{\mathscr{I}}$	w	$\left\{ q_S \xrightarrow{w:w} q_F \mid w \in \Sigma_{\mathscr{I}} \right\}$
$[\mathscr{I}]$	Uncaptured	$w \in \text{desc}(\Sigma_{\mathscr{I}})$	ϵ	$\left\{ q_S \xrightarrow{w:\epsilon} q_F \mid w \in \text{desc}(\Sigma_{\mathscr{I}}) \right\}$
	Captured	$w \in \text{desc}(\Sigma_{\mathscr{I}})$	w	$\left\{ q_S \xrightarrow{w:w} q_F \mid w \in \text{desc}(\Sigma_{\mathscr{I}}) \right\}$
[ ]	Uncaptured	$w \in \Sigma$	ϵ	$\left\{ q_S \xrightarrow{w:\epsilon} q_F \mid w \in \Sigma \right\}$
	Captured	$w \in \Sigma$	w	$\left\{ q_S \xrightarrow{w:w} q_F \mid w \in \Sigma \right\}$
$[I_1 \cap I_2]$	Uncaptured	$w \in \Sigma_{I_1} \cap \Sigma_{I_2}$	ϵ	$\left\{ q_S \xrightarrow{w:\epsilon} q_F \mid w \in \Sigma_{I_1} \cap \Sigma_{I_2} \right\}$
	Captured	$w \in \Sigma_{I_1} \cap \Sigma_{I_2}$	w	$\left\{ q_S \xrightarrow{w:w} q_F \mid w \in \Sigma_{I_1} \cap \Sigma_{I_2} \right\}$
$I{:}O$	Captured	$w \in \Sigma_I$	$\{ \text{anc}(w) \cap \Sigma_O \}$	$\left\{ q_S \xrightarrow{w:w'} q_F \mid w \in \Sigma_I, w' \in \text{anc}(w) \cap \Sigma_O \right\}$

Table 6.6: *Item expressions for advanced pattern expression language.*

Basic item expressions	Advanced item expressions
$w_{=}$	$[w]_{=}$
w	$[w]$
.	[ ]
$w^{\uparrow}$	$[w]$:$[w]$
$.^{\uparrow}$	[ ]:[ ]
$w_{=}^{\uparrow}$	$[w]$:$[w]_{=}$

Table 6.7: *Basic item expressions and their corresponding advanced item expressions.*

we use the item expression $[\mathscr{I}]$ (without =). Denote by

$$\text{desc}(\Sigma_{\mathscr{I}}) = \{\, w \mid w \in \text{desc}(w'), w' \in \Sigma_{\mathscr{I}} \,\}$$

the descendants of items described by item descriptor $\mathscr{I}$. The item expression $[\mathscr{I}]$ matches all items in $\text{desc}(\Sigma_{\mathscr{I}})$. For example, if $I = [B]$, then $G_I(b_2) = \{\, b_2 \,\}$. As another example, if $I = [\text{level} = \text{M} \wedge \text{type} = \text{y}]$, then $G_I(b_2) = \{\, b_2 \,\}$ and $G_I(b_{21}) = \{\, b_{21} \,\}$. The advanced language also provides the item expression [ ] for supporting wild cards, .i.e., to match any item. For example, if $I = [\,]$, then $G_I(B) = \{\, B \,\}$, $G_I(b_1) = \{\, b_1 \,\}$. Sometimes we want to match some specific items (e.g., all words that are adjectives and end with "er"). For this purpose, item expressions can be combined using the set intersection operator ($\cap$) to form a "compound" item expression. For example, if $I = [[B] \cap [\text{type} = \text{y}]_{=}]$, then $G_I(a_1) = \emptyset$, $G_I(b_1) = \emptyset$, $G_I(b_2) = \{\, b_2 \,\}$, $G_I(b_{21}) = \emptyset$, and $G_I(b_{31}) = \{\, b_{31} \,\}$.

Until now we only considered advanced item expressions that produce exactly the matched item. To support mining generalized sequences in controlled way, we make use of item expressions of the form $I^{\uparrow}O$, where the expression I defines the items that can be matched and the expression O defines the items that can be produced. Note that semantics of an expression are different when it is used define the output; see column "produces" in Table 6.8 for item expression of the form $I^{\uparrow}O$. For example, for the item expression $I = [b_2]^{\uparrow}[\,]$, we have $G_I(b_{21}) = \{\, b_{21}, b_2, B, B' \,\}$. If $I = [b_2]^{\uparrow}[\text{level} = \text{M}]$, we have $G_I(b_{21}) = \{\, b_{21}, b_2 \,\}$ and if $I = [b_2]^{\uparrow}[\text{level} = \text{M}]_{=}$, we have $G_I(b_{21}) = \{\, b_2 \,\}$.

Finally, Table 6.7 illustrates advanced item expressions that have the same semantics as our basic item expressions.

Composite expressions

Advanced item expressions can be arbitrarily combined with regular expression operators to form pattern expressions that match sequence of items. More formally, advanced pattern expressions are defined inductively as follows:

- An item expression is a pattern expression.

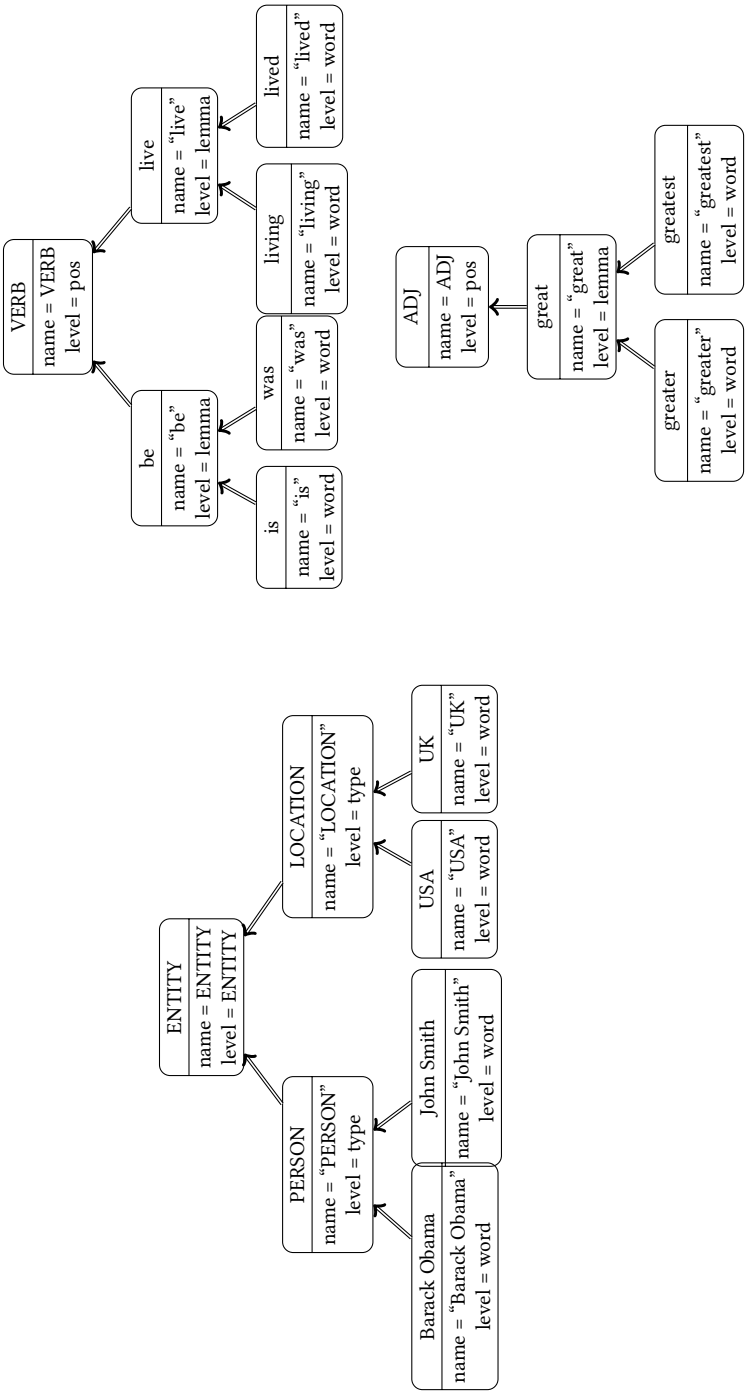


Figure 6.5: *Excerpt of an annotated item hierarchy for text mining applications.*

Advanced pattern Expression	*Description*
X_1: [VERB][PREP]	*Verb, preposition*
X_2: [VERB]:[VERB] [PREP]	*Verb (and its generalizations), preposition*
X_3: [VERB]:[level=lemma] [PREP]	*Verb and its generalizations up to lemma, preposition*
X_4: [VERB]:[level=lemma]$_=$ [PREP]	*Lemmatized verb, preposition*
X_5: -[ENTITY] [VERB]$^\uparrow$[level=lemma]$^{\pm}_{=}$[NOUN]*[PREP]? -[ENTITY]	*Relational phrases (with lemmatized verbs) between entities*
X_6: [ENTITY]$^\uparrow$[level=type]$_=$ [VERB]$^+$[NOUN]*[PREP]? [ENTITY]$^\uparrow$[level=type]$_=$	*Typed relational phrases (entities generalized to their types)*
X_7: [NOUN] -[be] [[ADJ] ∩ [regex.match(name,".*er|.*ier")] -[than]$_=$ [NOUN]	*Comparative facts*

Table 6.8: *Some examples of advanced pattern expressions useful for text mining applications.*

- If E is a pattern expression, so are $-E$, (E), $(E)^*$, $(E)^+$, $(E)?$, and for all $n, m \in \mathbb{N}$ with $n \leq m$, $(E)\{n\}$, $(E)\{n,\}$, and $(E)\{n,m\}$.
- If E_1 and E_2 are pattern expressions, so are (E_1E_2) and $(E_1|E_2)$.

In advanced pattern expressions, we use the usual precedence rules for regular expression operators to suppress parenthesis but not "-". Operators that appear earlier in the above definition have higher precedence. We use "-" operator to specify uncaptured expressions, i.e., subexpressions preceded with a "-" produce an empty output. For example, the expression

$$E_{adv} = -([c] \mid [d])([A]{:}[A] \mid [B]{:}[B]_=)^+ - [e]$$

describes the subsequence constraint of Example 6.1.

6.4.2 Examples

Table 6.8 shows some example advanced pattern expressions useful for text mining applications. Here we considered an annotated item hierarchy of which an excerpt is shown in Figure 6.5. Expressions X_1–X_6 illustrate how we can exercise more control over producing generalized sequences. For example, X_3 produces generalized sequences in which verbs are generalized only up to their lemmas. Similarly, X_5 produces relational phrases in which verbs are lemmatized and X_6 produces typed relational phrases in which entities are generalized to their types. Finally, expression X_7 illustrates how we can conveniently express which items to match. This expression produces phrases with comparative adjectives (that end with 'er' or 'ier'), which are useful for some information extraction applications (e.g., [Tandon et al. (2014)]).

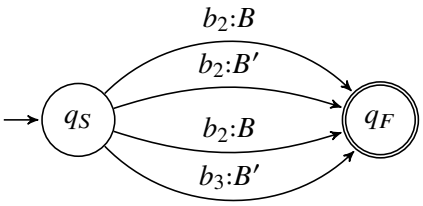


Figure 6.6: *FST for advanced item expression [level=M ∧ type=y]$^{\uparrow}_{=}$[level=R]$_{=}$.*

6.4.3 Translating Advanced Pattern Expression to FSTs

Advanced pattern expressions are translated to FSTs in a similar way as described in Section 6.3.2. The key difference lies is in translation rules for advanced item expressions. Like item expressions, advanced item expressions are translated into a two state FST. The transitions are summarized in Table. 6.6, column "FST". Figure 6.6 shows an example translation. Translation rules for composite expressions remain unmodified.

6.5 Summary

In this chapter, we introduced subsequence predicates as a general mechanism for unifying and extending subsequence constraints. We described our basic as well advanced pattern expressions as a simple and intuitive way to express subsequence constraints and proposed finite state transducers as an underlying computational model.

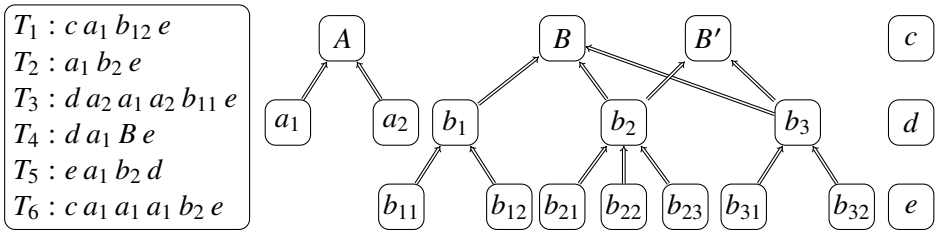
Chapter 7

Frequent Sequence Mining with Subsequence Constraints

In this chapter,[a] we focus on mining P-frequent sequences, where the subsequence predicate P is expressed using a pattern expression. We restrict our attention to pattern expressions; our methods can be extended to deal with advanced pattern expressions as well. We propose the DESQ system, which translates a given pattern expression to a *compressed* FST, which is subsequently optimized and simulated in a way suitable for frequent sequence mining.

In Section 7.1, we review the problem of mining P-frequent sequences. In Section 7.2, we provide methods, compress and optimize our specialized FSTs. In particular, we discuss compressed FST and a simulation algorithm that effectively handles large hierarchies. Although traditional FST libraries such as Open-FST [Allauzen et al. (2007)] can be used within DESQ, our compressed FSTs support more efficient mining. In Section 7.3, we discuss two efficient mining algorithms named DESQ-COUNT and DESQ-DFS. DESQ-COUNT is a match-and-count algorithm that aims at highly selective subsequence constraints, whereas DESQ-DFS can handle more demanding pattern expressions and is based on depth-first search approach described in Section 2.2.2. In Section 7.4, we propose techniques to improve simulation efficiency of compressed FSTs. In general, our FSTs are often non-deterministic and existing optimization methods (determization and minimization) do not apply. Our techniques aim to reduce overall nondeterminism by minimizing cFSTs, by pruning input sequences that do not produce an output, and by pruning non-accepting paths.

[a] The material in this chapter is based on Beedkar and Gemulla (2016).



(a) Example database **(b)** Example hierarchy

Figure 7.1: *A sequence database and its vocabulary*

7.1 FSM and Subsequence Predicates (recap)

We start by reviewing the problem of mining P-frequent sequences introduced in Section 6.1 (page 84).

Consider the example database and hierarchy of previous chapter (reproduced in Figure 7.1) and the pattern expression

$$E_{ex} = [c|d]([A^{\uparrow} \mid B^{\uparrow}_{=}]^{+})e. \tag{7.1}$$

The pattern expression describes consecutive subsequences of A's or its descendants and B's that occur in the input sequences between c or d and e. Given a subsequence predicate P, which is expressed using pattern expression, our goal is mine P-frequent sequences, i.e., P-sequences that have P-frequency $\geq \sigma$. In our example sequence database of Figure 7.1 and for the above expression E_{ex}, the set of all P-frequent sequences (along with their P-frequencies) for $\sigma = 2$ in is given by

$$\{\, AAAB : 2, AB : 2, Aa_1AB : 2, a_1B : 2 \,\}.$$

7.2 FST Optimizations

Our translation rules for pattern expressions 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 item expression "$.^{\uparrow}$" has $\Theta(nd)$ transitions. To avoid this explosion of FST size and support efficient mining, we make use of a compressed FST (cFST) representation for this purpose.

