Foundations of Regular Languages for Processing RDF and XML

Von der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von

Katja Losemann

Tag der Einreichung: 15. April 2015

Deutsche Kurzfassung

Die vorliegende Arbeit befasst sich mit der Komplexität und Optimierung von Problemen, die bei der Verwendung von regulären Sprachen für die web-basierte Datenverarbeitung entstehen. Die betrachteten Problemstellungen sind insbesondere durch Anwendungen motiviert die eines der folgenden zwei Datenformate benutzen: die *Extensible Markup Language (XML)* [BPSM$^+$08] und das *Resource Description Framework (RDF)* [CWL14]. Wir werden uns zunächst mit der Komplexität von regulären Sprachen in dem Kontext XML und RDF beschäftigen. Danach untersuchen wir die Auswertung von dynamischen Daten, die aus heutiger Sicht immer relevanter für die web-basiere Datenverarbeitung wird.

In dem ersten Teil dieser Arbeit liegt unser Fokus auf *regulären Ausdrücken* (kurz *RAs*) zur Repräsentation von regulären Sprachen. Reguläre Ausdrücke haben sich in praktischen Anwendungen für die XML- und RDF-Datenverarbeitung weitestgehend gegenüber anderen Repräsentationen für reguläre Sprachen, wie zum Beispiel endlichen Automaten oder logischen Charakterisierungen, als effizient wie auch nutzerfreundlich durchgesetzt. Um den einzelnen Anforderungen in den verschiedenen praktischen Anwendungen gerecht zu werden, sind außerdem mit der Zeit mehrere Varianten regulärer Ausdrücke entstanden. Eben diese Varianten und deren Komplexität sind Betrachtungsgegenstand unserer Forschung in dem ersten Teil dieser Dissertation. Zu diesem Zweck werden wir verschiedene Varianten regulärer Ausdrücke, die im Kontext der XML- und RDF-Datenverarbeitung benutzt werden, formal definieren und untersuchen, ob diese eine ihrem Verwendungszweck entsprechend effiziente Nutzung zulassen oder eventuell sogar verhindern. Wir beschäftigen uns dabei im Detail mit *Schemasprachen* für XML und *Anfragesprachen* für RDF, in denen durch semantische und syntaktische Zusatzbedingungen einige interessante Varianten regulärer Ausdrücke entstanden sind.

Anschließend werden wir in dem Kontext von Anfragesprachen die Komplexität regulärer Sprachen aus einer etwas allgemeineren Perspektive untersuchen. Dafür lösen wir unsere Betrachtungen sowohl von konkreten Datenformaten als auch von regulären Ausdrücken. Stattdessen untersuchen wir die web-basierte Datenverarbeitung mit dem Fokus auf das Verhalten moderner Datenbanken. In diesen sind heutzutage zwei Tendenzen deutlich zu erkennen: Erstens, die in den Datenbanken gespeicherten Datensätze werden immer größer; sie nehmen dabei Dimensionen an die mit klassischen Methoden nicht mehr effizient beherrschbar sind. Aus diesem Grund ist zu erwarten, dass die Ergebnisse von Datenbankanfragen immer komple-

xer werden und dem Nutzer eine (möglicherweise sehr große) Menge an korrekten Ergebnissen zurückgegeben wird. Zweitens, unterliegen die in den Datenbanken enthaltenen Daten ständigen Änderungen, so dass eine gerade abgeschlossene Analyse der Daten im nächsten Moment schon wieder veraltet sein kann.

Wir stellen uns die Frage, ob reguläre Datenbankanfragen in diesem Kontext effizient ausgewertet werden können. Um große Antwortmengen effizient verarbeiten zu können, untersuchen wir sogenannte *Enumerationsalgorithmen*, die das Ergebnis einer Anfrage nicht direkt in einem Schritt berechnen, sondern vielmehr Stück für Stück an den Nutzer zurückgeben. Da wir außerdem dynamische Daten betrachten, sollen diese Algorithmen zudem auf eintreffende Datenupdates schnell reagieren können. Dabei betrachten wir Anfragen, die in ihrem Kern regulär sind, und durch ein spezielles endliches Automatenmodell dargestellt werden können. Obwohl für ähnliche Anfragen bereits effiziente Enumerationsalgorithmen bekannt und gut untersucht sind (siehe zum Beispiel [Bag06, Cou09, DS11, KS13a, KS13b, Seg13]), können diese Algorithmen nicht dafür genutzt werden Anfragen über dynamischen Datenbanken effizient auszuwerten.

Im Folgenden erläutern wir den Aufbau dieser Dissertation und beschreiben die erarbeiteten Ergebnisse im Einzelnen.

Eine ausführliche Einleitung in unser Forschungsthema wird in **Kapitel 1** gegeben. Die notwendigen Grundlagen der Theoretischen Informatik, der Formalen Sprachen und der betrachteten Datenformate werden in **Kapitel 2** eingeführt und anhand von Beispielen erläutert. Um dem Leser den Überblick über die im Folgenden benutzten Varianten regulärer Ausdrücke zu erleichtern, werden wir in **Kapitel 3** die untersuchten semantischen und syntaktischen Zusatzbedingungen für reguläre Ausdrücke formal definieren und bereits bekannte relevante Ergebnisse wiederholen. Anschließend wenden wir uns den folgenden Forschungsthemen zu.

Reguläre Ausdrücke in dem Kontext XML-Schemasprachen Seit 1998 wird die Meta-Sprache XML von dem World-Wide-Web Consortium (W3C) als Standardformat für semi-strukturierte Daten empfohlen [BPSM$^+$08]. Die in einem XML-Dokument enthaltenen Daten werden dabei hierarchisch in einer Baumstruktur formatiert. Um zusätzliches Wissen über diese Struktur — sofern vorhanden — für die automatisierte Datenverarbeitung nutzen zu können, werden in der Praxis *XML-Schemasprachen* verwendet, die es dem Nutzer erlauben ein *Schema* über dem von ihm gewünschten Vokabular zu definieren, das dann die hierarchische Struktur von unter dem Schema gültigen XML-Dokumenten zweckmäßig einschränkt. In diesem Sinne stellt das Schema eine Art Schablone für die Struktur der zulässigen Dokumente dar. Das in dem Schema enthaltene Wissen kann in der Praxis zum Beispiel dafür genutzt werden Anfragen oder den Austausch von Daten effizienter auszuwerten. Die zurzeit am häufigsten genutzten XML-Schemasprachen sind *Document Type Definitions (DTDs)* [BPSM$^+$08], *XML Schema (XSDs)* [FW04] und *Relax NG* [CM01].

Für uns interessant ist die Verwendung von regulären Ausdrücken in den ersten beiden Schemasprachen. In diesen darf aufgrund einer Zusatzbedingung, die aus

Kompatibilitätsgründen in den Spezifikationen von DTDs und XSDs des W3C's enthalten ist, nur eine semantisch eingeschränkte Art von regulären Ausdrücken verwendet werden. Diese Art von Ausdrücken werden aufgrund ihrer Eigenschaften als *deterministische reguläre Ausdrücke* (oder *DRAs*) bezeichnet und in **Kapitel 4** genauer untersucht. Während ein deterministisches Verhalten Probleme aus der Informatik in vielen Fällen stark vereinfachen kann, werden wir sehen dass in diesem Fall grundlegende Probleme erheblich erschwert werden.

Zwei Eigenschaften von DRAs sind dabei besonders unangenehm für den Nutzer. Es ist bereits seit 1998 bekannt, dass nicht alle regulären Sprachen durch DRAs definiert werden können [BKW98], so dass Nutzer nicht beliebige reguläre Sprachen in einem Schema verwenden können. Zusätzlich ist für den Nutzer im Allgemeinen nur schwer erkennbar welche Ausdrücke in einem Schema gültig sind. Deswegen ist es in der Praxis erforderlich die Integrität von Schemata automatisiert zu testen um das Schema gegebenenfalls dem Nutzer zur Korrektur zurückgeben zu können. Um einen solchen Test durchzuführen muss insbesondere auch das folgende Problem gelöst werden, mit dessen Komplexität wir uns in Abschnitt 4.1 beschäftigen:

> Sei eine reguläre Sprache L gegeben.
> Kann die Sprache L auch durch einen DRA definiert werden?

Dieses Problem werden wir im weiteren Verlauf als DRE-Definability Problem bezeichnen. Bereits 1998 haben Büggemann-Klein und Wood gezeigt, dass DRE-Definability entscheidbar ist [BKW92, BKW98]. Daher sind wir im Folgenden an der exakten Komplexität des Problems interessiert. Die hier vorgestellten Ergebnisse sind in Zusammenarbeit mit Wojciech Czerwiński, Claire David und Wim Martens entstanden und wurden zuvor in [CDLM13] veröffentlicht. Als Grundlage unserer Arbeit dienten uns insbesondere die folgenden zwei bekannten Ergebnisse: DRE-Definability kann in polynomieller Zeit entschieden werden wenn die Eingabe ein minimaler *deterministischer endlicher Automat* (*DEA*) ist [BKW98] und es ist PSPACE-schwer wenn die Eingabe ein *nichtdeterministischer endlicher Automat* (*NEA*) oder ein RA ist [BGMN09].

Wir beweisen dass DRE-Definability für NEAs und RAs PSPACE-vollständig ist indem wir einen Algorithmus angeben der auf dem Ergebnis von Brüggemann-Klein und Wood in [BKW98] aufbaut und das Problem für NEAs in PSPACE löst. Mit Hilfe dieses Algorithmus zeigen wir außerdem dass DRE-Definability für RAs mit Zähloperatoren in EXPSPACE ist. Eine dazu passende untere Schranke beweisen wir durch Adaption einer Reduktion von Groz [Gro12]. Abschließend geben wir noch eine NLOGSPACE untere Schranke für DRE-Definability mit minimalen DEAs als Eingabe an.

In Abschnitt 4.2 befassen wir uns mit der Beschreibungskomplexität von deterministischen regulären Ausdrücken. Das heißt, wir untersuchen, ob — verglichen mit anderen Repräsentationen für reguläre Sprachen — die Repräsentation als DRA mit weniger oder mehr Aufwand möglich ist. Dabei werden wir zeigen, dass im Allgemeinen DRAs exponentiell größer als RAs und DEAs sind. Außerdem zeigen wir in Abschnitt 4.3 dass sich DRAs unter der Anwendung von Booleschen Operationen nicht optimal verhalten. Hierbei kann bereits die einmalige Anwendung einer Boole-

schen Operation auf zwei gegebene DRAs in einem DRA resultieren der exponentiell größer ist.

Die Ergebnisse in den Abschnitten 4.2 und 4.3 sind zum Teil in Zusammenarbeit mit Wim Martens und Matthias Niewerth entstanden und wurden zuvor in [LMN12] veröffentlicht. Einige der Ergebnisse aus [LMN12] stammen dabei bereits aus der Diplomarbeit der Autorin und werden aus diesem Grund zusätzlich mit [Los10] gekennzeichnet. Ergebnisse ohne entsprechende Referenzen wurden allein von der Autorin verfasst und sollen in Zukunft in einer Journalversion zu [LMN12] veröffentlicht werden.

Die Komplexität von Regulären Ausdrücken in SPARQL 1.1 Anfragen Um große unstrukturierte Datenmengen effizient auswerten zu können, entwickelte das W3C das *Resource Description Framework (RDF)* [CWL14], in dem Daten als Graphen dargestellt werden und mit beliebigen öffentlichen Bezeichnern gekennzeichnet werden. Zur Auswertung von RDF-Daten entwickelt das W3C seit 2008 außerdem die Anfragesprache *SPARQL Protocol and RDF Query Language (SPARQL)*[1]. Wir untersuchen die Komplexität der Auswertung von SPARQL Anfragen im Hinblick auf das sogenannte *Property Path Feature*. Dieses wurde 2010 zu der SPARQL 1.1 Spezifikation hinzugefügt und erlaubt es reguläre Ausdrücke in Anfragen zu verwenden. Als Grundlage für unsere Studie dient uns der vorläufige Entwurf der SPARQL Anfragesprache aus 2012 [HS12].

Laut dieser Spezifikation sind in SPARQL Anfragen reguläre Ausdrücke mit einigen zusätzlichen Operatoren, wie beispielsweise Zähloperatoren und eine eingeschränkte Negation, erlaubt. Außerdem wird in der Spezifikation eine spezielle Semantik zur Auswertung von Ausdrücken über Graphen definiert, die verlangt dass

- bestimmte Ausdrücke nur über *einfachen* Pfaden (d.h. über Pfaden in denen jeder Knoten maximal einmal besucht wird) ausgewertet werden dürfen und
- die Anzahl der richtigen Antworten auf eine Anfrage bestimmt werden muss.

Diese besonderen Bedingungen haben uns dazu veranlasst im November 2011 eine kurze Studie aktueller SPARQL Implementierungen durchzuführen. Im Einzelnen wurden die folgenden Implementierungen betrachtet: das Jena Semantic Web Framework [Apa11], Sesame [KFB08], RDF::Query [Wil], und Corese 3.0 [Cor12]. Im Laufe unserer Studie überraschte es uns, dass bereits für sehr einfache Anfragen auf sehr kleinen Graphen (mit nur zwei Knoten) alle Implementierungen doppeltexponentiell viel Zeit in der Größe der Eingabe benötigten um eine Antwort zu berechnen. Aus diesem Grund haben wir formal untersucht, ob ein solches Verhalten vermieden werden kann oder ob eine effiziente Auswertung von SPARQL Anfragen unter den oben-genannten Bedingungen nicht möglich ist. Unsere Ergebnisse präsentieren wir in **Kapitel 5**. Die vorgestellten Ergebnisse sind dabei in Zusammenarbeit mit Wim Martens entstanden und wurden zuvor in [LM12, LM13] veröffentlicht.

Um mögliche Ursachen für das ineffiziente Verhalten der SPARQL Implementierungen aufzudecken und unabhängig voneinander bewerten zu können, definieren

[1] Der Name der Anfragesprache ist tatsächlich ein rekursives Akronym.

wir zu Beginn des Kapitels in den Abschnitten 5.1 und 5.2 zwei verschiedene Semantiken für SPARQL Anfragen. Die erste Semantik, genannt *Simple-Walk-Semantik*, entspricht unserer Abstraktion der von dem W3C gewünschten Semantik einschließlich der oben genannten Bedingungen. Die zweite Semantik, genannt *Regular-Path-Semantik*, vernachlässigt indes die Bedingung bestimmte Ausdrücke auf einfachen Pfaden auszuwerten.