7.2.1 Compressed FST

Table 7.1 list translations rules to obtain compressed FST for item expressions. Note that both FST and cFST translations have the same from- and to-state, but cFST of an item expression has exactly one transition. Each transition in the cFST describes a set of transitions in the corresponding FST in a concise way. More specifically, cFSTs use as input labels ., w, and $w_{=}$ for all $w \in \Sigma$. Here "." matches all input items,

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

Table 7.1: *Translation rules for item expressions (where $w, w', w'' \in \Sigma$) to compressed FST.*

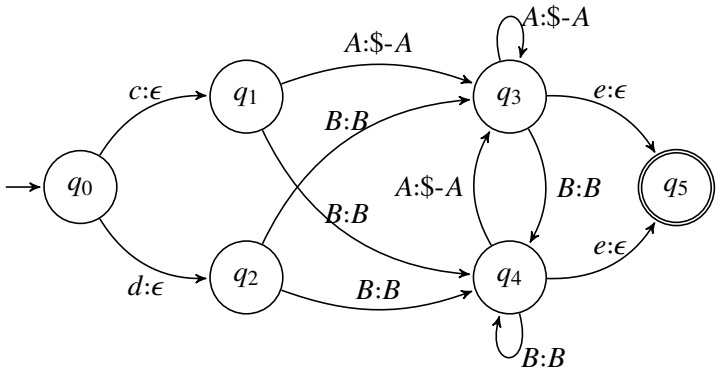


Figure 7.2: *Compressed FST for* $[c|d]([A^{\uparrow} \mid B^{\hat{\uparrow}}_{=}]^{+})e$.

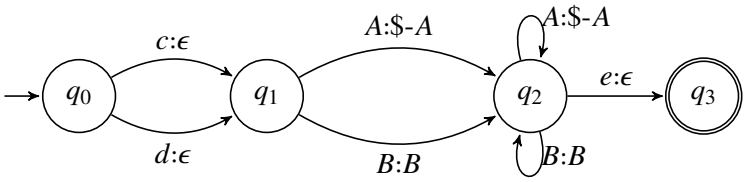


Figure 7.3: *Minimized cFST for* $[c|d]([A^{\uparrow} \mid B^{\hat{\uparrow}}_{=}]^{+})e$.

w matches all items in desc(w), and $w_{=}$ matches only item w. cFSTs use as output labels ϵ, w, \$, \$-w, and \$-$\top$ for $w \in \Sigma$. Each transition encodes the set of output labels in the corresponding FST: ϵ and w are as before, \$ encodes the matched input item, \$-w the matched input item and all its ancestors that are descendants of w, and \$-$\top$ the matched item and all its ancestors. The cFST translations for composite expressions remain unmodified.

Figure 7.2 shows the cFST $\mathcal{A}_{ex}$ for pattern expression $[c|d]([A^{\uparrow} \mid B^{\hat{\uparrow}}_{=}]^{+})e$. Observe that the cFST has fewer transitions than its uncompressed counterpart of Figure 6.2 on page 92. 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 cFSTs, which further reduce the number of states and transitions; we will discuss on how to minimize cFSTs later in Section 7.4.1. For our running example, we finally obtain the cFST shown in Figure 7.3.

7.2.2 Simulating compressed FST

We now discuss how to compute via simulation the set $G_{\mathcal{A}}(T)$ of all output sequences generated by a cFST $\mathcal{A}$ for input sequence T. Note that the computation of $G_{\mathcal{A}}(T)$ for all $T \in \mathcal{D}$ can be infeasible. Nevertheless, simulation forms the basis of the more efficient DESQ-DFS algorithm of Section. 7.3.3 so that we describe the ap-

proach briefly. We assume throughout (and without loss of generality) that $\mathcal{A}$ does not have ϵ-transitions.

Algorithm 7.1 shows how to simulate cFST $\mathcal{A} = (Q, q_s, Q_F, \Sigma, \Delta)$, where the *transition function*

$$\delta(q, w) = \{ (out, q_{to}) \mid (q, in, out, q_{to}) \in \Delta,\ in \text{ matches } w \} \tag{7.2}$$

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 7.1). Intuitively, we simulate the cFST by starting with the initial state q_S of the cFST (line 2) and repeatedly selecting a transition for which the input label matches the next input item t_{pos} (line 8). If there are multiple such transitions, 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 of the selected transitions to an output buffer (S, lines 9–20). As before, if a transition encodes more than one output item, we append them one by one (again via backtracking, lines 17–19).[b] To keep notation concise, we define $\text{desc}(\top) = \Sigma$. If we reach a final state after consuming all input items, we output the buffer, which then contains a generated sequence (lines 5–7).

Consider the sequence $T_3 = da_2a_1a_2b_{11}e$ of our example database $\mathcal{D}_{ex}$ (Figure 7.1a and the cFST $\mathcal{A}_{ex}$ of Figure 7.3. In the first invocation of Step, we have $q = q_0$, $t_{pos} = t_1 = d$, and $S = \epsilon$. Since $\delta(q_0, d) = \{ (\epsilon, q_1) \}$, we proceed to line 11 and invoke Step with $q = q_1, t_{pos} = t_2 = a_2, S = \epsilon$. We have $\delta(q_1, a_2) = \{ (\$\text{-}A, q_2) \}$, so that we proceed to line 18 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 $\delta(q_2, e) = \{ (\epsilon, q_3) \}$, 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 $G_{\mathcal{A}_{ex}}(T_3)$ in line 6. The algorithm then backtracks and generates sequences a_2a_1AB, a_2Aa_2B, a_2AAB, Aa_1a_2B, Aa_1AB, AAa_2B, and $AAAB$.

Partial matches

The simulation algorithm only generates an output when the entire input sequence is matched. If we are interested in matching pattern expressions that occur somewhere in the input sequence instead, we construct a cFST for $.^*E$ (instead of for E) and modify the above simulation such that it adds the buffered output to $G_{\mathcal{A}}(T)$ whenever a final state is reached, whether or not the entire input has been consumed (i.e., we omit the condition $pos > |T|$ in line 5.[c]

[b] 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 Sec. 7.3.

[c] This approach is more efficient than using expression $.^*E.^*$ for constructing the cFST.

Algorithm 7.1 Simulate a cFST

Require: cFST $\mathcal{A} = (Q, q_s, Q_F, \Sigma, \Delta)$, $T = t_1 \dots t_{|T|}$
Ensure: $G_{\mathcal{A}}(T)$

```
1:  G_A(T) ← ∅                                          // set of generated sequences
2:  Step(q_S, 1, ϵ)
3:
4:  void Step(q, pos, S):                               // (current state, input pos., buffer)
5:  if q ∈ Q_F and pos > |T| and S ≠ ϵ then
6:     G_A(T) ← G_A(T) ∪ { S }
7:  end if
8:  for all (out, q_to) ∈ δ(q, t_pos) do                // empty if pos > |T|
9:     switch (out)
10:       case ϵ:
11:          Step(q_to, pos + 1, S)
12:       case w:
13:          Step(q_to, pos + 1, Sw)
14:       case $:
15:          Step(q_to, pos + 1, St_pos)
16:       case $-x for x ∈ Σ ∪ {⊤}:
17:          for all w′ ∈ anc(t_pos) ∩ desc(x) do
18:             Step(q_to, pos + 1, Sw′)
19:          end for
20:    end switch
21: end for
```

Nondeterminism

Note that cFST simulation involves backtracking when multiple transitions match the same input item and/or a transition has an output label of form \$-w or \$-$\top$. The standard way to avoid non-determinism is to use some form of FST determinization [Mohri (1997)]. However, these methods do not directly apply to our FSTs. We discuss methods that aim to reduce nondeterminism in Sec. 7.4.

7.3 Mining P-Frequent Sequences

We now turn attention to mining P-frequent sequences from a sequence database. We assume that subsequence predicate P is described by a cFST $\mathcal{A}$ (e.g., obtained by translating a pattern expression). 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 [Pei et al. (2001, 2002)] and is generally more efficient than DESQ-COUNT when the set of P-generated sequences is large.

7.3.1 Naïve Approach

The naïve "generate-and-count" approach is to compute $G_{\mathcal{A}}(T)$ for each input sequence $T \in \mathcal{D}$ via cFST simulation and count how often each sequence has been generated (cf. Equation (6.1)). The naïve approach is outlined as Algorithm 7.2; it is generally inefficient because it considers many globally infrequent sequences. For example, we obtain

$$\begin{aligned} G_{A_{ex}}(T_3) = \{&AAAB, AAa_2B, Aa_1AB, Aa_1a_2B, \\ &a_2AAB, a_2Aa_2B, a_2a_1AB, a_2a_1a_2B\} \end{aligned} \tag{7.3}$$

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

7.3.2 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 $f(w, \mathcal{D}) = |\{T \in \mathcal{D} \mid w \sqsubseteq T\}|$ the *frequency* of item w. We say that item w is *frequent* if $f(w, \mathcal{D}) \geq \sigma$. Similar to many prior FSM algorithms, DESQ-COUNT first generates an *f-list* F, which contains all frequent items along with their frequency. For our example database, we obtain f-list

$$F_{ex} = \{A{:}6, e{:}6, B{:}6, a_1{:}6, d{:}3, b_2{:}3, b_1{:}2, c{:}2, b_{12}{:}1, b_{11}{:}1, a_2{:}1\}. \tag{7.4}$$

Algorithm 7.2 Naïve approach

Require: $\mathscr{D}$, cFST $\mathcal{A} = (Q, q_s, Q_F, \Sigma, \Delta)$, σ
Ensure: P-frequent sequences for $\mathcal{A}$ in $\mathscr{D}$

```
1:  M ← ∅                                   // A map from sequence to its frequency
2:  for each T ∈ 𝒟 do
3:     Compute G_A(T) via cFST simulation    // using Algorithm 7.1
4:     for each S ∈ G_A(T) do
5:        M[S] ← M[S] + 1
6:     end for
7:  end for
8:  for each S ∈ Keys(M) do
9:     if M[S] ≥ σ then
10:       Output(S,M[S])
11:    end if
12: end for
```

Note that the f-list is independent of the notion of 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 $G^{F_{ex}}_{\mathcal{A}_{ex}}(T_3) = \{\, AAAB, Aa_1AB \}$, which is much smaller than the full set $G_{\mathcal{A}_{ex}}(T_3)$ 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 6.1, which states that FSTs specify subsequence predicates, and the following observation.

Lemma 7.1. *(Item antimonotonicity) Let P be a relevance predicate and $S \in \Sigma^+$ be any sequence. Then for all $w \in S$, $f(w, \mathscr{D}) \geq f_P(S, \mathscr{D})$.*

Proof. Pick any $w \in S$ and input sequence $T \in \mathscr{D}$ such that $S \in G_P(T)$. Since P is a relevance 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 $G_\mathcal{A}(T)$ so that DESQ-COUNT is correct.

As mentioned above, we directly compute the reduced set $G^F_{\mathcal{A}}(T)$ by adapting cFST simulation (Algorithm 7.1) to work with the f-list. In more detail, we stop exploring a path through the cFST (via Step) as soon as an infrequent item is produced. To do so, we execute lines 13, 15, and 18 of Algorithm 7.1 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, addresses such cases.

7.3.3 DESQ-DFS

DESQ-DFS adapts the pattern-growth framework of PrefixSpan [Pei et al. (2001)] to FSTs. Recall the DFS approach from Section 2.2.2. Pattern-growth approaches 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 a performing series of *expansions*. In each expansion, a frequent sequence S (of l items) is expanded to generate sequences with prefix S (of $l+1$ items), 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 cFSTs is shown as Algorithm 7.3 and illustrated on our running example in Figure 7.4.

Projected databases

For a 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 FST simulation is in, and which output has been produced so far. In more detail, suppose that we simulate an $\mathcal{A}$ on input sequence $T = t_1 \dots t_n$. Consider a partial run $p = p_1 \dots p_m$ obtained after $m \leq n$ steps. We generated output $S = O(p)$ and, under our running assumption that $\mathcal{A}$ does not contain ϵ-transitions, consumed prefix $T' = t_1 \dots t_m$ 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), 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 (current state of $\mathcal{A}$). The *projected database* for S consists of all snapshots for S and is given by

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

Figure 7.4b 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 (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:

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

In our example, $\text{Presup}_{\mathcal{A}_{F7.4}}(a_1, \mathscr{D}_{ex}) = \{T_1, T_4, T_6\}$. Note that even if an input sequence has multiple snapshots for S, it contributes only once to the prefix support.

Expansions

We start with root node ϵ and all snapshots for ϵ (lines 1 and 2) and then perform a series of expansions (lines 3 and 14). In each expansion, we scan the projected database sequentially. For each snapshot $T[pos@q]$ (lines 6–7), we resume the FST for T at item t_{pos} in state q (via IncStep, lines 18–35). The transducer is stopped as soon as an 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 27, 29, and 32). In the later case, if we end up in a final state (lines 19–20), 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 (7.4)). 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

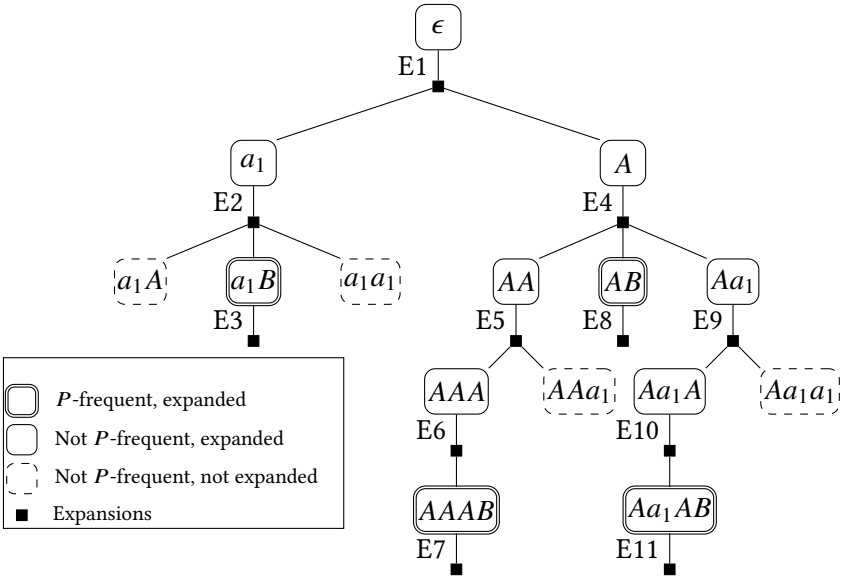
Note that the size of the prefix support is monotonically decreasing as we go down the tree but always stays at least as large as the support. This property, which we establish next, is key to the correctness of DESQ-DFS.

Lemma 7.2. *For any 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 7.3 DESQ-DFS

Require: $\mathscr{D}$, cFST $\mathcal{A} = (Q, q_S, Q_F, \Sigma, \Delta)$, σ, f-list F
Ensure: P-frequent sequences for $\mathcal{A}$ in $\mathscr{D}$

```
1:  S ← ϵ                                   // create root node; initially fields S.Proj = S.Sup = ∅
2:  S.Proj ← { T_1[1@q_S], ..., T_|D|[1@q_S] }
3:  Expand(S)
4:
5:  void Expand(S):
6:  for all T[pos@q] ∈ S.Proj do
7:    IncStep(T, pos, q, S)
8:  end for
9:  if |S.Sup| ≥ σ then
10:   Output(S,|S.Sup|)
11: end if
12: for all S' ∈ S.Children do              // expand if prefix support large enough
13:   if |{ T | ∃pos, q: T[pos@q] ∈ S.Proj }| ≥ σ then
14:     Expand(S')
15:   end if
16: end for
17:
18: void IncStep(T, pos, q, S):
19: if q ∈ Q_F and pos > |T| and S ≠ ϵ then
20:   S.Sup ← S.Sup ∪ { T }                                  // initially empty
21: end if
22: for all (out, q_to) ∈ δ(q, t_pos) do
23:   switch (out)
24:     case ϵ:
25:       IncStep(T, pos+1, q_to, S)
26:     case w:
27:       if f(w, D) ≥ σ then Append(S, w, T, pos+1, q_to)
28:     case $:
29:       if f(t_pos, D) ≥ σ then Append(S, t_pos, T, pos+1, q_to)
30:     case $-x, x ∈ Σ ∪ { ⊤ }:
31:       for all w' ∈ anc(t_pos) ∩ desc(x) do
32:         if f(w', D) ≥ σ then Append(S, w', T, pos+1, q_to)
33:       end for
34:   end switch
35: end for
36:
37: void Append(S, w, T, pos, q):
38: S.Children ← S.Children ∪ { Sw }                // node Sw is created if new
39: Sw.Proj ← Sw.Proj ∪ { T[pos@q] }                                // initially empty
```



(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

Figure 7.4: *Illustration of DESQ-DFS for $\mathcal{D}_{ex}$, $\mathcal{A}_{Fex}$, and $\sigma = 2$*

Proof. Pick any $S \in \Sigma^*$, $w \in \Sigma$, and $T = t_1 \dots t_n \in \mathscr{D}$ with $T \in \text{Presup}_{\mathcal{A}}(Sw, \mathscr{D})$. Then there exists a run $p = p_1 \dots p_m$ for prefix $T' = t_1 \dots t_m$ 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 $p' = p_1 \dots p_{m'}$ 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 $t_1 \dots t_{m'}$, 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 7.1. *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 an FST and pick any sequence $S \in \Sigma^+$. We start with showing that Algorithm 7.3 correctly computes the P-support of S when expanding node S, i.e., $S.\text{Sup} = \text{Sup}_{\mathcal{A}}(S, \mathscr{D})$ after the expansion. First pick any $T \in \text{Sup}(S, \mathscr{D})$ with $T = t_1 \dots t_n$. Then there is an accepting run $p = p_1 \dots p_n$ for T. By arguments as in the proof of Lemma 7.2, there must be a smallest run $p' = p_1 \dots p_m$, $m \leq n$, 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 $T[pos@q_m] \in \text{Proj}_{\mathcal{A}}(S, \mathscr{D})$, where $pos = m+1$, and thus $T \in \text{Presup}(S, \mathscr{D})$. Since by definition $p_{m+1} \dots p_n$ must output ϵ, Algorithm 7.3 follows transitions $p_{m+1} \dots p_n$ without stopping when resuming snapshot $T[pos@q_m]$. By doing so, it consumes all the remaining items $t_{m+1} \dots t_n$ of T and reaches final state q_n. It thus includes T into $S.\text{Sup}$ (lines 19–20). Now pick $T \notin \text{Sup}_{\mathcal{A}}(S, \mathscr{D})$. Since there is no accepting run for T, Algorithm 7.3 cannot reach a final state after consuming T so that it does not include T into $S.\text{Sup}$. Putting both together, $S.\text{Sup} = \text{Sup}_{\mathcal{A}}(S, \mathscr{D})$ after expanding S, as desired. We conclude that Algorithm 7.3 computes the correct frequency $f_P(S, \mathscr{D}) = |\text{Sup}_{\mathcal{A}}(S, \mathscr{D})|$. S is output only if it is P-frequent (line 10). Note that for $S = \epsilon$, we have $\epsilon.\text{Sup} = \emptyset$ (see line 19) so that ϵ is not output.

Let $S \in \Sigma^+$ be a P-frequent sequence. It remains to show that Algorithm 7.3 reaches and expands node S. First observe that for any prefix S' of S, we have

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

Here the first inclusion follows from Lemma 7.2, and the second inclusion follows from the above arguments. Since S is P-frequent, we have $|\text{Sup}(S, \mathscr{D})| \geq \sigma$, which implies $|\text{Presup}(S', \mathscr{D})| \geq \sigma$. Since every node on the path from ϵ to S corresponds to a prefix of S, Algorithm 7.3 does not prune any of these nodes due to too low prefix support (line 14). To complete the proof, recall that S cannot contain an infrequent item by Lemma 7.1. Thus none of the nodes on the path from ϵ to S are pruned due to too low item frequency either (lines 27, 29, or 32). We conclude that Algorithm 7.3 reaches and expands node S. □

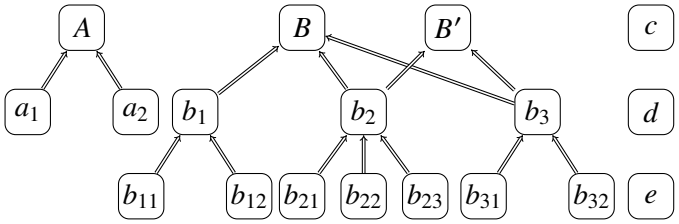


Figure 7.5: *An Example hierarchy.*

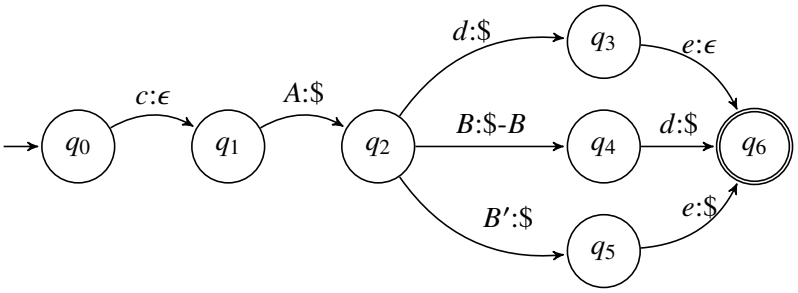


Figure 7.6: *cFST for pattern expression* $c\,(A)\,\big[(d)e \mid (B^{\uparrow}d) \mid (B'e)\big]$.

7.4 Reducing Nondeterminism

Recall from the discussion on nondeterminism in Section 7.2.2 that cFST 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 $\$\text{-}w$ or $\$\text{-}\top$. Backtracking is acceptable and in fact necessary for generating all output sequences (P-sequences). Recall the example in Section 7.2.2; the cFST simulation involved backtracking to generate all output sequences. Also observe that there was no "unnecessary" backtracking in that the backtracking always lead to an accepting path. However, this is not always the case. For example, consider our example hierarchy reproduced in Figure 7.5 and the cFST $\mathcal{A}_{F_{7.6}}$ shown in Figure 7.6. Here transitions $q_2 \xrightarrow{B:\$-B} q_4$ and $q_2 \xrightarrow{B':\$} q_5$ can match the input item b_2. Thus, we have $\delta(q_2, b_2) = \{\,(\$\text{-}B, q_4), (\$, q_5)\,\}$ and cFST simulation becomes nondeterministic and backtracking needs to be performed. For example, when we simulate $\mathcal{A}_{F_{7.6}}$ for the input sequence ca_1b_2e, transition $q_2 \xrightarrow{B:\$-B} q_4$ leads to non-accepting path. Such backtracking can be often expensive due to wasted computation for non-accepting paths.

The standard way to avoid nondeterminism is to use some form of FST determinization [Mohri (1997)]. In general, these methods do not directly apply to our FSTs because there are no sequential or even p-sequential transducers for some pattern expressions. An FST is *sequential* if for each input there is at most one output. Mohri (1997) showed that the classical powerset construction algorithm by Rabin and Scott (1959) for non-deterministic finite state automaton (NFA) can be extended to determinize sequential FSTs. Similarly, an FST is *p-subsequential* if there are

at most p outputs per input; these FSTs can be optimized by delaying output (and thus nondeterminism) until after the input has been consumed entirely. In our setting, such delayed output bars us from efficient mining (particularly in DESQ-DFS). Moreover, our FSTs are often not p-subsequential. For example, the number of outputs for expression $[.^*(.)]^+$ (all subsequences) is exponential in the input size and thus unbounded. For this reason avoiding nondeterminism without limiting out pattern language is challenging.

In what follows, we propose three techniques to reduce "unnecessary" nondeterminism—i.e., when backtracking leads to non-accepting paths—and thus improve efficiency of our mining algorithms.

7.4.1 Minimization

Classical FST minimization techniques have been studied for sequential transducers [Mohri (2000)] and do not apply to our FSTs for the reasons mentioned above. However, we can leverage minimization techniques for finite state automaton to minimize our FSTs to the extent possible. Note that, even though such minimization may not provide a deterministic transducer, it reduces nondeterminism in cases when a state has two transitions with the same input and output label. Consider for example input sequence $T = a_1 c$. When we simulate the cFST of Figure 7.7a, we have $\delta(q_0, a_1) = \{(a_1, q_1), (a_1, q_2)\}$ so that cFST simulation tries both options via backtracking. Figure 7.7e show the corresponding minimized cFST for which $\delta(q_0, a_1) = \{(a_1, q_1)\}$ and thus simulation avoids any backtracking. Moreover, as mentioned in Section 7.2.1, minimization helps to reduce the number of transitions and states, which support more efficient mining.

FSTs are isomorphic to NFAs if we treat input and output labels on each transition as one label. We can therefore apply any minimization algorithm for finite state automaton to minimize our FSTs. To minimize cFSTs, we use the algorithm by Brzozowski (1962), which can be applied to any NFA. The algorithm when applied to cFSTs is illustrated in Figure 7.7. We start with the cFST $\mathcal{A}$ (Figure 7.7a; obtained after translating a pattern expression) and construct a *reverse cFST* $R(\mathcal{A})$ (Figure 7.7b) by (i) reversing the direction of the transitions of $\mathcal{A}$, (ii) making the initial state as the final state, and (iii) making the final state(s) as the initial state(s). We then obtain the cFST $D(R(\mathcal{A}))$ (Figure 7.7c) by applying the powerset construction algorithm for converting NFA to DFA on $R(\mathcal{A})$ (note that here we treat cFST as an NFA by considering input and output labels as one single label.) We then repeat the process one more time (Figures 7.7d and 7.7e) to obtain the minimal cFST (Figure 7.7e). Even though, Brzozowski's algorithm runs in exponential time, Almeida et al. (2007) observed that it is the fastest one for most practical NFAs.

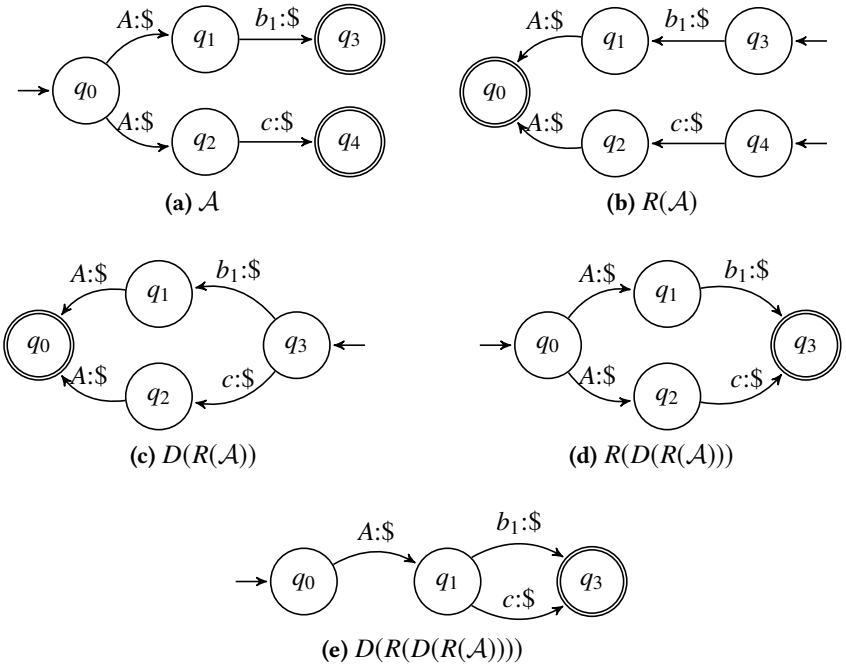


Figure 7.7: *Minimizing cFST for expression* $(Ab_1|Ac)$ *via Brzozowski' algorithm.*

7.4.2 Pruning Irrelevant Input Sequences

An input sequence T is $\mathcal{A}$-*relevant* for a cFST $\mathcal{A}$ 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, consider the hierarchy of Figure 7.5 and the cFST $\mathcal{A}_{F_{7.6}}$. Then the input sequence $T = ca_1b_2e$ is $\mathcal{A}$-relevant where as $T = ca_1b_2a_2$ is $\mathcal{A}$-irrelevant. Note that $G_{\mathcal{A}}(T) = \emptyset$ does not necessarily imply that T is $\mathcal{A}$-irrelevant in our definition.[d]

cFST 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 DESQ mining algorithms. For example, for the input sequence $T = ca_1b_2a_2$, we can completely avoid computing outputs on non-accepting paths $q_0 \xrightarrow{c:\epsilon} q_1 \xrightarrow{A:\$} q_2 \xrightarrow{B:\$\text{-}B} q_4$ and $q_0 \xrightarrow{c:\epsilon} q_1 \xrightarrow{A:\$} q_2 \xrightarrow{B':\$} q_5$. However, we require an efficient way to determine if an input sequence is $\mathcal{A}$-irrelevant, without cFST simulation of course.

The key idea to efficiently determine if a input sequence T is $\mathcal{A}$-relevant is to use a DFA that accepts the same language as an FST. Recall that FSTs are similar to finite state automaton but transitions are additionally labeled with outputs. Therefore, if we ignore the output labels of an FST, we obtain an (non deterministic) finite state automaton (NFA), from which we can construct an equivalent DFA. Thus, if the DFA obtained from an FST accepts an input sequence, so will the FST and vice-versa.

[d] This can happen when $\mathcal{A}$ has an accepting run with ϵ output.

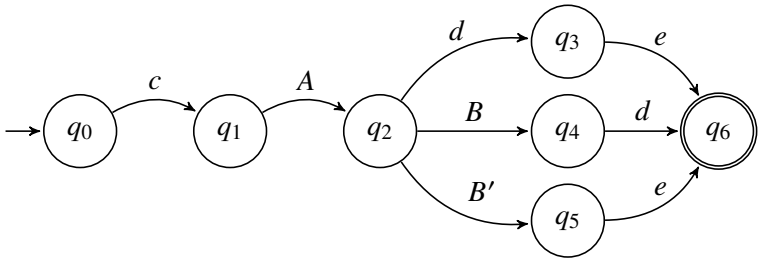
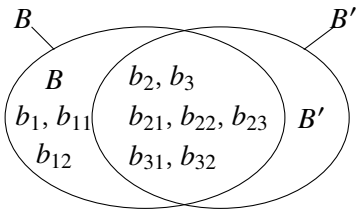


Figure 7.8: *Compressed NFA obtained from cFST $\mathcal{A}_{F_{7.6}}$.*

Thus, using a such a DFA, we can determine in linear time if an input sequence is relevant or not.

Obtaining a DFA from a cFST is however not trivial. Our cFSTs directly exploit the knowledge about the hierarchy to concisely describe the transitions. Thus, ignoring output labels of a cFST results in a "compressed" NFA (cNFA). Figure 7.8 shows cNFA obtained from cFST $\mathcal{A}_{F_{7.6}}$. cNFAs obtained from cFSTs are different than classical NFAs obtained from FSTs. In classical NFAs, two transitions leaving a state either match same items or match disjoint items, where as in cNFAs, two transitions leaving a state can either match same items, or match disjoint items, or match overlapping items, or match items subsumed by other. For example, consider the cNFA shown in Figure 7.8 and the transitions $q_2 \xrightarrow{B} q_4$ and $q_2 \xrightarrow{B'} q_5$. The following Venn diagram illustrates items that these transitions can match.



In this example, both transitions match some overlapping items (e.g., b_2, b_3) as well as match some disjoint items(e.g., input label B can match b_1, which B' can not). Therefore, we cannot readily apply the classical NFA-to-DFA conversion [Rabin and Scott (1959)] process to cNFAs.

One approach to obtain DFA from cNFA is to first "uncompress" the cNFA and then apply the NFA-to-DFA conversion. For example, we obtain from the cNFA of Figure 7.8 the NFA shown in Figure 7.9 and then the resulting DFA shown in Figure 7.10. Observe that the DFA will reject the input sequence $T = ca_1b_2a_2$ and thus we can safely prune T.