In Abschnitt 5.3 untersuchen wir zunächst die Komplexität der zusätzlichen Operatoren die laut Spezifikation in den Ausdrücken erlaubt sind. Insbesondere gilt für diese Operatoren, dass durch die verwendeten Zähloperatoren reguläre Ausdrücke im Allgemeinen exponentiell kürzer werden [KT03]. Für reguläre Ausdrücke (ohne zusätzliche Operatoren) war dagegen bereits vorher bekannt, dass diese über Graphen in polynomieller Zeit ausgewertet werden können [MW95, AV99, ABE09]. Wir zeigen, dass auch die zusätzlichen SPARQL Operatoren (inklusive der Zähloperatoren) auf Graphen in polynomieller Zeit ausgewertet werden können. Dabei ist zu beachten, dass dieses Ergebnis nur für die Auswertung unter Regular-Path-Semantik gilt, das heißt ohne die zusätzlichen Bedingungen aus der SPARQL Spezifikation. Wir zeigen, dass die Auswertung von SPARQL Anfragen unter Simple-Walk-Semantik bereits für stark eingeschränkte reguläre Ausdrücke NP-schwer und für die Menge aller erlaubten SPARQL Ausdrücke NP-vollständig ist. Eine effiziente Umsetzung der Bedingung, dass alle richtigen Antworten zu einer Anfrage gezählt werden müssen, ist sogar unter beiden Semantiken nicht effizient umsetzbar. In den Abschnitten 5.4 und 5.5 beweisen wir, dass das dazugehörende Zählproblem bereits für stark eingeschränkte reguläre Ausdrücke #P-schwer und für alle SPARQL Ausdrücke #P-vollständig ist.

Zusammenfassend können wir sagen, dass die zusätzlich definierten Operatoren kein Problem in der SPARQL Spezifikation darstellen. Die Einschränkung auf einfache Pfade und die Bedingung, dass Antworten gezählt werden sollen, verhindern dagegen eine effiziente Implementierung. Unter Regular-Path-Semantik wäre eine effiziente Implementierung von SPARQL Anfragen jedoch umsetzbar. In Abschnitt 5.6 zeigen wir darüber hinaus, dass unsere Ergebnisse für die Auswertung von Anfragen unter Regular-Path-Semantik auch auf verschachtelte Ausdrücke (sogenannte *Nested Regular Expressions* [PAG10]) erweitert werden können.

Abschließend möchten wir noch betonen, dass ähnliche Ergebnisse über die Komplexität von SPARQL Anfragen zeitgleich zu unserer Arbeit in [LM12] von Arenas et al. in [ACP12] vorgestellt wurden. Zusammen konnten wir überzeugende Argumente vorlegen die Spezifikation von SPARQL 1.1 Anfragen, die seit 2013 auch offiziell vom W3C für die Auswertung von RDF-Daten empfohlen wird, in eine entsprechende Richtung anzupassen. So wurden mittlerweile die Auswertung auf einfachen Pfaden und das Zählen von Antworten aus der Spezifikation entfernt. Bedauerlicherweise sind außerdem auch die Zähloperatoren aus der Spezifikation entfernt worden. Eine ausführliche Diskussion über die Veränderungen in der aktuellen SPARQL Spezifikation [HS13] wird in Abschnitt 5.7 gegeben.

Enumerationsalgorithmen für Reguläre Anfragen auf Dynamischen Daten Wie oben bereits erwähnt untersuchen wir zum Schluss dieser Dissertation die Komplexität von regulären Anfragesprachen mit unserem Fokus auf Anfragen, die potenziell eine große Menge von richtigen Antworten an den Nutzer zurückgeben, und auf dynamischen Daten, die durch Datenupdates regelmäßig verändert werden. Um unter diesen Bedingungen Anfragen weiterhin effizient auswerten zu können untersuchen wir in **Kapitel 6** Enumerationsalgorithmen, die auf eintreffende Datenupdates schnell reagieren können.

Als Anfragen dienen uns spezielle endliche Automaten, sogenannte *k-ary nondeterministic finite selecting (tree) automata*, die wir in Abschnitt 6.1 formal definieren und die über Wörtern und Bäumen ausgewertet werden können. Eine formale Definition der betrachteten Probleme geben wir in Abschnitt 6.2. Die im Folgenden präsentierten Ergebnisse sind in Zusammenarbeit mit Wim Martens entstanden und wurden zu großen Teilen bereits in [LM14] veröffentlicht.

Für die Auswertung über Wörtern der Länge n konstruieren wir in Abschnitt 6.3 einen Algorithmus, der für eine gegebene Anfrage Vorberechnungen in Zeit $O(n)$ durchführt und danach alle Antworten mit $O(\log n)$ Verzögerung nacheinander ausgibt. Trifft ein Update für das Wort ein hält der Algorithmus sofort an und kann nach Zeit $O(\log n)$ erneut damit anfangen richtige Antworten für die Anfrage auszugeben. In Abschnitt 6.4 erweitern wir diesen Algorithmus für die Auswertung über Bäumen der Größe n. Der vorgestellte Algorithmus gibt, für eine gegebene Anfrage und nach Vorberechnungen in Zeit $O(n)$, alle Antworten mit einer Verzögerung von $O(\log^2 n)$ nacheinander aus und reagiert auf Updates in Zeit $O(\log^2 n)$. In den einzelnen Abschnitten erläutern wir die genaue Funktionsweise der Algorithmen, beweisen ihre Korrektheit und führen außerdem eine genauere Komplexitätsanalyse durch.

Zusätzlich zeigen wir in Abschnitt 6.5, dass unsere Ergebnisse auch für Anfragen, deren Ergebnis ein *Multiset* ist, angewendet werden können. Ein praktisches Anwendungsbeispiel für unsere Ergebnisse erläutern wir in Abschnitt 6.6 näher. In Abschnitt 6.7 diskutieren wir abschließend zukünftige Anwendungen und mögliche Optimierungen unserer Ergebnisse.

Schließlich fassen wir in **Kapitel 7** die Ergebnisse dieser Dissertation noch einmal kurz zusammen und diskutieren mögliche zukünftige Forschungsfragen im Detail.

Die meisten Ergebnisse in dieser Dissertationsschrift sind bereits im Vorfeld in folgenden Publikationen veröffentlicht worden: [LMN12, LM12, Los12, CDLM13, LM13, LM14].

Acknowledgements

First of all, my thanks go to my advisor Wim Martens for all his support during my time as a PhD student. After finishing my diploma in computer science at the end of 2010, he gave me the opportunity to join his group and start my PhD studies at the TU Dortmund. After approximately one year in Dortmund, the time was come to move to Bayreuth. Although our research group in Bayreuth was not very big, I always felt that I was exposed to many interesting new research problems that not only provided a good foundation for our research but also were attractive due to their practical relevance. I am very grateful for the time Wim spent with me in a lot of fruitful discussions on our research problems.

Special thanks go to Claire David for introducing me to new research topics and for many interesting discussions. I thank Thomas Schwentick for providing constructive comments on my work every time I visited my former group at the TU Dortmund. For his careful proofreading I thank Thomas Zeume.

I also thank my family and friends who always supported me during my time as a PhD student.