DFAs obtained from cFSTs following the above approach can be very large because pattern expressions translate to very large FSTs and can incur substantial memory overhead. For this reason, we make use of a *compressed DFA* (cDFA), in which we group outgoing transitions that go the same state into one transition. For

example, we obtain the cDFA shown in Figure 7.11 after such a grouping. To concisely describe transitions of the cDFA, we make use of labels that encode set of items. For example, in the cDFA shown in Figure 7.11, label I_2^1 encodes items a_1, a_2, and A. Similarly, label I_4^2 encodes items b_{11}, b_{12}, b_1, and B.

In practice, we construct the cDFA directly from a cFST by adapting the power set construction algorithm Rabin and Scott (1959) for NFA-to-DFA conversion to cFSTs and perform on-the-fly grouping of DFA transitions. As in the powerset construction algorithm, each cDFA state corresponds to a set of cFST states. To determine cDFA transitions, we determine for each set of cFST states, which cFST states are "reachable". In more detail, let $\mathcal{A} = (Q, q_S, Q_F, \Sigma, \Delta)$ be the cFST to be converted and let $\mathcal{A}^d = (Q^d, q_S^d, Q_F^d, \Sigma, \Delta^d)$ the resulting cDFA. We construct $\mathcal{A}^d$ on a state-by-state basis; the conversion process is given as Algorithm 7.4. The algorithm starts with the initial state $q_s^d = \{ q_s \}$ (line 1) and maintains the set of unprocessed states (variable Z). Suppose that the algorithm processes state $q^d \in Z$ (line 5); by construction, $q^d \subseteq Q$. Denote by

$$\Sigma_{q^d} = \left\{ w \mid (q_{from}, in, out, q_{to}) \in \Delta, in \text{ matches } w, q_{from} \in q^d \right\}$$

the set of items in Σ that can be matched by outgoing transition from states in q^d. For example, in the cFST $\mathcal{A}_{F7.6}$, we have $\Sigma_{\{q_1\}} = \{ a_1, a_2, A \}$. Let

$$\delta_q(q, w) = \{ q_{to} \mid (q, in, out, q_{to}) \in \Delta, in \text{ matches } w \} . \tag{7.5}$$

denote the set of states from $\mathcal{A}$ that can be reached from the state q via a transition with an input label that matches w. In our running example, $\delta_q(q_2, b_2) = \{ q_4, q_5 \}$. Using δ_q, we compute the reachable states Q_w for each item in $w \in \Sigma_{q^d}$ and group items that reach the same set of reachable states (lines 7–10). For each set of reachable states, we create a cDFA state for this set (if not already created) and add the corresponding encoded transition I_{to}^{from} (lines 11–17). After all new transitions have been added, we mark q^d as processed and add it to set of final states if necessary (lines 18–21).

A transcript of this conversion for cFST $\mathcal{A}_{F_{7.6}}$ is illustrated in Table 7.12, the corresponding cDFA is shown in Figure 7.11. Here, names of cDFA states indicate the corresponding cFST states (e.g., $q_{45}^d = \{ q_4, q_5 \}$). For efficient cDFA simulation we encode cDFA transition labels compactly using bit vectors. For example, an input item w matches a cDFA transition I_{to}^{from} if the bit $\mathrm{id}(w) = 1$, where $\mathrm{id}(w)$ is an integer identifier for item w (see Section 7.5).

Pruning $\mathcal{A}$-irrelevant sequences is beneficial if the input sequence database consist high number of such sequences. In the worst case if all input sequences are $\mathcal{A}$-relevant, this method leads to an additional overhead of simulating the cDFA. Moreover, even if an input sequence is $\mathcal{A}$-relevant, cFST simulation may still involve backtracking that leads to computing outputs on non-accepting paths. For example, simulating the cFST $\mathcal{A}_{F_{7.6}}$ on $\mathcal{A}$-relevant input sequence $T = ca_1b_{21}e$ has

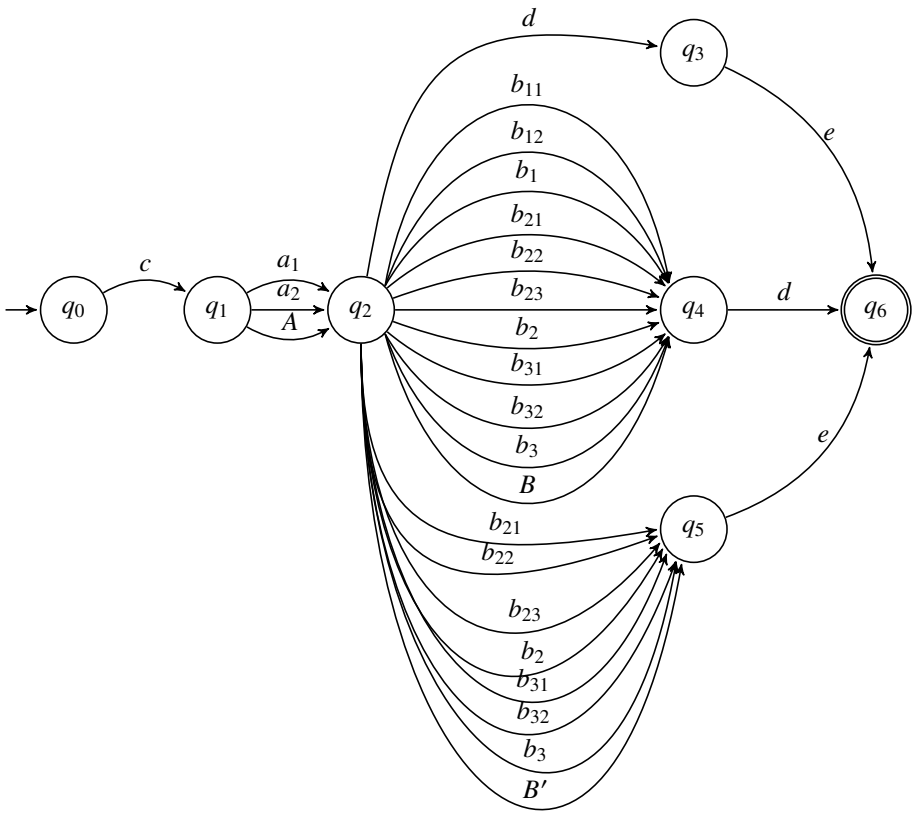


Figure 7.9: *NFA.*

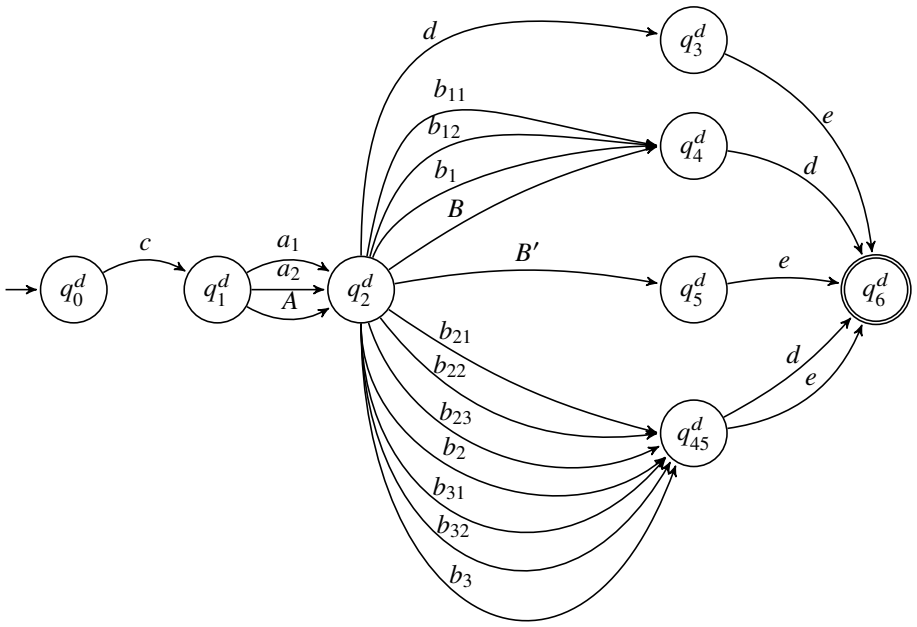


Figure 7.10: *DFA.*

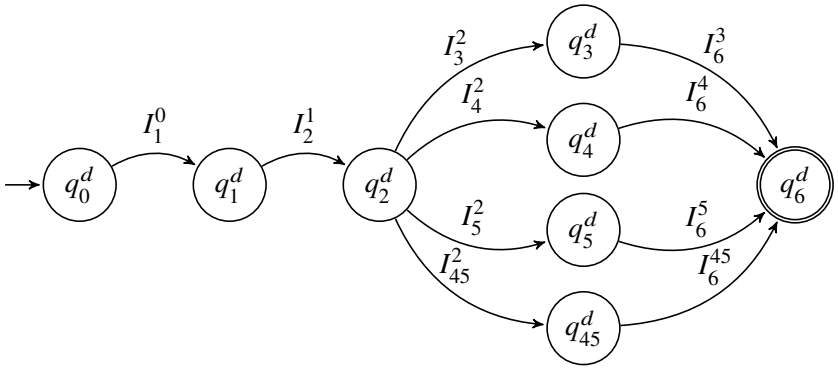


Figure 7.11: *Compressed DFA for* $c\,(A)\,\big[(d)e \mid (B^{\uparrow}d) \mid (B'e)\big]$.

q^d	M	I_{to}^{from}	Transitions
q_0^d	$\{\{q_1\}:\{c\}\}$	$\{c\}$	$q_0^d \xrightarrow{I_1^0} q_1^d$
q_1^d	$\{\{q_2\}:\{a_1, a_2, A\}\}$	$\{A, a_1, a_2\}$	$q_1^d \xrightarrow{I_2^1} q_2^d$
q_2^d	$\{\{q_3\}:\{d\},$	$\{d\}$	$q_2^d \xrightarrow{I_3^2} q_3^d$
	$\{q_4\}:\{B, b_1, b_{11}, b_{12}\},$	$\{B, b_1, b_{11}, b_{12}\}$	$q_2^d \xrightarrow{I_4^2} q_4^d$
	$\{q_5\}:\{B'\},$	$\{B'\}$	$q_2^d \xrightarrow{I_5^2} q_5^d$
	$\{q_4, q_5\}:\{b_2, b_3, b_{21}, b_{22},$ $b_{23}, b_{31}, b_{32}\}\}$	$\{b_2, b_3, b_{21}, b_{22},$ $b_{23}, b_{31}, b_{32}\}\}$	$q_2^d \xrightarrow{I_{45}^2} q_{45}^d$
q_3^d	$\{\{q_6\}:\{e\},$	$\{e\}$	$q_3^d \xrightarrow{I_6^3} q_6^d$
q_4^d	$\{\{q_6\}:\{d\},$	$\{d\}$	$q_4^d \xrightarrow{I_6^4} q_6^d$
q_5^d	$\{\{q_6\}:\{e\},$	$\{e\}$	$q_5^d \xrightarrow{I_6^5} q_6^d$
q_{45}^d	$\{\{q_6\}:\{d, e\},$	$\{d, e\}$	$q_{45}^d \xrightarrow{I_6^{45}} q_6^d$
q_6^d	$\emptyset$	-	-

Figure 7.12: *Transcript of conversion for cFST of Figure 7.6 to cDFA of Figure 7.11*

Algorithm 7.4 Convert a cFST to a cDFA

Require: cFST $\mathcal{A} = (Q, q_S, Q_F, \Sigma, \Delta)$
Ensure: cDFA $\mathcal{A}^d = (Q^d, q_S^d, Q_F^d, \Sigma, \Delta^d)$

1: $q_s^d \leftarrow \{ q_s \}$ // *starting state*
2: $Z \leftarrow \left\{ q_s^d \right\}$ // *unprocessed cDFA states*
3: $Q^d \leftarrow Q_F^d \leftarrow \Delta^d \leftarrow \emptyset$
4: **while** $Z \neq \emptyset$ **do** // *process a cDFA state*
5: $q_{from}^d \leftarrow$ pick any state from Z
6: $M \leftarrow \emptyset$ // *a map from cDFA state to set of items (encoded transition)*
7: **for all** $w \in \Sigma_{q_{from}^d}$ **do**
8: $Q_w \leftarrow \bigcup_{q \in q_{from}^d} \delta_q(q, w)$ // *Reachable states for item w, initially empty*
9: $M[Q_w] \leftarrow M[Q_w] \cup \{ w \}$
10: **end for**
11: **for all** $q_{to}^d \in \text{KEYS}(M)$ **do** // *add transitions*
12: $I_{to}^{from} \leftarrow M[q_{to}^d]$
13: **if** $q_{to}^d \notin Q^d$ **then**
14: $Z \leftarrow Z \cup \left\{ q_{to}^d \right\}$ // *new state?*
15: **end if**
16: $\Delta^d \leftarrow \Delta^d \cup \left\{ (q_{from}^d, I_{to}^{from}, q_{to}^d) \right\}$
17: **end for**
18: $Z \leftarrow Z \setminus \left\{ q_{from}^d \right\}$ // *Mark q_{from}^d as processed*
19: $Q^d \leftarrow Q^d \cup \left\{ q_{from}^d \right\}$
20: **if** $q_{from}^d \cap Q_F \neq \emptyset$ **then** // *final state?*
21: $Q_F^d \leftarrow Q_F^d \cup \left\{ q_{from}^d \right\}$
22: **end if**
23: **end while**

an overhead over computing partial outputs a_1b_{21}, a_1b_2, and a_1B as a result of selecting transition $q_2 \xrightarrow{B:\$-B} q_4$, which leads to a non-final state. In the next section, we propose a technique that completely avoids selecting such transitions and thus removes any unnecessary nondeterminism.

7.4.3 Two-pass Simulation

In the two-pass simulation approach we only generate output(s) only for *relevant transitions*, i.e., transitions that eventually lead to a final state. For example, for the cFST $\mathcal{A}_{F_{7.13}}$ shown in Figure 7.13 and the input sequence $T = ca_1b_{21}e$, transition $q_2 \xrightarrow{B':\$} q_5$ is relevant where as the transition $q_2 \xrightarrow{B:\$-B} q_3$ is irrelevant. As another example, for input sequence $T = ca_1b_{21}d$, transition $q_2 \xrightarrow{B:\$-B} q_5$ is relevant where as transition $q_2 \xrightarrow{B':\$} q_5$ is irrelevant. More formally, for a given cFST $\mathcal{A}$, we say that a transition $(q_{from}, in, out, q_{to}) \in \Delta$ is T-relevant if there exists an accepting run containing the transition. Otherwise we say that the transition is T-irrelevant. Avoiding T-irrelevant transitions during cFST simulation can significantly improve overall efficiency of DESQ mining algorithms as the simulation will not generate any unnecessary partial outputs.

The idea behind the two-pass approach is to pre-compute (before cFST simulation), the set of states from which the final state can be reached. And thus, during cFST simulation, we only consider transitions $(q_{from}, in, out, q_{to})$ where a final state can be reached from q_{to}. The two-pass simulation is given as Algorithm 7.5. We first make a "forward pass" (lines 6–10), in which read the input sequence $T = t_1t_2 \ldots t_{|T|}$ from left to right and incrementally compute the set Q_{pos} of states reached after consuming partial input $t_1 \ldots t_{pos}$ for $1 \leq pos \leq |T|$. Here we use the transition function of Equation 7.5 that ignores output labels. Figure 7.14a illustrates the forward pass on input sequence $T = ca_1b_{12}e$ for cFST $\mathcal{A}_{F_{7.6}}$. For example, we reach states q_4 and q_5 after consuming ca_1b_{12}.