Bayreuth, April 2015
Katja Losemann

Contents

1

Introduction

We investigate the complexity of regular languages for data processing on the web. More precisely, our focus is on problems arising from two particular data formats: the *Extensible Markup Language (XML)* [BPSM$^+$08] and the *Resource Description Framework (RDF)* [CWL14]. Regular languages can be found in a wide array of technology on the web. To process XML- or RDF-data, *regular expressions* (*REs*) became a well-established concept because of their user-friendly behavior and good complexity compared to other representations for regular languages, e.g., finite automata or logical characterizations. Thereby, different practical applications require different variants of regular expressions that are enhanced with additional operators or restrained by semantical constraints.

These variants and their computational complexity are the main object of our studies in the first part of this dissertation. We examine a wide range of distinct classes of regular expressions that are used in prevalent XML or RDF applications. In detail, we focus on the use of regular expressions in *schema languages* for XML and *query languages* for RDF where certain syntactical and semantical constraints form interesting and challenging variants of regular expressions. Although some of these constraints are very specific, it is the case that several of them are reused in different applications in another context. Therefore, our results will likely be relevant in other contexts than XML and RDF processing as well.

In the last part of this dissertation we study query languages for data processing on the web from a more general point of view. Here, our main motivation comes from two problems arising in today's databases: First, the amount of data that is stored in databases exploded during the last decade such that classical methods from database theory are sometimes not sufficient anymore. When querying these data it is likely that queries return a huge set of rightful answers in general. Second, very often the stored data is dynamic, i.e., the data is subject to frequent updates over time. We are interested in whether efficient query evaluation is still feasible under these constraints. Towards a positive answer, we examine

enumeration algorithms which, after some preprocessing phase, compute the query result step by step and output answers sequentially within a certain delay. Although enumeration algorithms for similar queries were examined in the literature before [Bag06, Cou09, DS11, KS13a, KS13b, Seg13], there exist - as far as we know - no such algorithms that can efficiently handle data updates. For that matter we investigate whether it is possible to construct an enumeration algorithm for regular queries that can quickly recompute the new query result after a data update occurred.

We present a detailed overview of the structure of this dissertation next.

In **Chapter 2** we review basic notions of theoretical computer science and define abstract models for XML and RDF data. Moreover, we review fundamental results that are used throughout the dissertation. A complete overview of the different classes of regular expressions that are examined in this dissertation can be found in **Chapter 3**.

Afterwards, we investigate the complexity of the following problems on regular languages. Although we motivate the problems under consideration by prevalent practical applications, we formalize our results independently (from the particular data format, schema language, etc.) in order to emphasize their connection to formal language theory.

Regular Expressions in XML Schema Languages The *Extensible Markup Language (XML)* is the W3C-recommended standard for semi-structured data since 1998 [BPSM$^+$08]. Data in an XML document is hierarchically formatted in a tree-like form via the use of application-defined tags and labels. To use additional knowledge about the structure of XML documents, the user can use the option to write schemas. Such schemas, which are written in an *XML schema language*, constrain the hierarchical structure of valid XML documents to the particular purpose. In this way schemas serve as patterns for valid XML-documents. In the presence of a schema, applications can then be optimized to exploit the schema information before processing the XML document itself, e.g., schemas can be used to make query answering or data exchange for XML more efficient. Renowned schema languages for XML are *Document Type Definitions (DTDs)* [BPSM$^+$08], *XML Schema (XSDs)* [FW04], and *Relax NG* [CM01].

In **Chapter 4** we examine the use of regular expressions in the first two of the above XML schema languages. In DTD- and XSD-schemas it is not allowed to use arbitrary regular expressions due to compatibility reasons with *SGML*, which is a more general mark-up language to format documents. Instead, the user is compelled to use expressions that can always be processed in a deterministic way. These expressions are also known as *deterministic regular expressions* (or *DREs*) in the literature [BKW98]. For example, the expression $(a + b)^*a(a + b)^*$ is not deterministic because when reading an a in the beginning of a word it is ambiguous whether the a should be "matched" to the first or the second a in the expression. The equivalent expression $b^*a(a + b)^*$, however, is deterministic. Thus, in order to apprehend the complexity of schema languages for XML, it is important to develop

a good understanding of DREs first. Moreover, since the concept of determinism in regular expressions is rather fundamental, we believe our results to be relevant in a larger scope as well.

The concept of determinism is often used to decrease the complexity of computational problems in computer science. Although there exist problems that are computationally much easier for DREs (e.g., language inclusion [Hov12] or constructing an equivalent deterministic finite automaton [BKW92]), we believe that there are more drawbacks than benefits of this representation for regular languages in the context of schema languages for XML. One property of the class of DREs that is especially cumbersome to the user is that not every regular language can be defined via a deterministic regular expression [BKW98]. For example, the language of the expression $(a+b)^*a(a+b)$ cannot be expressed by an equivalent deterministic expression [BKW98]. Therefore, not every regular language can be used in a DTD or XSD schema such that the following natural question has to be answered when checking schemas for validity:

Given a regular language L, is it possible to define L as a DRE?

It is already known that the corresponding problem is decidable [BKW92, BKW98]. Here, we are interested in the exact complexity of the problem, which we investigate in Section 4.1. The following relevant complexity results were known before: The problem can be solved in polynomial time when the input language is given as a minimal *deterministic finite automaton* (or a *DFA*) [BKW98], and the problem is PSPACE-hard when the input is given as an RE or a *nondeterministic finite automaton* (*NFA*) [BGMN09]. Using the polynomial-time result of Brüggemann-Klein and Wood, it follows that the problem can also be solved for REs and NFAs by converting them into a minimal DFA before running their procedure. In this way, an exponential upper bound on the problem for REs and NFAs is achieved.

We can improve this upper bound and show that the problem can be solved in PSPACE for REs and NFAs, i.e., together with the lower bound by Bex et al. we show that the problem is PSPACE-complete. Towards the proof, we exhibit structural properties of the polynomial-time decision algorithm of Brüggemann-Klein and Wood which we exploit to show that their algorithm can be implemented for NFAs in PSPACE. Although the proof is rather technical, it provides fundamental insights into the class of deterministic regular expressions which might be useful for future work on DREs as well.

Using our PSPACE algorithm for NFAs we also obtain an EXPSPACE upper bound for the problem when the input is a regular expression with counting operators. We also prove that this problem is EXPSPACE-complete by providing a matching lower bound via a reduction from universality for regular expressions with squaring [MS72]. At last, we provide an NLOGSPACE lower bound for the problem when the input is given as a minimal DFA by a reduction from the reachability problem in directed acyclic graphs [Jon75].

The aforementioned results are joint work with Wojciech Czerwiński, Claire David and Wim Martens and have been previously published in [CDLM13].

We have seen that it is not easy to decide whether a particular language can be used in a schema or not due to the determinism constraint in DTDs and XSDs. However, if we assume that this problem is solved then the user may want to write an expression that is as succinct as possible to optimize the processing of the schema. By examining the relative descriptional complexity of deterministic regular expressions, we prove in Section 4.2 that such a succinct expression does not always exist. More precisely, we investigate whether deterministic regular expressions are more succinct than other representations for regular languages, e.g., finite automata or (arbitrary) regular expressions. In fact, we show that, when converting an RE to a minimal DRE, an exponential blow-up cannot be avoided in general. To strengthen these results we consider in Section 4.3 the descriptional complexity of several operations on deterministic regular expressions, i.e., we study the complement, union, intersection, reversal, and concatenation operation on DREs. We show that, after applying one of these operations only once on two arbitrary DREs, the resulting DRE can be exponentially larger in general.