Observe that for any state $q \in Q_{pos}$, there exist a path from the starting state q_S to q for the sequence $t_1 \ldots t_{pos}$. Intuitively, if we reverse the direction of the transitions in $\mathcal{A}$ then there will be a path from a state $q \in Q_{pos}$ to the initial state q_S for the sequence $t_{pos}t_{pos-1} \ldots t_1$. Once we compute the sets Q_{pos} for all input items and if $Q_{|T|} \cap Q_F \neq \emptyset$, then we know that T is $\mathcal{A}$-relevant and we generate the output sequences by simulating the cFST in "reverse". In more detail, denote by $\mathcal{A}'$, the cFST obtained by reversing the direction of transitions of $\mathcal{A}$ and by $\delta'(q, w)$, it's transition function. We make a "backward pass" where we simulate $\mathcal{A}'$ on $t_1 \ldots t_{|T|}$ as follows. For each final state $q \in Q_{|T|} \cap Q_F$, we invoke the Step function with q, $t_{|T|}$, and ϵ (lines 13–15) and repeatedly select transitions that (i) are consistent with the previous input item t_{rpos} and (ii) have q_{to} that are reachable from the initial state (line 21). As we move from state to state, we *prepend* items to the output buffer (lines 22–33). After we consume all items (i.e., $rpos < 1$), we output the buffer, which

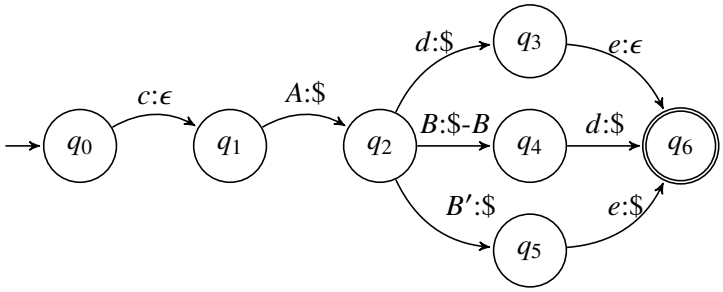
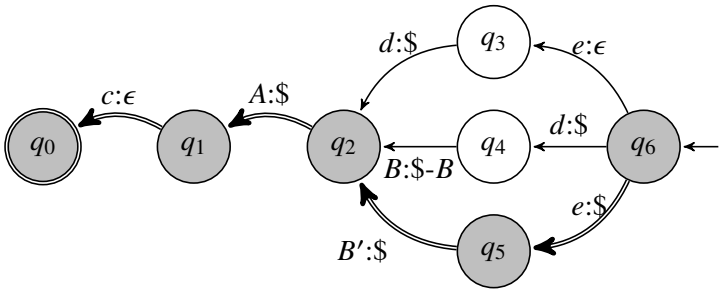


Figure 7.13: *cFST for pattern expression* $c\,(A)\,\big[(d)e \mid (B^{\uparrow}d) \mid (B'e)\big]$.

pos	0	1	2	3	4
t_{pos}	–	c	a_1	b_{12}	e
Q_{pos}	$\{ q_0 \}$	$\{ q_1 \}$	$\{ q_2 \}$	$\{ q_4, q_5 \}$	$\{ q_6 \}$

(a) Forward pass



(b) Backward pass

Figure 7.14: *Illustration of two-pass method for* $T = ca_1b_{12}e$.

contains the generated sequence.

The backward pass is illustrated in Figure 7.14b on our running example. We start with $q = q_6$, $t_4 = e$, and $S = \epsilon$. In the first invocation of the Step, we have $\delta'(q_6, e) = \{ (\epsilon, q_3), (\$, q_5) \}$. Since $q_3 \notin Q_3$, we know that we can not reach the final state (initial state of $\mathcal{A}$) via this transition therefore we only consider $(\$, q_5)$ and generate $S = e$. The simulation then consumes items b_{21}, a_1, and c and generates the output $S = a_1 b_{21} e$ as desired. No other unnecessary partial outputs are generated.

7.4.4 Integrating pruning input sequences and two-pass into mining

We now discuss how to integrate pruning $\mathcal{A}$-irrelevant input sequences and two-pass simulation approach into our mining algorithms DESQ-COUNT and DESQ-DFS. We denote by $\mathcal{A}^d$ the cDFA and by $\mathcal{A}'$ the reverse cFST corresponding to cFST $\mathcal{A}$.

To integrate pruning irrelevant input sequences in DESQ-COUNT, we simulate $\mathcal{A}$ on input sequences $T \in \mathscr{D}$ that are accepted by $\mathcal{A}^d$. We then proceed as in DESQ-COUNT. To integrate two-pass simulation in DESQ-COUNT, we compute the set $G^F_{\mathcal{A}}(T)$ by adapting Algorithm 7.5's backward pass to work with the f-list, i.e., we stop exploring a path as soon as an infrequent item is produced in lines 26, 28, and 31.

We integrate pruning irrelevant input sequences in DESQ-DFS by only consider snapshots $T[1@q_S]$ for ϵ (Algorithm 7.3;line 2) for which T is accepted by $\mathcal{A}^d$. The initial projected database for ϵ is the only difference to DESQ-DFS, i.e., we now proceed with expansions as in DESQ-DFS unmodified.

Integrating two-pass simulation approach in DESQ-DFS is slightly more involved. Recall that in two-pass approach, we compute the sets $Q_1, Q_2, \ldots, Q_{|T|}$ of reachable states for each input item in T in the forward pass. Since DESQ-DFS incrementally simulates $\mathcal{A}$ on all input sequences, we need to store sets of reachable states for each input sequence. Denote by $T.Q_{pos}$ the set of reachable states for input sequence T at position pos. Two-pass approach with DESQ-DFS is illustrated in Algorithm 7.6.

Initially, while scanning the input database, we make the forward pass for each input sequence $T \in \mathscr{D}$ and compute the sets $T.Q_0, T.Q_1, T.Q_2, \ldots, T.Q_{|T|}$ (lines 3–9). If the reachable states $T.Q_{|T|}$ does not contain a final state after reading the input, we discard the input sequence other wise for each final state $q \in Q_{|T|}$ we add the snapshot $T[|T|@q]$ to the projected database of ϵ (lines 6–8). We then perform expansions almost as described in Algorithm 7.3. The key difference is that we use IncStepBack (lines 12–29), in which incrementally simulate we simulate $\mathcal{A}'$ using the transition function $\delta'(q, t_{pos})$ and consider output label-reachable state pairs (out, q_{to}) for which $q_{to} \in T.Q_{pos-1}$ (line 16). If we produced an item w and reached

Algorithm 7.5 Two-pass simulation

Require: cFST $\mathcal{A} = (Q, q_S, Q_F, \Sigma, \Delta), T = t_1 \dots t_{|T|}$
Ensure: $G_{\mathcal{A}}(T)$

$G_{\mathcal{A}}(T) \leftarrow \emptyset$
$\mathcal{A}' \leftarrow$ Reverse($\mathcal{A}$) // *Reverse cFST*
$Q_0 \leftarrow \{ q_S \}$

//forward pass, compute reachable states for each input item
for $pos \leftarrow 1$ **to** $|T|$ **do**
  for all $q \in Q_{pos-1}$ **do**
    $Q_{pos} \leftarrow Q_{pos} \cup \delta_q(q, t_{pos})$
  end for
end for

//Backward pass, simulate $\mathcal{A}'$ *on* $T = t_1 \dots t_{|T|}$ read backwards
for all $q \in Q_{|T|} \cap Q_F$ **do**
  Step($q, t_{|T|}, \epsilon$)
end for

void Step($q, rpos, S$):
if $rpos < 1$ **and** $S \neq \epsilon$ **then**
  $G_{\mathcal{A}}(T) \leftarrow G_{\mathcal{A}}(T) \cup \{ S \}$
end if
for all $(out, q_{to}) \in \delta'(q, t_{rpos})$ **such that** $q_{to} \in Q_{rpos-1}$ **do** // *empty if* $rpos < 1$
  switch (out)
    case ϵ**:**
      Step($q_{to}, rpos - 1, S$)
    case w**:**
      Step($q_{to}, rpos - 1, wS$)
    case \$**:**
      Step($q_{to}, rpos - 1, t_{rpos}S$)
    case \$-x for $x \in \Sigma \cup \{\top\}$**:**
      for all $w' \in \text{anc}(t_{rpos}) \cap \text{desc}(x)$ **do**
        Step($q_{to}, rpos + 1, w'S$)
      end for
  end switch
end for

Algorithm 7.6 Integrating two-pass approach in DESQ-DFS

Require: $\mathscr{D}$, cFST $\mathcal{A} = (Q, q_S, Q_F, \Sigma, \Delta)$, σ, f-list F
Ensure: P-frequent sequences for $\mathcal{A}$ in $\mathscr{D}$

```
1:  A' ← Reverse(A)
2:  S ← ε                          // create root node; initially fields S.Proj = S.Sup = ∅
3:  for T ∈ D do
4:    T.Q_0 ← { q_S }
5:    Make forward to compute T.Q_pos for 1 ≤ pos ≤ |T| as in Algorithm 7.5 (lines 6–10)
6:    for q ∈ Q_F ∩ Q_|T| do
7:      S.Proj ← S.Proj ∪ { T[|T|@q] }
8:    end for
9:  end for
10: Expand(S)          // Perform expansions as in Algorithm 7.3 (lines 5–16) using
                       IncStepBack(T, pos, q, S) instead of IncStep(T, pos, q, S) in line 7
11:
12: void IncStepBack(T, pos, q, S):
13: if pos < 1 and S ≠ ε then
14:   S.Sup ← S.Sup ∪ { T }                                   // initially empty
15: end if
16: for all (out, q_to) ∈ δ'(q, t_pos) such that q_to ∈ T.Q_{pos−1} do
17:   switch (out)
18:     case ε:
19:       IncStepBack(T, pos − 1, q_to, S)
20:     case w:
21:       if f(w, D) ≥ σ then Prepend(S, w, T, pos − 1, q_to)
22:     case $:
23:       if f(t_pos, D) ≥ σ then Prepend(S, t_pos, T, pos − 1, q_to)
24:     case $-x, x ∈ Σ ∪ { ⊤ }:
25:       for all w' ∈ anc(t_pos) ∩ desc(x) do
26:         if f(w', D) ≥ σ then Prepend(S, w', T, pos − 1, q_to)
27:       end for
28:   end switch
29: end for
30:
31: void Prepend(S, w, T, pos, q):
32: S.Children ← S.Children ∪ { wS }                  // node wS is created if new
33: wS.Proj ← wS.Proj ∪ { T[pos@q] }                  // initially empty
```

state q_{to}, we add the snapshot $T[pos-1@q_{to}]$ to the child node wS (lines 21, 23, and 26). If we consumed T reading backwards, we add T to the support set of S (line 14).

7.5 Experiments

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.

Summary of our results

1) Many subsequence constraints can be expressed with pattern expressions.
2) cFSTs sped up pattern matching by multiple orders of magnitude when compared to 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 subsequences per input are generated, DESQ-DFS was more than an order of magnitude faster than DESQ-COUNT and Naïve.
6) DESQ has acceptable overhead over state-of-the-art specialized sequence miners for common subsequence constraints.

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

7.5.1 Experimental Setup

Datasets

Table 7.2 summarizes our datasets. NYT is a subset of The New York Times corpus and contains news articles. We generated an item hierarchy using annotations from the Stanford CoreNLP tools. The NYT hierarchy consists of named entities, which generalize to their type (PERSON, ORGANIZATION, LOCATION, MISC) and then to ENTITY, and of words, which generalize to their lemma and then to their part-of-speech tag. For example, "Maradona"⇒PERSON⇒ENTITY and "is"⇒"be"⇒VERB.

AMZN is a dataset of Amazon product reviews [Web data: Amazon reviews] from which we extracted sequences of products (ordered by review timestamps) for each user. We used the Amazon product hierarchy as our item hierarchy. For example, "Canon 5D" ⇒"Digital Cameras"⇒"Camera & Photo"⇒ "Electronics".

PRT is a dataset of protein sequences obtained from SMA composed of 25 amino acid codes (items). The hierarchy is flat, i.e., there are no generalizations.

Table 7.2: *Dataset statistics*

		NYT	AMZN	PRT
Sequence database	# Sequences	21,590,967	6,643,666	103,120
	Avg. length	19.9	4.5	482
	Max. length	5,042	25,630	600
	Total items	430,279,662	29,667,966	49,729,890
	Distinct items	3,975,859	2,374,096	25
Hierarchy	Total items	4,136,774	2,385,775	103,120
	Leaf items	3,901,118	2,371,522	103,120
	Interm. items	235,633	11,630	0
	Root items	23	2,623	103,120
	Max. depth	3	8	1
	Avg. fan-out	17.5	204	0
	Max. fan-out	1,505,913	332,723	0

Pattern Expressions

We considered pattern expressions that express constraints in information extraction (IE), natural language processing (NLP), and customer behavior mining applications. These expressions are shown in Table 7.3 along with some mining results. Expressions N_1–N_5 express constraints useful for IE and NLP applications and are inspired from Fader et al. (2011); Nakashole et al. (2012); Del Corro et al. (2015); Lin et al. (2012); these expressions were used on the NYT dataset. Expressions A_1–A_4 expresses constraints useful for market-basket analysis and apply to AMZN. Expressions from the third category (P_1–P_4) are used to mine protein sequence motifs from the PRT dataset; the subsequence constraints were taken from the PROSITE database. The fourth category (T_1–T_3) models traditional constraints. We used NYT with these expressions.

Implementation and setup

We implemented DESQ in Java (JDK 1.8). We used ANTLR to generate a parser for pattern expressions. The cFST is constructed from the resulting parse tree, which is subsequently minimized. To measure the overhead of DESQ for common subsequence constraints, we compared it against state-of-the- art methods. For length and gap constraints, we used (1) C++ implementation of cSPADE [Zaki (2000)] from the author, (2) our implementation of SPADE in Java that additionally handles hierarchy constraints, (3) our implementation of prefix-growth [Pei et al. (2002)] in Java. For RE constraints, we used (1) prefix-growth and a C++ executable of SMA [Trasarti et al. (2008)] obtained from the authors. To evaluate cFSTs we compared it against state-of-the-art FST library OpenFST v1.5.0.

		Example patterns from NYT dataset (frequency)
N_1: ENTITY (VERB$^+$ NOUN$^+$? PREP?) ENTITY	*Relational phrase between entities*	lives in (847), is being advised by (15), has coached (10)
N_2: (ENTITY$^\uparrow$ VERB$^+$ NOUN$^+$? PREP? ENTITY$^\uparrow$)	*Typed relational phrases*	ORG headed by ENTITY (275), PER born in LOC (481)
N_3: (ENTITY$^\uparrow$ be$^\uparrow_=$) DET? (ADV? ADJ? NOUN)	*Copular relation for an entity*	PER be novelist (165), LOC be great place (38),
N_4: (.$^\uparrow$){3} NOUN	*Generalized 3-grams before a noun*	NOUN PREP DET (4,223,219), DET ADV ADJ (350,005)
N_5: ([.$^\uparrow$ . .]|[. .$^\uparrow$.]|[. . .$^\uparrow$])	*Generalized 3-grams, where at most one item is generalized*	the ADJ human (1,238), for DET book (1,704)
		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" (556), "MP3 Players", "Headphones" (814)
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" (11), "Batteries", "SD Cards" (12)
A_4: (MInstr$^\uparrow$)[.{0,2}(MInstr$^\uparrow$)]{1,4}	*Generalized sequences of musical instruments*	"Saxophones", "Bags & Cases", "Instr. Accessories" (127)
		Example patterns from PRT dataset (frequency)
P_1: ([S|T]) .*(.) .*([R|K])	*subsequences that match RE≡[S|T] . [R|K]*	S L R (103,093), T A K (102,941), S A K (102,946)
P_2: ([I|V]) .*(D) .*(L) .*(G) .*(T) .*([S|T]) .*(.) .*([S|C])	*subsequences that match RE≡[I|V] D L G T [S|T] .[S|C]*	I D L G T T L S (102,975), V D L G T S T C (92,662) V D L G T S D S (102,901)
P_3: ([S|T] . [R|K])	*contiguous subsequences that match RE≡[S|T] . [R|K]*	S L R (14,995), T A K (8,840), S A K (10,397)
P_4: ([S|T] . . [D|E])	*contiguous subsequences that match RE≡[S|T] . . [D|E]*	S D L E (2,015), T L E E (2,329), S G L D (1,054)
Pattern expression	Description	Example patterns from NYT dataset (frequency)
T_1: (.){1,λ}	*n-grams of up to λ words*	green tea (337), editor in chief (3275)
T_2: (.)[.{0,γ}(.)]{1,$\lambda-1$}	*Skip n-grams with gap at most γ words and of up to length λ*	flight from to (758), son of and of (15896)
T_3: (.$^\uparrow$){1,λ}	*Generalized n-grams of up to λ words*	NOUN PREP DET NOUN (4.2M), PERSON be NOUN (2199)

Table 7.3: *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)*

	N_1	N_2	N_3	N_4	N_5	A_1	A_2	A_3	A_4
OpenFST	1 hr	1 hr	1 hr	6 hr	>12 hr	>12 hr	>12 hr	>12 hr	50 min
cFST	2 min	2 min	3 min	2 hr	>12 hr	1 hr	15 min	30 min	1 min

Table 7.4: *Runtimes of Naïve with cFST and openFST*

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 arrays of item identifiers and use variable-length byte encoding to compress projected databases. 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 i7-4712HQ and 16GB RAM running Windows 10. We used a different setup for the PRT dataset as the SMA implementation is provided as a Windows binary only. All experiments were run single-threaded.

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 (including I/O). All measurements were averaged over three independent runs. Unless stated otherwise, all methods produced the same result.

7.5.2 Results

A. FST Optimizations

We first evaluated the effectiveness of our FST optimizations (compression and minimization). We used the Naïve approach (Algorithm 7.2) on pattern expressions N_1–N_5 and A_1–A_4. We used (1) cFSTs with our simulation algorithm (Algorithm 7.1) and (2) uncompressed FSTs with state-of-the-art library OpenFST. The results are shown in Table 7.4. We observed that Naïve was orders of magnitude faster when used with cFST simulation than when used with OpenFST. This is because pattern expressions often translate to excessively large FSTs, which are inefficient to simulate (see Table 6.1 on page 88 and discussion on cFSTs in Sec. 7.2 on page 104). Moreover, OpenFST cannot directly handle hierarchies and, as discussed in Sec. 7.4 (page 116), and many of our pattern expressions cannot be determinized. We conclude that cFST compression and minimization is effective.

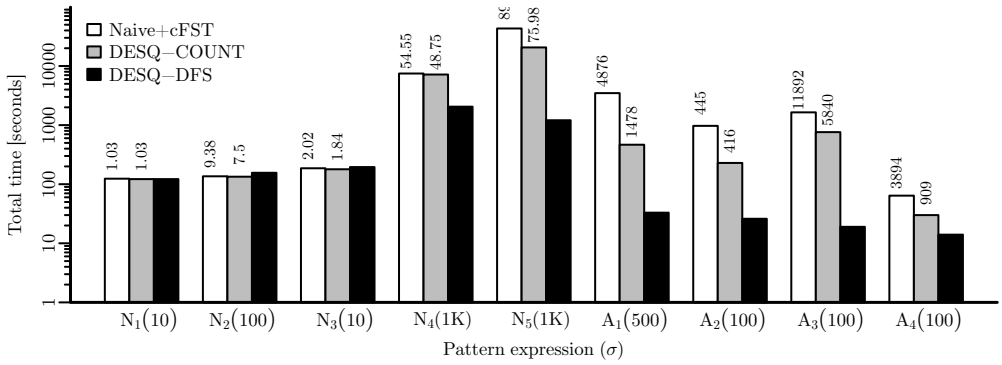


Figure 7.15: *Performance of DESQ mining algorithms. The numbers on top of the bars indicate the average number of P-subsequences per input sequence.*

B. DESQ Mining Algorithms

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 Figure 7.15, which also gives the minimum support threshold σ used for each pattern expression (chosen empirically). The runtimes are given in log-scale.

On the NYT dataset, for expressions N_1–N_3, DESQ-COUNT and DESQ-DFS had similar performance and finished in a few minutes. For N_4–N_5, however, runtimes were higher and DESQ-DFS was significantly faster than DESQ-COUNT (up to 14x). To gain insight into these results, we computed the average number μ of P-sequences (average of $|G_P^F(T)|$).[e] These numbers are shown above the bars for each pattern expression. We observed that for small values of μ, DESQ-COUNT and DESQ-DFS had similar performance, whereas for larger values of μ, DESQ-DFS was much more efficient. When μ is small, the simple counting method of DESQ-COUNT is expected to work well because few sequences are generated. The advanced pruning methods of DESQ-DFS are then not needed. When μ is large, however, 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) DESQ-DFS consistently outperformed DESQ-COUNT (up to 22x). This behavior is explained by the observation that μ was large for all pattern expressions.

Based on these results, we conclude that DESQ-DFS consistently worked well in our experiments. Although DESQ-COUNT was slightly faster in some cases, it blew up on others. Thus we consider it generally safer to use DESQ-DFS in practice.

[e] We averaged over input sequences T for which $G_P^F(T) \neq \emptyset$.

C. Effectiveness of Pruning Irrelevant Sequences and Two-pass

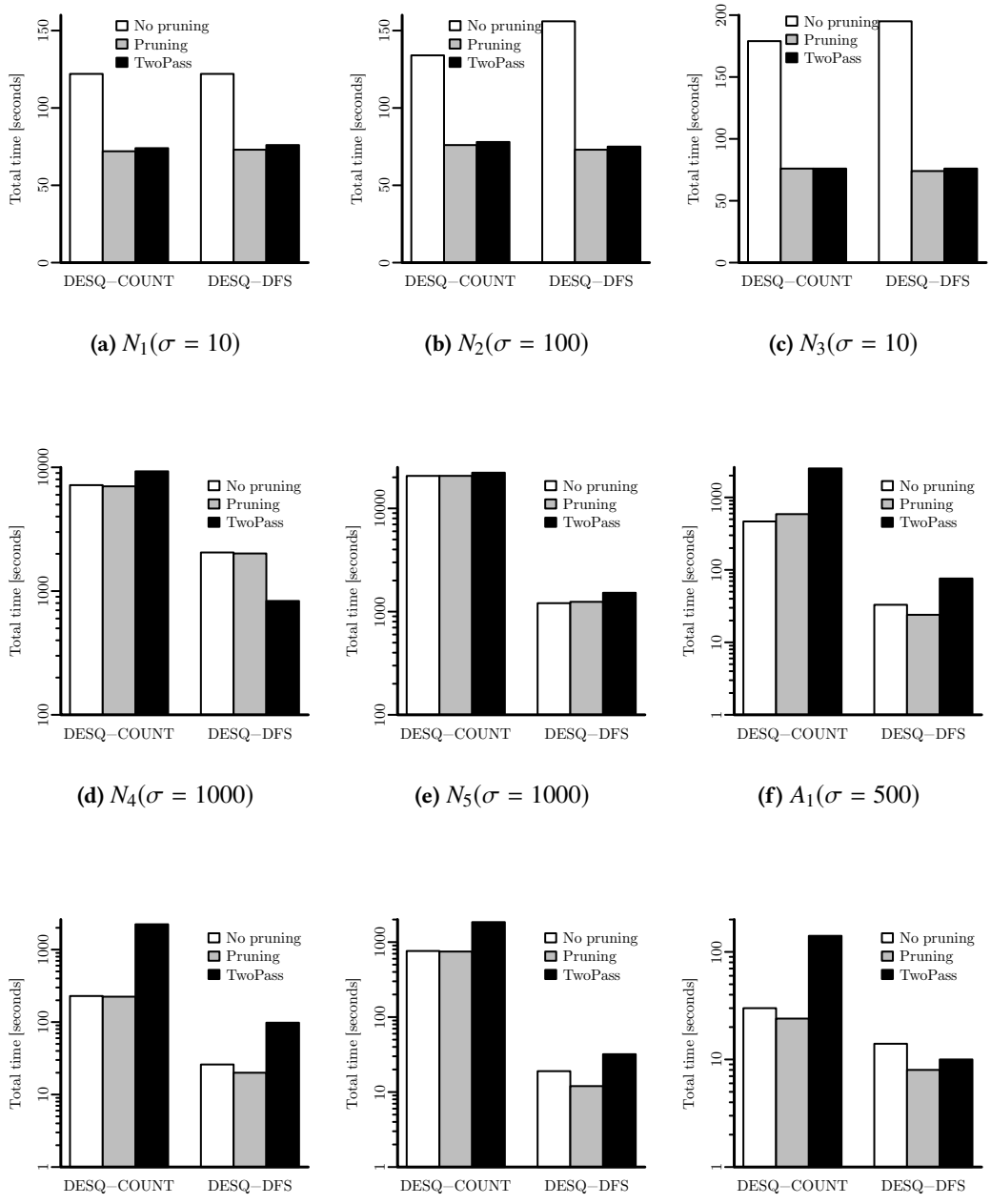
In these set of experiments, we investigated how integrating pruning irrelevant input sequences and two-pass simulation approach effect our DESQ-COUNT and DESQ-DFS mining algorithms. The results are shown in Figure 7.16 for expression N_1–N_5 (a–e) and A_1–A_4 (f–i). Note that the runtimes for (d)–(i) are shown using log scale. In each figure, the first group (first three bars) show the runtimes for DESQ-COUNT and the second group (last three bars) for DESQ-DFS.

We first focus on NYT. For expression N_1–N_3, pruning input sequences and two-pass, both improved runtimes of DESQ-COUNT and DESQ-DFS by more than factor 2. Pruning input sequences with DESQ-COUNT and DESQ-DFS on expressions N_4 and N_5 was not very effective; runtimes improved only by 2% for N_4 and, worsened by 2% for N_5. The two-pass approach with DESQ-COUNT was ineffective for N_4 (two-pass was up to 30% slower), where as with DESQ-DFS it sped up mining by more than factor 2. For expression N_5, DESQ-COUNT with two-pass was slower by 7% and DESQ-DFS with two-pass was slower by 20%.

To gain further insight for these runtimes, we computed cFST simulation statistics for pruning input sequences and two-pass with DESQ-COUNT and DESQ-DFS. In particular, for each pattern expression we computed (1) the percentage of $\mathcal{A}$-relevant input sequences and, (2) number of Steps (IncSteps) executed by DESQ-COUNT (DESQ-DFS). The statistics are shown in Table 7.5.

We observed that for expression N_1–N_3, 97–99% of input sequence were $\mathcal{A}$-irrelevant and thus pruning using cDFA for these expression was very effective. For expression N_4 and N_5, very small fraction of the input sequences were $\mathcal{A}$-irrelevant (less than 10% and 4%, resp.) and thus pruning was not very effective. We also observed that number of Steps executed by DESQ-COUNT with two-pass was factor 1.6× higher for expression N_4, which explains its ineffectiveness. This is because, we create a cFST for $.^*E$ and produce an output whenever a final state is reached whether or not entire input is consumed to be able match pattern expression E anywhere in the input (see discussion on partial matches in Section 7.2.2; page 107). This increases the number of Steps in the backward pass of two-pass when a final state is reached for multiple positions in the input sequence and thus the sequence has to be read multiple times (see lines 13–15; Algorithm 7.5). On expression N_5, this increase in number of steps is even more pronounced (up to 5×) because, the cFST for N_5 has more than one final state. Moreover, every path of the cFST for N_5 is an accepting path. Contrary to DESQ-COUNT with two pass on expression N_4, DESQ-DFS with two-pass was very effective. Less number IncSteps also support this. Note that, for N_4, number of IncSteps is slightly higher for DESQ-COUNT with two-pass, still runtimes are faster. This is because, IncSteps (or Steps) for computing $.^*$ is the beginning do not incur significant cost. DESQ-DFS with two-pass for N_5 was again infective because of the high number of IncSteps.

On AMZN (A_1–A_3) DESQ-COUNT with pruning input sequences was consist-



(a) $N_1(\sigma = 10)$ **(b)** $N_2(\sigma = 100)$ **(c)** $N_3(\sigma = 10)$

(d) $N_4(\sigma = 1000)$ **(e)** $N_5(\sigma = 1000)$ **(f)** $A_1(\sigma = 500)$

(g) $A_2(\sigma = 100)$ **(h)** $A_3(\sigma = 100)$ **(i)** $A_4(\sigma = 100)$

Figure 7.16: *Effectiveness of pruning input sequences and two-pass in DESQ-COUNT and DESQ-DFS for pattern expression N_1–N_5 and A_1–A_4.*

ently slower than DESQ-COUNT even though a large fraction of input sequences were $\mathcal{A}$-irrelevant. The longest sequence in AMZN has $\tilde{2}$5K items but on an average has only 4.5 items (cf. Table 7.2; page 130). Thus pruning in this case only prunes very short sequences, which is not effective. This is also supported by the number of Steps executed by DESQ-COUNT with and with-out pruning, which are very close. For A_4, however, DESQ-COUNT with pruning was slightly effective. For all expression, DESQ-DFS with pruning consistently performed well and improved mining times by up to an order of magnitude. Thus cDFA based pruning combined with support based pruning of DFS seems beneficial. For A_1–A_4, DESQ-COUNT and DESQ-DFS with two-pass was considerably slower (by up to an order of magnitude). This is because, all these expression have multiple final states, which results in multiple passes over each input sequence. This in turn increases the number of Steps and IncSteps executed as seen in Table 7.5.

Our results indicate that effectiveness of pruning input sequences and two-pass depends on the pattern expression and also on the input data. Pruning consistently worked well for NYT where as it was not effective on AMZN. Two-pass worked well only for some expressions. An interesting direction for future work will be to investigate how we can determine the best method for a given pattern expression and the input data. One approach could to be to first compute a random sample of the input data and determine the cost (in terms of runtime or number of Steps/IncSteps) and then use the best performing method on the entire input data.

D. DESQ for RE constraints

In this set of experiments, we evaluated the efficiency of DESQ for mining frequent subsequences (all or contiguous) that match a RE. Our pattern expressions allows us to express REs with their equivalent pattern expressions (cf. Table 6.2 on page 90 and expressions P_1–P_4 of Table 7.3 on page 131). We compared DESQ's performance against state-of-the-art RE-constraint FSM methods SMA and prefix-growth. We used the PRT dataset; the runtimes are shown in log-scale in Figure 7.17a. We observed that DESQ was up to 2.5x slower than SMA for P_1 and up to 1.3x slower than SMA on P_2. We do not give SMA results for P_3 and P_4 because the implementation produced incorrect results (acknowledged by the original authors). We did not investigate this further as the SMA source is not available. DESQ was roughly on par with prefix-growth for P_1–P_4 (up to 1.3x) slower. The overhead of DESQ thus appears acceptable.

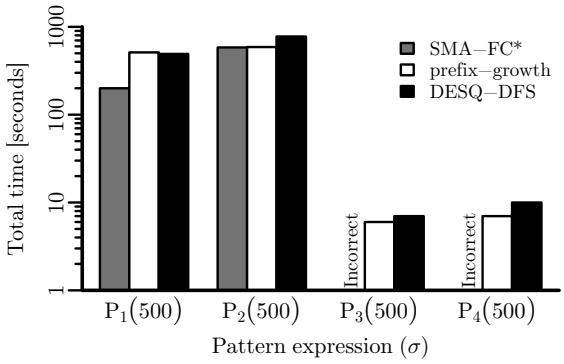
E. DESQ for traditional subsequence constraints

Lastly, we investigated the overhead of DESQ compared to specialized miners for traditional subsequence constraints.