Parts of the results in Sections 4.2 and 4.3 are joint work with Wim Martens and Matthias Niewerth and are previously published in [LMN12]. Since some of these results were already obtained during the master's thesis of the author, they are marked by an additional reference to [Los10]. Results without a reference are obtained by the author alone and will be published in the long version of [LMN12].

Regular Expressions in the RDF Query Language SPARQL The *Resource Description Framework (RDF)* [CWL14] was developed by the W3C to represent and process data in the *semantic web*. In particular, RDF data is designed to serve as *linked data*. Here, linked data is (unstructured) data that is published in a form that is easy to process automatically and easy to extend by new data [BHBL09]. RDF documents, often denoted as *RDF graphs*, consist of a set of RDF triples which define an edge labeled graph where the labels are specified by global identifiers which are published on the web.

We examine the complexity of regular expressions in the *SPARQL Protocol and RDF Query Language (SPARQL)*[1]. SPARQL is the W3C-recommend query language for RDF data since 2008. Since 2010, regular expressions can be used in SPARQL 1.1 queries in form of *SPARQL property paths*. Our results are based on the January 2012 working draft of SPARQL 1.1 [HS12]. According to the draft, regular expressions in SPARQL queries are defined as regular expressions with additional syntactical operators and semantical constraints. For example, SPARQL regular expressions are allowed to use counting operators (making them exponentially more succinct than standard regular expressions [KT03]) and a limited form of negation. Moreover, the evaluation semantics of SPARQL queries has to fulfill the following constraints:

– the *simple walk requirement* which says that when evaluating a SPARQL regular expression over an RDF graph some subexpressions are allowed to match

[1]SPARQL is a recursive acronym.

only onto simple paths (which is a path without loops), whereas other subexpressions can be matched onto arbitrary paths, and

– the *path counting requirement* which says that the number of different answers to a query has to be counted.

Apart from these non-standard semantics our theoretical investigation was motivated by an experimental study that we conducted in November 2011 on several prevalent SPARQL engines, i.e., the Jena Semantic Web Framework [Apa11], Sesame [KFB08], RDF::Query [Wil], and Corese 3.0 [Cor12]. In this study we discovered that these engines deal with property paths very inefficiently. All of them require double-exponential time for query evaluation in the size of the query. This holds even for very small queries on very small data graphs (with two nodes).

We investigate in **Chapter 5** whether SPARQL query evaluation is still feasible under these constraints. Since SPARQL queries are built over a (possibly) infinite vocabulary consisting of the global label identifiers (called *IRIs*), we adapt our formal definitions of finite automata and regular expressions in Section 5.1. Although our results do not depend on the infinite vocabulary, we choose to model vocabulary and queries accordingly to their formal specification. To investigate the complexity of the previously mentioned constraints independently from each other, we define two different semantics in Section 5.2. First, *simple walk semantics* which is our abstract model of the semantics proposed by the W3C and, second, *regular path semantics* which neglects the simple walk requirement.

For regular path semantics it was already known that regular expressions (without counting operators) can be evaluated over graphs in polynomial time [MW95, AV99, ABE09]. We prove in Section 5.3 that the additional operators (including the counting operators) in SPARQL regular expressions can be evaluated over graphs in polynomial time as well and, therefore, that they do not prevent efficient SPARQL query evaluation under regular path semantics. Opposed to that SPARQL query evaluation is infeasible under simple walk semantics because of the proposed simple walk requirement. More precisely, we show that evaluating SPARQL regular expressions under simple walk semantics is NP-complete even for very restricted classes of regular expressions. Concerning the path counting requirement the situation is even more critical, which we show in Sections 5.4 and 5.5. In these sections we show that counting the number of correct answers to a SPARQL query (using regular expressions) is #P-complete already for very restricted classes of regular expressions (and even under regular path semantics). Under both semantics, the path counting problem is only tractable for (restricted) classes of unambiguous expressions. Finally, we examine the complexity of deciding whether the query result contains a finite number of answers to get a more comprehensive overview of the complexity of the path counting requirement. In this way we provide a fair comparison between the two semantics since a query answer is always finite under simple walk semantics. For SPARQL regular expressions under regular path semantics the corresponding problem can be solved in polynomial time.

In summary, our results show that, opposed to the two additional requirements, the supplementary operators (including the counting operators) in SPARQL regular

expressions do not pose a problem for efficient query evaluation. Furthermore, we show that our results can be extended for *nested regular expressions* which we illustrate in Section 5.6. Nested regular expressions were proposed as an extension for SPARQL queries that is more expressive and still efficient by Pérez et al. [PAG10].

The previously mentioned results are obtained in joint work with Wim Martens and have been previously published in [LM12, LM13]. We remark that similar results were obtained by Arenas et al. [ACP12]. Moreover, the results in [ACP12] and our results were able to provide good arguments to cause some changes in the current SPARQL specification [HS13]. These changes and more developments regarding SPARQL 1.1, which became an official W3C recommendation in 2013, are discussed in Section 5.7.

Enumeration Algorithms for Regular Queries on Dynamic Data To get a better understanding of the challenges arising in query evaluation of today's databases we also investigate query evaluation from a more general perspective. More precisely, we investigate the query evaluation problem for queries that return a set or multiset of answers instead of a boolean answer. Since such answer sets can be extremely large in general, it may be infeasible to compute the set in its entirety before returning the result to the user. Instead algorithms evolved that do not return all answers at the same time but sequentially enumerate the answer set to the user. The corresponding problems are known as *query enumeration* problems and attracted some attention in the last decade (see, e.g., [Bag06, Cou09, DS11, KS13a, KS13b, Seg13]). Another database problem that recently got more attention again is the *incremental maintenance* problem for databases, i.e., efficiently computing the answer to a fixed query over a database that is subject to small updates.

Here, we study algorithms that tackle both of the aforementioned problems. In particular, we study efficient enumeration algorithms for queries defined over data that is likely to change. We denote the corresponding problem as *incremental enumeration* and, to the best of our knowledge, we are the first to consider the problem for queries with arbitrary arity. (For boolean queries the problem was studied by Balmin et al. [BPV04], though in this case only one answer is returned and no enumeration is needed.)

To be able to perform efficient query evaluation under the aforementioned constraints, we examine in **Chapter 6** enumeration algorithms that are sensitive to updates. Our queries are specified by *k-ary nondeterministic finite selecting (tree) automata* (*k-NSFA*s or *k-NSFTA*s, respectively), which we define in Section 6.1 and which are evaluated over words and trees. Since it is known that run-based node-selecting tree automata are as expressive as MSO-queries on trees (see, e.g., [NPTT05, TW68]), we believe that our results for k-NSFTAs are a good yardstick for the complexity of these kind of algorithms. In Section 6.2 we give a formal definition of the considered problems (including the allowed types of updates).

We start investigating the computational complexity of the incremental enumeration problem for words in Section 6.3. For a word of size n, our algorithm solves the problem with a linear time preprocessing in the beginning. Afterwards, it subsequently enumerates all answers with $O(\log n)$ delay between two answers. In the

case an update of the data occurs, the algorithm stops immediately and starts enumerating the new answer set within $O(\log n)$ time. In Section 6.4 we show how to extend this algorithm to trees by adapting a technique from Balmin et al. [BPV04]. In this way we can solve the problem for trees of size n within a linear time preprocessing and $O(\log^2 n)$ enumeration delay such that updates can be processed in $O(\log^2 n)$ time. In the respective sections we explain these algorithms in full detail, prove their correctness, and analyze their complexities in terms of data and query complexity.

Moreover, we show in Section 6.5 that the previously mentioned results can be extended to work for more expressive semantics, i.e., *multiset semantics*. Although we were mainly motivated by the goal to provide a proof of concept for such algorithms, there exists a direct practical application of our results which originates from *Information Extraction* and is illustrated in Section 6.6. In Section 6.7 we discuss possible research questions on incremental enumeration in the future.

The presented results on incremental enumeration are joint work with Wim Martens and large parts of them were previously published in [LM14].

Conclusions can be found in **Chapter 7** where we briefly summarize the presented results and discuss some further questions and open problems.

Most of the results in this dissertation were previously published in [LMN12, LM12, Los12, CDLM13, LM13, LM14].

2

Preliminaries

In this chapter, we review basic notions for regular languages, graphs, XML, and RDF that will be used in the following.

For a finite set S we denote by $|S|$ the cardinality of S and by $\mathcal{P}(S)$ the power set of S. If not mentioned otherwise then S contains every element at most once. We sometimes also need to model multisets where it is allowed that elements can be contained more than once in one set. We define multisets as follows. For a finite set S we define a *multiset M over S* as a function $m_M : S \to \mathbb{N}$. Here, $m_M(a)$ is the *multiplicity* of a in S. We say that $a \in M$ if $m_M(a) > 0$. The size of M, denoted $|M|$, is the sum $\sum_{a \in S} m_M(a)$ of all multiplicities of elements in M. We denote multisets in brackets $\{\!\!\{$ and $\}\!\!\}$. For example, for $M = \{\!\!\{1, 1, 3\}\!\!\}$ we have that $m_M(1) = 2$ and $m_M(3) = 1$. The union $M = M' \cup M''$ (intersection $M = M' \cap M''$) of multisets is defined as usual, taking $m_M(a) = m'_M(a) + m''_M(a)$ ($M(a) = \min\{m'_M(a), m''_M(a)\}$, respectively) for every $a \in S$. We say that $M' \subseteq M''$ if $m'_M(a) \leq m''_M(a)$ for all $a \in S$.

2.1 Representations of Regular Languages

An *alphabet* is a finite and non-empty set of symbols. We always denote an alphabet by Σ and call the elements in the alphabet *(Σ-)symbols*. If Σ contains only one symbol then we call Σ a *unary* alphabet.[1] A *(Σ)-word* w is a finite sequence $a_1 \cdots a_n$ of symbols where, for $i \in \{1, \ldots, n\}$, each $a_i \in \Sigma$. The set of *positions of w* is the set $\{1, \ldots, n\}$ and the *symbol of w at position i* is a_i. By $|w|$ we denote the *length* n of w. The empty word is denoted by ε. A set of words is called a *(word) language*. The language that contains all Σ-words is denoted by Σ^* and is called the *universal language*. By $w_1 \cdot w_2$ or $w_1 w_2$ we denote the *concatenation* of two words w_1 and w_2. For two word languages $L_1, L_2 \subseteq \Sigma^*$, we define their concatenation, denoted

[1] In Chapter 5, we also examine infinite sets of symbols to study the SPARQL query language.

by $L_1 \cdot L_2$, to be the set $\{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$. We abbreviate $L \cdot L \cdots L$ (i times) by L^i. The *reversal* L^R of a language L is the set of words $\{a_n \cdots a_1 \mid a_1 \cdots a_n \in L\}$. There exists various formalisms to represent regular languages, i.e., several types of finite automata and regular expressions.

2.1.1 Definition. A *(nondeterministic) finite automaton* (or *NFA*) N is a tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet, δ is the transition function with signature $Q \times \Sigma \to \mathcal{P}(Q)$, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states.

Whenever $q_2 \in \delta(q_1, a)$ we know that, if N is in state q_1 then reading an $a \in \Sigma$ takes N to state q_2. For readability, we sometimes denote $q_2 \in \delta(q_1, a)$ as $q_1 \xrightarrow{a} q_2 \in \delta$ or as a tuple (q_1, a, q_2). Moreover, we say that the transition $q_2 \in \delta(q_1, a)$ is *q_1-outgoing*, *q_2-incoming*, or *a-labeled*. A *run of N on a word $w = a_1 \cdots a_n$* is a sequence $r = q_0 \cdots q_n$ such that $q_i \in \delta(q_{i-1}, a_i)$ for every $i \in \{1, \ldots, n\}$. For such a run r, we say that r *visits position i in state q_i*, denoted $r(i) = q_i$, for each i. If $q_n \in F$, then the run is *accepting*. A word w is *accepted by N* if there exists an accepting run of N on w. By the *language $L(N)$ of N* we denote the set of all words accepted by N. The class of *regular languages* is the set of all languages L such that there exists an NFA N with $L(N) = L$. We extend δ to words in the canonical way. That is, we define a function δ^* such that $\delta^*(q, a) = \delta(q, a)$ and $\delta^*(q, aw) = \cup_{q' \in \delta(q,a)} \delta^*(q', w)$ for $q, q' \in Q$, $a \in \Sigma$, and $w \in \Sigma^*$. We say that a run r of N on w is a *partial* run if r fulfills all conditions of a run except that r does not have to start with the initial state q_0. A state in an automaton is *useful* if it appears in at least one accepting run. For the remaining chapters, we assume that all states of an automaton are *useful* unless mentioned otherwise.

2.1.2 Definition. An *unambiguous finite automaton* (or a *UFA*) is an NFA N where, for every word $w \in L(N)$, there exists exactly one accepting run of N on w.

2.1.3 Definition. A *deterministic finite automaton* (or *DFA*) is an NFA where δ is a (partial) function with signature $Q \times \Sigma \to Q$.

Notice that every DFA is a UFA but not every UFA is a DFA. All previously defined notions for NFAs transfer to UFAs and DFAs, except that we write $\delta(q_1, a) = q_2$ if there exists an a-labeled transition from a state q_1 to a state q_2 in a DFA. We sometimes abuse notation and denote the minimal DFA with no states by $\emptyset$. For an NFA N we denote the *size* of the NFA by $|N| = \sum_{q,a} |\delta(q, a)|$, i.e., the total number of transitions in N. In the case that N is deterministic we define the size of N as the cardinality of $\{(q, a) \mid \delta(q, a) \text{ is defined}\}$.

2.1.4 Definition. Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA. Then, the *power set automaton of N* is defined as the DFA $\mathcal{P}(N) = (\mathcal{P}(N), \Sigma, \delta_\mathcal{P}, \{q_0\}, F_\mathcal{P})$ where,

- $\delta_\mathcal{P}(S, a) = \bigcup_{q \in S} \delta(q, a)$ for every $S \in \mathcal{P}(Q)$ and $a \in \Sigma$, and
- $F_\mathcal{P} = \{S \in \mathcal{P}(Q) \mid S \cap F \neq \emptyset\}$.