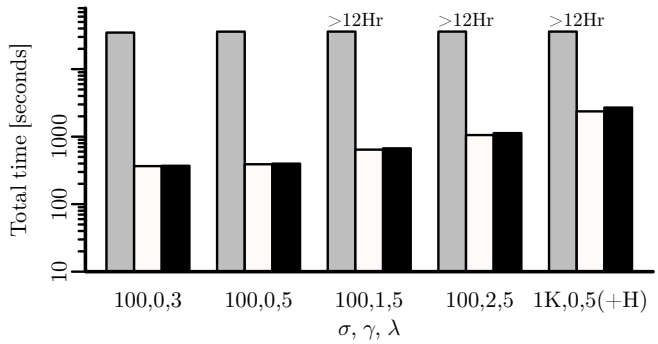
We considered length and gap constraints as well as item hierarchies. We map these constraints to pattern expressions and obtain T_1–T_3 of Table 7.3 (page 131). The

Pattern Expr.	$\mathcal{A}$-relevant inputs (%)	DESQ-COUNT (#Steps) No pruning	Pruning	TwoPass	DESQ-DFS (#IncSteps) No pruning	Pruning	TwoPass
N_1	2.07	463,911,064	14,220,102	5,626,302	463,025,362	13,948,868	4,592,201
N_2	2.07	522,977,157	21,369,711	31,406,977	511,811,293	17,270,738	10,336,244
N_3	0.93	479,264,155	6,612,820	3,368,665	478,319,569	5,979,150	1,234,596
N_4	89.31	7,978,342,975	7,847,720,831	13,019,897,877	5,523,358,624	5,412,851,875	6,560,928,573
N_5	96.52	5,715,941,254	5,713,681,132	30,877,300,511	3,311,332,824	3,308,776,942	8,439,419,085
A_1	2.41	1,272,989,661	1,243,872,771	34,227,210,177	87,725,169	58,473,759	807,248,058
A_2	15.67	386,902,636	371,547,460	38,576,697,219	60,365,344	44,729,288	1,142,122,508
A_3	0.42	518,814,454	489,941,094	20,777,145,559	38,392,935	9,519,575	276,843,663
A_4	0.11	114,060,273	84,385,840	2,333,778,277	31,457,472	1,768,815	18,816,336

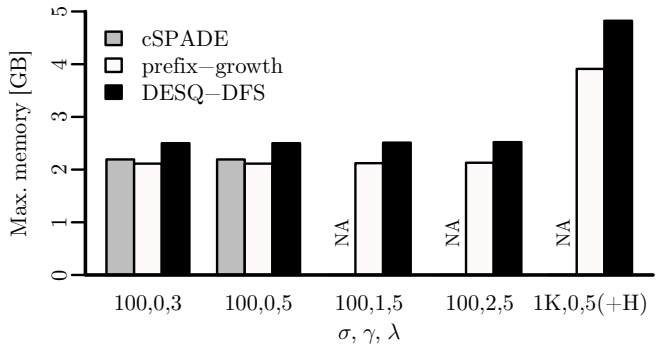
Table 7.5: *cFST simulation statistics for DESQ-COUNT and DESQ-DFS*



(a) RE-constraints



(b) NYT (length, gap, hierarchy)-constraints



(c) NYT (length, gap, hierarchy)-constraints

Figure 7.17: *Overhead of DESQ for RE constraints and traditional subsequence constraints.*

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 7.17b using log-scale. For n-grams (first two groups), we observed that DESQ-DFS was up two orders of magnitude faster than cSPADE. We only show the result for our own cSPADE implementation; the original C++ implementation was significantly slower. For example, for mining 10% of NYT, the original cSPADE implementation took more than 3 hours whereas our implementation took 400 seconds. Both cSPADE implementations were significantly slower than prefix-growth and DESQ-DFS, however, because cSPADE follows a candidate-generation-and-test approach and suffers from an excessive number of generated candidates. To keep our study manageable, we stopped cSAPDE after 12 hours. Compared to prefix-growth, DESQ-DFS had negligible overhead (less than 2.5%). For gap constraints (third and fourth group), DESQ-DFS was competitive and had an overhead of less than 10% over prefix- growth. This overhead is expected as pattern expressions for gap constrains have uncaptured wildcards (cf. T_2 in Table 7.3), which increases nondeterminism in the corresponding cFSTs and thus leads to more snapshots. For generalized n-grams (last group), where we additionally considered item hierarchies, the overhead was slightly more pronounced (up to 13%). Here the amount of backtracking performed by DESQ increased with the depth of hierarchy (cf. line 31 of Algorithm 7.3 and discussion in Section 7.2.2).

We also investigated the overhead in terms of memory consumption. The results are shown in Figure 7.17c. For cSPADE, we report the maximum size of the inverted index and for prefix-growth and DESQ-DFS, we report the maximum size of the projected database. For n-grams and gap-constraints, DESQ-DFS had an overhead of up 18% and for generalized n-grams up to 23%. The overhead is unavoidable as for DESQ-DFS, we need to store cFST snapshots compared to only positional information as in prefix-growth and cSPADE. We may, however, improve memory consumption by swapping projected databases to disk [Pei et al. (2001)].

7.6 Related Work

Several algorithms and methods for integrating subsequence constraints into mining have been studied in literature. While, most focus on traditional subsequence constraints (e.g., those discussed Chapters 3–5), very few available methods support general subsequence constraints, albeit with limitations.

Most of the prior work on supporting general constraint has been confined to regular expressions. Garofalakis et al. (1999) proposed the SPIRIT family of algorithms, in which they introduced regular expressions (RE) constraints that subsequences need to satisfy. It translates the provided RE into a DFA and adapts a GSP-like algorithm to mine frequent sequential patterns. Antunes and Oliveira (2002)

adapted ideas from SPIRIT to pushdown automaton to deal with context-free grammars. Albert-Lorincz and Boulicaut (2003) proposed RE-Hackle algorithm, which represents the RE via its abstract syntax tree and uses bottom-up approach combined with frequency based pruning to evaluate sequences. Pei et al. (2002) advocated the prefix-growth method as an extension to PrefixSpan to handle RE constraints. RE constraints have also been studied by Trasarti et al. (2008). They proposed the SMA algorithm, which uses Petri nets to match an RE. These methods do not support capture groups, which are key to express many traditional constraints in a unified way (see Table 6.2; page 90). Antunes and Oliveira (2004) also proposed ϵ-*accepts*, based on pattern-growth approach to mine sequences that approximately match a given RE. To find approximate sequences, they run input sequences through a DFA and either replace, or delete, or insert an item when every their is no matching transition in the corresponding DFA. More recently, Jean-Philippe Metivier and Charnois (2013) and Negrevergne and Guns (2015) described how constraint programming can also be used to mine sequences that satisfy a given constraint; their approaches however focus on supporting traditional constraints and RE constraints.

Some of the subsequence (e.g., gap constraints) target the input sequence, whereas others (e.g., length constraints, RE constraints) target the subsequence. Our pattern language unifies both targets and allows us to express all of the subsequence constraints discussed before (see Table 6.2 for some examples). In addition, it allows us to describe the context in which subsequences should be considered relevant (e.g., subsequences that appear between certain items in the input sequence), incorporates item hierarchies, uses a more powerful computational model based on finite state transducers.

Our work is also related to pattern matching. There are many languages and systems for pattern matching over sequences. For example, SystemT's AQL language [Krishnamurthy et al. (2009)] provides a SQL-like syntax to specify and extract pattern matches from text documents. Languages based on cascaded grammars such as CPSL [Appelt and Onyshkevych (1998)] are also used in many information extraction engines. Christ (1994) proposed a Corpus Query Language based on regular expressions for searching pattern matches in text corpora. Pattern matching is also crucial for complex event processing tasks [Dindar et al. (2009); Demers et al. (2007)], which aim to detect pattern matches in (live or archived) event sequences. Our pattern expressions are simpler than most pattern matching languages, yet expressive enough to specify many subsequence constraints that arise in 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 matching languages.

Finite state transducers [Mohri (1997); Mohri et al. (2002)] 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 expressions. In contrast to existing work on FSTs, our FSTs are often neither sequential nor p-subsequential (see discussion in Section 7.4;page 116) so that many existing optimization methods do not apply (e.g., minimization, determinization). We provide methods to extend, compress, and optimize our special FSTs in order to effectively handle pattern mining tasks and large hierarchies. Although traditional FST libraries such as OpenFST [Allauzen et al. (2007)] can also be used within DESQ, our experimental study suggests that compressed FSTs support more efficient mining.

7.7 Summary

In this chapter, we proposed compressed FST and a simulation algorithm that effectively handles large hierarchies for generating sequences produced by our pattern expressions. We subsequently, proposed two efficient mining algorithms DESQ-COUNT and DESQ-DFS based on simulation. While DESQ-COUNT fares well for pattern expressions that are selective, DESQ-DFS can handle more demanding pattern expressions. We also studied how to minimize our specialized FSTs and proposed novel techniques to reduce nondeterminism based on pruning input sequences using DFA and a two-pass simulation approach. Our experimental study indicates that DESQ is an efficient, general purpose FSM framework for traditional as well as customized subsequence constraints.

Part III

Wrapping Up

Chapter 8

Conclusions

Summary

In this thesis, we presented scalable and general-purpose methods for frequent sequence mining for traditional as well as customized notions of subsequence constraints that arise in applications.

We extended the MG-FSM framework, which provides a distributed framework for mining very large collection of sequences. We proposed novel algorithms that improve and extend this basic framework to support many traditional subsequence constraints. In particular, we proposed a special-purpose pivot sequence miner, which led to an higher overall efficiency and proposed methods that extend MG-FSM to support long input sequences, temporal gap constraints, and mining of only maximal and closed sequences. We also proposed the LASH algorithm that efficiently incorporates item hierarchies into MG-FSM's partitioning framework to mine hierarchical patterns. In our experimental study, we demonstrated that our algorithms are efficient, scale to large real-world datasets, and are multiple orders magnitude faster and efficient than existing baseline methods.

We also proposed DESQ, a general-purpose framework for frequent sequence mining. We introduced subsequence predicates as general model for unifying and extending subsequence constraints for FSM. We proposed pattern expressions as a simple, intuitive way to express subsequence constraints, and suggested finite state transducers as an underlying computational model. We provided methods to extend, compress, and optimize our specialized FSTs in order to effectively handle pattern mining task and large hierarchies and proposed the DESQ-COUNT and DESQ-DFS algorithms for efficient frequent sequence mining. Our experiments indicate that DESQ is an efficient general-purpose FSM framework for traditional as well as cus-

tomized subsequence constraints.

Future Work

This work leads to a number of interesting directions for future research. We list some of them that we consider are most important.

Combining scalability and expressibility

Combining expressibility and scalability is perhaps the most important direction for future work. We showed that a general purpose framework greatly improves the usability of pattern mining systems. At the same time, distributed and scalable solutions are essential. The key to achieve scalability lies in partitioning the input sequences by carefully rewriting them into many smaller partitions that can be mined independently and in parallel. We presented such rewrites for traditional notions of subsequence constraints. An interesting direction for future research will be to investigate rewriting techniques for general subsequence constraints, which are modeled using pattern expressions.

Maximality and closedness constraints for generalized subsequences

The set of maximal and closed sequences concisely represent the set of all frequent sequences, thus maximality and closedness constraints restricts output to non-redundant sequences. Our output statistics in Table 5.3; page 79 reveal that up to 95% of generalized sequences can be redundant, depending on the dataset and parameter settings. To the best of our knowledge, maximality and closnedness constraints for generalized subsequences has not been studied in context of generalized sequence mining and is an important research problem.

Mining sequences of itemsets

In this thesis, we focused on sequences of items. Some applications involve sequences of itemsets. For the special case of consecutive subsequences, we showed how our temporal rewrites (Section 3.4) can be used to mine sequences of itemsets in a scalable fashion. We would like to explore how we can extend MG-FSM and LASH to support itemsets for gap-constrained sequences. Also, extending DESQ to support itemsets is also an interesting direction in terms of extending our pattern expression language and FSTs.

Bibliography

Agarwal, R. C., C. C. Aggarwal, and V. V. V. Prasad (2001). A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing 61*(3), 350–371.

Agrawal, R. and R. Srikant (1995). Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, ICDE '95, pp. 3–14.

Albert-Lorincz, H. and J.-F. Boulicaut (2003). Mining frequent sequential patterns under regular expressions: a highly adaptative strategy for pushing constraints. In *Proceedings of the 2003 SIAM International Conference on Data Mining*, pp. 316–320.

Allauzen, C., M. Riley, J. Schalkwyk, W. Skut, and M. Mohri (2007). OpenFst: A general and efficient weighted finite-state transducer library. In *Implementation and Application of Automata*, Volume 4783, pp. 11–23.

Almeida, M., N. Moreira, and R. Reis (2007, June). On the performance of automata minimization algorithms. techreport DCC-2007-03, Universidade do Porto.

Anh, L. V. Q. and M. Gertz (2012). Mining spatio-temporal patterns in the presence of concept hierarchies. In *Proceedings of the 2012 IEEE 12th International Conference on Data Mining Workshops*, ICDMW '12, pp. 765–772.

ANTLR. ANother Tool for Language Recognition). http://www.antlr.org/.

Antunes, C. M. and A. L. Oliveira (2002). *Inference of Sequential Association Rules Guided by Context-Free Grammars*, pp. 1–13. Springer Berlin Heidelberg.

Antunes, C. M. and A. L. Oliveira (2004). Sequential pattern mining with approximated constraints. In *In Proceedings of the International Conference on Applied Computing*, pp. 131–138.

Apache Hadoop. http://hadoop.apache.org.

Appelt, D. E. and B. Onyshkevych (1998). The common pattern specification language. In *TIPSTER*, pp. 23–30.

Ayres, J., J. Flannick, J. Gehrke, and T. Yiu (2002). Sequential pattern mining us-

ing a bitmap representation. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 429–435.

Beedkar, K., K. Berberich, R. Gemulla, and I. Miliaraki (2015). Closing the Gap: Sequence mining at scale. *ACM Trans. Database Syst. 40*(2), 8:1–8:44.

Beedkar, K. and R. Gemulla (2015). LASH: Large-scale sequence mining with hierarchies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, New York, NY, USA, pp. 491–503. ACM.

Beedkar, K. and R. Gemulla (2016). DESQ: Frequent sequence mining with subsequence constraints. In *2016 IEEE International Conference on Data Mining*, pp. 793–798.

Bennett, J. and S. Lanning (2007). The Netflix prize. In *Proceedings of KDD Cup and Workshop*.

Benson, G. and M. Waterman (1994). A method for fast database search for all k-nucleotide repeats. *Nucleic Acids Research 22*(22), 4828–4836.

Berberich, K. and S. Bedathur (2013). Computing n-gram statistics in mapreduce. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pp. 101–112.

Brants, T., A. C. Popat, P. Xu, F. J. Och, and J. Dean (2007). Large language models in machine translation. In *Proceedings of the Conference on Empirical Methods on Natural Language Processing (EMNLP)*, pp. 858–867.

Brazma, A., I. Jonassen, J. Vilo, and E. Ukkonen (1998). Pattern discovery in biosequences. *LNCS 1433*, 257–270.

Brzozowski, J. (1962). Canonical regular expressions and minimal state graphs for definite events. *Mathematical Theory of Automata 12*, 529–561.

Buehrer, G., S. Parthasarathy, S. Tatikonda, T. Kurc, and J. Saltz (2007). Toward terabyte pattern mining: An architecture-conscious solution. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 2–12.

Chen, Y.-L. and T. C.-K. Huang (2008). A novel knowledge discovering model for mining fuzzy multi-level sequential patterns in sequence databases. *Data Knowl. Eng. 66*(3), 349–367.

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

Church, K. W. and P. Hanks (1990, March). Word association norms, mutual information, and lexicography. *Comput. Linguist. 16*(1), 22–29.

Cong, S., J. Han, and D. Padua (2005). Parallel mining of closed sequential patterns. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, pp. 562–567.

Dean, J. and S. Ghemawat (2008, January). Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*(1), 107–113.