For a regular language L we define the set of *Myhill-Nerode classes of* L as the equivalence classes of the relation $R = \{(v, w) \mid v, w \in \Sigma^* \wedge \forall z \in \Sigma^* : v \cdot z \in L \Leftrightarrow w \cdot z \in L\}$. For an NFA N we use $[N]$ to denote the minimal DFA for $L(N)$. The minimal DFA $[N]$ is unique for $L(N)$ and the number of states in $[N]$ is equal to the number of Myhill-Nerode classes of L.

2.1.5 Definition. The set of *regular expressions (RE)* over Σ is defined as follows:

- $\emptyset$, ε and every Σ-symbol is a regular expression, and
- whenever r and s are regular expressions, then $(r \cdot s)$, $(r + s)$, and (r^*) are also regular expressions.

Without loss of generality we assume that $\emptyset$ is not used as a subexpression of another regular expression. We also call Σ-symbols, ε, and $\emptyset$ *atomic* expressions. For readability, we often omit concatenation operators and parentheses in examples. The language $L(r)$ of a regular expression r is defined as usual:

- $L(\emptyset) = \emptyset$, $L(\varepsilon) = \{\varepsilon\}$, $L(a) = \{a\}$,
- $L(r_1 \cdot r_2) = L(r_1) \cdot L(r_2)$,
- $L(r_1 + r_2) = L(r_1) \cup L(r_2)$, and
- $L(r^*) = \{\varepsilon\} \cup \bigcup_{i=1}^{\infty} L(r)^i$.

In addition, we define the following abbreviations: $L(r?) = \varepsilon + L(r)$ and $L(r^+) = L(r)L(r^*)$. Let r_1 and r_2 be regular expressions. We say that r_1 and r_2 are *equivalent* if $L(r_1) = L(r_2)$. By $|r_1|$ we denote the *size* of r which we define as the number of occurrences of ε, $\emptyset$, Σ-symbols, and operators occurring in r, i.e., the number of nodes in the syntax tree of r. A regular expression r is *minimal* if there does not exist a regular expression r' with $L(r') = L(r)$ and $|r'| < |r|$. However, notice that minimal regular expressions are not unique for a regular language (even up to reordering disjunctions). For example, the expressions $r_1 = a \cdot (b + \varepsilon)$ and $r_2 = a + (a \cdot b)$ are equivalent and of the same size, i.e., $|r_1| = |r_2| = 5$. Furthermore, we define, for a regular language L and symbol $a \in \Sigma$,

- $\text{first}(L) = \{a \in \Sigma \mid \exists\, w \in \Sigma^* : aw \in L\}$,
- $\text{last}(L) = \{a \in \Sigma \mid \exists\, w \in \Sigma^* : wa \in L\}$,
- $\text{followlast}(L) = \{a \in \Sigma \mid \exists\, v \in L, w \in \Sigma^* : vaw \in L\}$, and
- $\text{follow}(L, a) = \{b \in \Sigma \mid \exists\, v, w \in \Sigma^* : vabw \in L\}$.

For example, consider the language $L = L((a + b)^*a)$. Then, $\text{first}(L) = \{a, b\}$, $\text{last}(L) = \{a\}$, $\text{followlast}(L) = \{a, b\}$, and $\text{follow}(L, a) = \text{follow}(L, b) = \{a, b\}$. We analogously define, for a regular expression r and a symbol a, the sets $\text{first}(r)$, $\text{last}(r)$, $\text{followlast}(r)$, and $\text{follow}(r, a)$ according to the language $L(r)$.

2.1.6 Definition. For a regular language L and a word w, the *Brzozowski derivative* $w\backslash L$ is defined as $w\backslash L = \{v \in \Sigma^* \mid wv \in L\}$. For two languages L_1 and L_2 the derivative is defined as $L_2\backslash L_1 = \{v \in \Sigma^* \mid w \in L_2 \wedge wv \in L_1\}$.

For example, we have that $abba\backslash L = L$ for the language $L = L((a+b)^*a)$. Moreover, regular languages are closed under Brzozowski derivatives [Brz64].

2.2 Graphs and Trees

We use graphs basically in two contexts. First, we examine the exact underlying structure of finite automata in terms of graphs, i.e., we model the structure of an automaton as a directed labeled graph where initial and accepting states are projected away. Second, we consider graph-structured input data for query evaluation. For more details on graph theory see, e.g., [Die12].

A *directed graph* G is a tuple (V, E) where V is the set of *nodes* and $E \subseteq V \times V$ is the set of *edges*. An edge in the graph is denoted by an ordered tuple (u, v) where $u, v \in V$. A directed *(edge-)labeled* graph is a tuple (V, E) where the edge relation is a subset of $V \times \Sigma \times V$. An edge e in a labeled directed graph is denoted by an ordered triple (u, a, v). Moreover, we say that the edge e is *a-labeled*. We also consider *undirected* (unlabeled) graphs (V, E) where edges (u, v) are unordered tuples. The *connectivity matrix* of a graph $G = (V, E)$ is a $V \times V$-matrix M where $M[u, v] = 1$ if $u = v$ or $(u, v) \in E$, and $M[u, v] = 0$ otherwise.

For an undirected graph $G = (V, E)$, a *path* from node x to node y in G is a sequence $p = v_0 v_1 \cdots v_n$ such that $v_0 = x$, $v_n = y$, and (v_{i-1}, v_i) is an edge for each $i = 1, \ldots, n$. We say that G is *connected* if there exists a path from x to y for every pair of nodes $x, y \in V$. For a directed labeled graph $G = (V, E)$, a *path* from node x to node y in G is a sequence $p = v_0[a_1]v_1[a_2]v_2 \cdots v_{n-1}[a_n]v_n$ such that $v_0 = x$, $v_n = y$, and (v_{i-1}, a_i, v_i) is an edge for each $i = 1, \ldots, n$. If additionally $v_0 = v_n$ then p is also called a *cycle*. The *Σ-word* induced by the path p in G is $a_1 \cdots a_n$ and is denoted by $\mathrm{lab}^G(p)$. If G is clear from the context then we sometimes also write only $\mathrm{lab}(p)$. When we are not interested in the labels on the edges we write $p = v_0 v_1 \cdots v_n$. The *length* of p is defined as n. Notice that a path of length zero does not follow any edges. If there is a path from node x to node y then we say that y is *reachable* from x. A graph that contains a cycle is called *cyclic*; otherwise the graph is *acyclic*. In Section 5.3.3 we are especially interested in paths or cycles that contain every node at most once.

2.2.1 Definition. Let $n \geq 0$ and $p = v_0[a_1]v_1[a_2]v_2 \cdots v_{n-1}[a_n]v_n$ be a path. Then

- p is a *simple path* if each node v_i for $i = 0, \ldots, n$ in p occurs exactly once,
- p is a *simple cycle* if $n \geq 1$, $v_0 = v_n$ and each node v_i for $i = 1, \ldots, n-1$ in p occurs exactly once, and
- p is a *simple walk* if p is either a simple path or a simple cycle.