Del Corro, L., A. Abujabal, R. Gemulla, and G. Weikum (2015). Finet: Context-aware fine-grained named entity typing. In *Proceedings of the Conference on Empirical Methods on Natural Language Processing (EMNLP)*, pp. 868–878.

Del Corro, L. and R. Gemulla (2013). Clausie: Clause-based open information extraction. In *Proceedings of the International Conference on World Wide Web (WWW)*, pp. 355–366.

Demers, A., J. Gehrke, and B. P (2007). Cayuga: A general purpose event monitoring system. In *In CIDR*, pp. 412–422.

Dindar, N., B. Güç, P. Lau, A. Ozal, M. Soner, and N. Tatbul (2009). Dejavu: Declarative pattern matching over live and archived streams of events. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1023–1026.

Fader, A., S. Soderland, and O. Etzioni (2011). Identifying relations for open information extraction. In *Proceedings of the Conference on Empirical Methods on Natural Language Processing (EMNLP)*, pp. 1535–1545.

Fournier-Viger, P., C.-W. Wu, A. Gomariz, and V. S. Tseng (2014). Vmsp: Efficient vertical mining of maximal sequential patterns. In *Canadian Conference on AI*, pp. 83–94.

Fournier-Viger, P., C.-W. Wu, and V. S. Tseng (2013). Mining maximal sequential patterns without candidate maintenance. In *Advanced Data Mining and Applications*, pp. 169–180.

Garofalakis, M. N., R. Rastogi, and K. Shim (1999). Spirit: Sequential pattern mining with regular expression constraints. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pp. 223–234.

Giannotti, F., M. Nanni, and D. Pedreschi (2006). Efficient mining of temporally annotated sequences. In *SIAM International Conference on Data Mining (SDM)*, pp. 346–357.

Gomariz, A., M. Campos, R. Marín, and B. Goethals (2013). Clasp: An efficient algorithm for mining frequent closed sequences. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pp. 50–61.

Google n-Grams. https://books.google.com/ngrams/.

Guralnik, V., N. Garg, and G. Karypis (2001). *Parallel Tree Projection Algorithm for Sequence Mining*, pp. 310–320. Berlin, Heidelberg: Springer Berlin Heidelberg.

Guralnik, V. and G. Karypis (2004). Parallel tree-projection-based sequence mining algorithms. *Parallel Comput. 30*(4), 443–472.

Han, J., J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu (2000). Freespan: Frequent pattern-projected sequential pattern mining. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, New York, NY, USA, pp. 355–359. ACM.

Han, J., J. Pei, Y. Yin, and R. Mao (2004). Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery 8*(1), 53–87.

Hollink, L., P. Mika, and R. Blanco (2013). Web usage mining with semantic analysis. In *Proceedings of the International Conference on World Wide Web (WWW)*, 2013, Republic and Canton of Geneva, Switzerland, pp. 561–570. International World Wide Web Conferences Steering Committee.

Hsu, C.-M., C.-Y. Chen, B.-J. Liu, C.-C. Huang, M.-H. Laio, C.-C. Lin, and T.-L. Wu (2007). Identification of hot regions in protein-protein interactions by sequential pattern mining. *BMC Bioinformatics 8*(5), 1–15.

Huang, T. C.-k. (2009). Developing an efficient knowledge discovering model for mining fuzzy multi-level sequential patterns in sequence databases. In *Proceedings of the 2009 International Conference on New Trends in Information and Service Science*, NISS '09, pp. 362–371.

Huston, S., A. Moffat, and W. B. Croft (2011). Efficient indexing of repeated n-grams. In *Proceedings of the ACM International Conference on Web Search and Data Mining (WSDM)*, pp. 127–136.

Jang, H. and J. Mostow (2012). Inferring selectional preferences from part-of-speech n-grams. In *European Chapter of the Association for Computational Linguistics (EACL)*.

Jean-Philippe Metivier, S. L. and T. Charnois (2013). A constraint programming approach for mining sequential patterns in a sequence database. In *ECML/PKDD 2013 Workshop on Languages for Data Mining and Machine Learning*, pp. 50–65.

Krishnamurthy, R., Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu (2009). Systemt: A system for declarative information extraction. *SIGMOD Rec. 37*(4), 7–13.

Li, C. and J. Wang (2008). Efficiently mining closed subsequences with gap constraints. In *SIAM International Conference on Data Mining (SDM)*, pp. 313–322.

Li, H., Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang (2008). PFP: Parallel FP-growth

for query recommendation. In *Proc. of the ACM Conf. on Recommender Systems (RecSys)*, RecSys '08, pp. 107–114. ACM.

Liao, Z., D. Jiang, E. Chen, J. Pei, H. Cao, and H. Li (2011). Mining concept sequences from large-scale search logs for context-aware query suggestion. *ACM Trans. Intell. Syst. Technol. 3*(1), 17:1–17:40.

Lin, Y., J.-B. Michel, E. L. Aiden, J. Orwant, W. Brockman, and S. Petrov (2012). Syntactic annotations for the google books ngram corpus. In *Proceedings of the ACL 2012 System Demonstrations*, ACL '12, Stroudsburg, PA, USA, pp. 169–174. Association for Computational Linguistics.

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

Luo, C. and S. M. Chung (2005). Efficient mining of maximal sequential patterns using multiple samples. In *SIAM International Conference on Data Mining (SDM)*, pp. 415–426.

Mannila, H., H. Toivonen, and A. I. Verkamo (1997). Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery 1*(3), 259–289.

Manning, C. D. and H. Schütze (1999). *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press.

Masseglia, F., F. Cathala, and P. Poncelet (1998). The psp approach for mining sequential patterns. In *Proceedings of the Second European Symposium on Principles of Data Mining and Knowledge Discovery*, PKDD '98, London, UK, UK, pp. 176–184. Springer-Verlag.

Miliaraki, I., K. Berberich, R. Gemulla, and S. Zoupanos (2013). Mind the gap: Large-scale frequent sequence mining. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pp. 797–808.

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

Mohri, M. (2000, March). Minimization algorithms for sequential transducers. *Theor. Comput. Sci. 234*(1-2), 177–201.

Mohri, M., F. Pereira, and M. Riley (2002). Weighted finite-state transducers in speech recognition. *Computer Speech & Language 16*(1), 69–88.

Nakashole, N., M. Theobald, and G. Weikum (2011). Scalable knowledge harvesting with high precision and high recall. In *Proceedings of the ACM International Conference on Web Search and Data Mining (WSDM)*, pp. 227–236.

Nakashole, N., G. Weikum, and F. Suchanek (2012). Patty: A taxonomy of relational patterns with semantic types. In *EMNLP-CoNLL*, 2012, Stroudsburg, PA, USA, pp. 1135–1145. Association for Computational Linguistics.

Negrevergne, B. and T. Guns (2015). *Constraint-Based Sequence Mining Using Constraint Programming*, pp. 288–305.

Netspeak. http://www.netspeak.org.

OpenFST. http://www.openfst.org.

PCRE. Perl Compatible Regular Expressions. http://www.pcre.org.

Pei, J., J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu (2001). Prefixspan: Mining sequential patterns by prefix-projected growth. In *Proceedings of the 17th International Conference on Data Engineering*, pp. 215–224.

Pei, J., J. Han, B. Mortazavi-Asl, and H. Zhu (2000). Mining access patterns efficiently from web logs. In *Proceedings of the 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining, Current Issues and New Applications*, PADKK '00, pp. 396–407.

Pei, J., J. Han, and W. Wang (2002). Mining sequential patterns with constraints in large databases. In *Proceedings of the Conference on Information and Knowledge Management (CIKM)*, pp. 18–25.

Plantevit, M., A. Laurent, D. Laurent, M. Teisseire, and Y. W. Choong (2010). Mining multidimensional and multilevel sequential patterns. *ACM Trans. Knowl. Discov. Data 4*(1), 4:1–4:37.

Plantevit, M., A. Laurent, and M. Teisseire (2006). Hype: Mining hierarchical sequential patterns. In *Proceedings of the 9th ACM International Workshop on Data Warehousing and OLAP*, DOLAP '06, pp. 19–26.

PROSITE. http://prosite.expasy.org/.

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

SMA. http://www-kdd.isti.cnr.it/SMA/.

Srikant, R. and R. Agrawal (1996). Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '96, pp. 3–17.

Srivastava, J., R. Cooley, M. Deshpande, and P.-N. Tan (2000). Web usage mining: Discovery and applications of usage patterns from web data. *SIGKDD Explor. Newsl. 1*(2), 12–23.

Stanford CoreNLP parser. http://nlp.stanford.edu/software/corenlp.shtml.

Stolcke, A. (2002). SRILM - an extensible language modeling toolkit. In *Interspeech*.

Tandon, N., G. De Melo, and G. Weikum (2014). Acquiring comparative commonsense knowledge from the web. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, AAAI'14, pp. 166–172. AAAI Press.

The New York Times corpus (2008). https://catalog.ldc.upenn.edu/LDC2008T19.

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

Trasarti, R., F. Bonchi, and B. 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 (ICDE)*, pp. 1061–1066.

Trummer, I., A. Halevy, H. Lee, S. Sarawagi, and R. Gupta (2015). Mining subjective properties on the web. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1745–1760.

Wang, J. and J. Han (2004). BIDE: Efficient mining of frequent closed sequences. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pp. 79–90.

Wang, K., Y. Xu, and J. X. Yu (2004). Scalable sequential pattern mining for biological sequences. In *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management*, CIKM '04, pp. 178–187.

Wang, W. and D. Vergyri (2006). The use of word n-grams and parts of speech for hierarchical cluster language modeling. In *ICASSP*.

Web data: Amazon reviews. snap.stanford.edu/data/web-Amazon.html.

Yan, X., J. Han, and R. Afshar (2003). Clospan: Mining closed sequential patterns in large databases. In *SIAM International Conference on Data Mining (SDM)*, pp. 166–177.

Zaki, M. J. (2000). Sequence mining in categorical domains: Incorporating constraints. In *Proceedings of the Ninth International Conference on Information and Knowledge Management*, CIKM '00, pp. 422–429.

Zaki, M. J. (2001a). Parallel sequence mining on shared-memory machines. In *Journal of Parallel and Distributed Computing*, pp. 401–426. Springer-Verlag.

Zaki, M. J. (2001b). Spade: An efficient algorithm for mining frequent sequences. *Mach. Learn. 42*(1-2), 31–60.

List of Figures

2.1 An example sequence database . . . 6
2.2 Frequent sequence enumeration using BFS approach. . . . 8
2.3 Some inverted indexes and supports for each sequence explored by by BFS. . . . 8
2.4 Frequent sequence enumeration using a DFS approach. . . . 11
2.5 Some expansions, projected databases and supports. . . . 11

3.1 Pivot sequence enumeration for partition $\mathscr{P}_d$ for $\sigma = 2$, $\gamma = 1$ and $\lambda = 4$. 24
3.2 Effectiveness of PSM . . . 32
3.3 Temporal sequences . . . 32
3.4 Effectiveness of indexing long sequences . . . 35

4.1 Mining maximal sequences with MG-FSM$^+$ ($\sigma = 2$, $\gamma = 1$, $\lambda = 4$). . . . 46
4.2 Mining closed sequences with MG-FSM$^+$ ($\sigma = 2$, $\gamma = 1$, $\lambda = 4$). . . . 50
4.3 Performance of mining maximal and closed sequences . . . 52

5.1 A sequence database and its vocabulary . . . 57
5.2 Preprocessing, partitioning and mining phases of LASH for $\sigma = 2$, $\gamma = 1$ and $\lambda = 3$. . . . 68
5.3 Performance of distributed algorithms. . . . 74
5.4 Performance of sequential algorithms. . . . 74
5.5 Effect of different parameters . . . 75
5.6 Effect of different hierarchies. . . . 75
5.7 Scalability results . . . 77

6.1 A sequence database and its vocabulary . . . 84
6.2 FST for the pattern expression $[c|d]([A^\uparrow \mid B^{\uparrow}_{=}]^+)e$. . . . 92
6.3 FST for basic item expression $(A^\uparrow)$ . . . 93
6.4 Example annotated hierarchy. . . . 96
6.5 Excerpt of an annotated item hierarchy for text mining applications. . . . 100
6.6 FST for advanced item expression $[\text{level=M} \wedge \text{type=y}]^{\uparrow}_{=}[\text{level=R}]_{=}$. . . . 102

7.1 A sequence database and its vocabulary . . . 104
7.2 Compressed FST for $[c|d]([A^\uparrow \mid B^{\uparrow}_{=}]^+)e$. . . . 106

7.3 Minimized cFST for $[c|d]([A^{\uparrow} \mid B^{\uparrow}_{=}]^{+})e$. 106
7.4 Illustration of DESQ-DFS for $\mathcal{D}_{ex}$, $\mathcal{A}_{Fex}$, and $\sigma = 2$ 114
7.5 An Example hierarchy. 116
7.6 cFST for pattern expression $c\,(A)\,\big[(d)e \mid (B^{\uparrow}d) \mid (B'e)\big]$. 116
7.7 Minimizing cFST for expression $(Ab_1|Ac)$ via Brzozowski' algorithm. . 118
7.8 Compressed NFA obtained from cFST $\mathcal{A}_{F_{7.6}}$. 119
7.9 NFA. 121
7.10 DFA. 121
7.11 Compressed DFA for $c\,(A)\,\big[(d)e \mid (B^{\uparrow}d) \mid (B'e)\big]$. 122
7.12 Transcript of conversion for cFST of Figure 7.6 to cDFA of Figure 7.11 . 122
7.13 cFST for pattern expression $c\,(A)\,\big[(d)e \mid (B^{\uparrow}d) \mid (B'e)\big]$. 125
7.14 Illustration of two-pass method for $T = ca_1b_{12}e$. 125
7.15 Performance of DESQ mining algorithms. The numbers on top of the bars indicate the average number of P-subsequences per input sequence. 133
7.16 Effectiveness of pruning input sequences and two-pass in DESQ-COUNT and DESQ-DFS for pattern expression N_1–N_5 and A_1–A_4. 135
7.17 Overhead of DESQ for RE constraints and traditional subsequence constraints. 138

List of Tables

3.1 Dataset characteristics . . . 31
3.2 Example frequent sequences from Netflix ($\sigma = 1000, \lambda = 5, \tau = 1\ day$) 33

5.1 Dataset characteristics . . . 71
5.2 Hierarchy characteristics . . . 71
5.3 Output Statistics . . . 79

6.1 Translation rules for basic item expressions (where $w, w', w'' \in \Sigma$) . . . 88
6.2 Pattern expressions for traditional subsequence constraints. . . . 90
6.3 Pattern expressions for subsequence constraints in information extraction and natural language processing applications. . . . 90
6.4 Pattern expressions for subsequence constraints in customer behavior mining applications. . . . 90
6.5 Item descriptors and their corresponding examples. . . . 97
6.6 Item expressions for advanced pattern expression language. . . . 98
6.7 Basic item expressions and their corresponding advanced item expressions. . . . 99
6.8 Some examples of advanced pattern expressions useful for text mining applications. . . . 101

7.1 Translation rules for item expressions (where $w, w', w'' \in \Sigma$) to compressed FST. . . . 105
7.2 Dataset statistics . . . 130
7.3 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) . . . 131
7.4 Runtimes of Naïve with cFST and openFST . . . 132
7.5 cFST simulation statistics for DESQ-COUNT and DESQ-DFS . . . 137

List of Algorithms

2.1 Breadth-first search 7
2.2 Depth-first search 10
3.1 The MG-FSM algorithm 18
3.2 Mining pivot sequences 23
4.1 The MG-FSM$^+$ algorithm 44
5.1 Naïve GSM approach 59
5.2 Computing generalized f-list 60
5.3 Partitioning and mining phase of LASH 62
7.1 Simulate a cFST 108
7.2 Naïve approach 110
7.3 DESQ-DFS 113
7.4 Convert a cFST to a cDFA 123
7.5 Two-pass simulation 127
7.6 Integrating two-pass approach in DESQ-DFS 128