We define the *concatenation* of two paths $p_1 = v_0[a_1]v_1 \cdots v_{n-1}[a_n]v_n$ and $p_2 = v_n[a_{n+1}]v_{n_1} \cdots v_{n+m-1}[a_{n+m}]v_{n+m}$ to be the path

$$p_1 \cdot p_2 := v_0[a_1]v_1 \cdots v_{n-1}[a_n]v_n[a_{n+1}]v_{n_1} \cdots v_{n+m-1}[a_{n+m}]v_{n+m}.$$

We consider two paths $p_1 = v_0^1[a_1^1]v_1^1 \cdots [a_n^1]v_n^1$ and $p_2 = v_0^2[a_1^2]v_1^2 \cdots [a_m^2]v_m^2$ in a graph to be *different* when either the sequences of nodes or the sequences of labels are different, i.e., $v_0^1 v_1^1 \cdots v_n^1 \neq v_0^2 v_1^2 \cdots v_m^2$ or $\mathrm{lab}(p_1) \neq \mathrm{lab}(p_2)$. This implies that we consider two paths going through the same sequence of nodes but using different edge labels to be different.

2.2.2 Definition. Let $G = (V, E)$ be a directed graph. A set of nodes $V' \subseteq V$ is *strongly connected* if, for every pair of nodes $x, y \in V'$, x is reachable from y and y is reachable from x. Then, we also say V' is a *strongly connected component of* G. If there exists no set $V'' \subseteq V$ such that V'' is a strongly connected component of G and $V' \subseteq V''$ then V' is *maximal*.

For an NFA N we often refer to N as the graph that is obtained by considering its states as nodes and its transitions as labeled directed edges. Then, we use the notation for graphs analogous for automata. For example, we refer to a path in N or the strongly connected components of N. We also consider paths in a graph that are specified by fixed sets of source and target nodes.

2.2.3 Definition. Let $G = (V, E)$ be a directed labeled graph, $x \in V$, and $Y \subseteq V$. Then, (V, E, x, Y) is the *s-t graph of* G *with respect to* x *and* Y. The node x is called the *source node* and the nodes in Y are called *target nodes* of G. If Y contains only one node y then (V, E, x, y) is an *s-t graph of* G *with respect to* x *and* y.

Moreover, we sometimes leave the facts that x and y are source and target implicit and just refer to (V, E, x, y) as a graph. We recall the following graph-theoretical result which states that the number of arbitrary paths between two nodes in a graph can be counted efficiently.

2.2.4 Theorem ([Ber73]). *Let* (V, E, x, y) *be an s-t graph and let* max *be a number given in binary. Then, the number of paths from* x *to* y *of length at most* max *can be computed in polynomial time in* G *and the number of bits of* max.

The reason why the number of paths can be counted efficiently is due to fast squaring. Using fast squaring one can compute, for a square matrix M, the matrix M^k by performing $O(\log k)$ matrix multiplications. Furthermore, observe that if M is the connectivity matrix of G then $M^k[x, y]$ is the number of paths from x to y of length at most k.

A *tree* t is an undirected, connected, acyclic graph with a unique *root node*. For every tree t we denote the *set of nodes of* t by $\mathrm{Nodes}(t)$ and the *number of nodes* (or the *size*) of t by $|t|$. We call a tree *labeled* if the nodes $v \in \mathrm{Nodes}(t)$ bear a (unique) Σ-label, denoted by $\mathrm{lab}(v)$. Every node in a tree has a finite set of *child-nodes*. A tree where every node has at most two children is called *binary*. If there is no limit

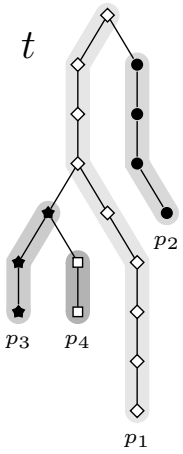


Figure 2.1: A binary tree t with its maximal heavy paths $\mathrm{HPaths}(t)$.

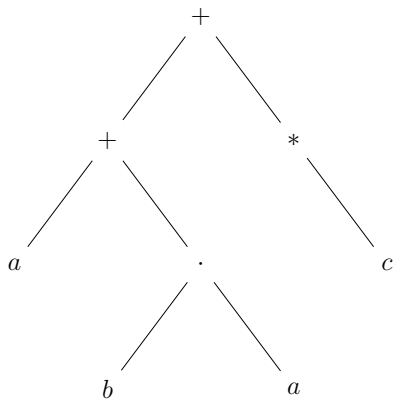


Figure 2.2: Syntax tree for the regular expression $r = a + ba + c^*$.

to the number of children for a node in the tree, we call the tree *unranked*. (Notice that the set of children has to be finite.) Nodes in trees that have no children are called *leaves*. For a tree t and a node v in t, we denote by t_v the subtree of t that is rooted at v.

2.2.5 Definition ([HT84, ST83]). For a node v the *heavy path* $\mathrm{hp}(v)$ of v is defined as follows:

- v belongs to $\mathrm{hp}(v)$, and
- if $v' \in \mathrm{hp}(v)$ has children v_1 and v_2, then v_1 belongs to $\mathrm{hp}(v)$ if $|t_{v_1}| \geq |t_{v_2}|$.

A heavy path of v is *maximal* if it is not included in another heavy path (i.e., if it is not included in the heavy path of v's parent). The *(maximal) heavy path of* t, denoted $\mathrm{hp}(t)$, is the path $\mathrm{hp}(r)$ where r is the root of t. The set $\mathrm{HPaths}(t)$ is the set of all maximal heavy paths of nodes in t.

For a binary tree t, the set $\mathrm{HPaths}(t)$ can be calculated in time and space linear in t [HT84, ST83]. In Figure 2.1, we illustrate the maximal heavy paths for a binary tree t. The set of all maximal heavy paths for the tree t is $\mathrm{HPaths}(t) = \{p_1, p_2, p_3, p_4\}$. Each heavy path is encircled and depicted by a separate shape of nodes in Figure 2.1. Later, we need the following result on the set of all maximal heavy paths for a binary tree.

2.2.6 Lemma ([ST83]). *Let t be a binary tree. The maximum number of distinct maximal heavy paths crossed by any path from the root of t to some leaf is at most* $\log |t|$.

In the following, we mainly use trees to model the abstract structure of input data. However, we also consider another special kind of tree that is closely connected to regular expressions, namely the *syntax tree* of a regular expression. The syntax

tree of a regular expression r represents the abstract syntactical structure of r and, therefore, can be defined by structural induction on the syntax of regular expressions. In Figure 2.2 we illustrate the syntax tree for the expression $r = a + ba + c^*$. Notice that we consider syntax trees to be binary trees. Thus, it is possible that there exist more than one syntax tree for an expression. However, it is irrelevant for our proofs which binary syntax tree is examined. For simplicity, we therefore always refer to *the* syntax tree of an expression in the following.

2.3 Relations and Join Operations

For finite sets $S_1, \ldots, S_k$ we define a *k-ary relation R over domain $S_1 \times \cdots \times S_k$* to be a subset of $S_1 \times \ldots \times S_k$. If $k = 2$ we call the relation *binary*. The *size of a relation R*, denoted by $|R|$, is the number of elements in R. Let S be a finite set and let R_1 and R_2 be binary relations over the signature $S \times S$. We define the *natural join of R_1 and R_2* to be the set

$$R_1 \bowtie R_2 = \{(u, v) \mid \exists z \in S : (u, z) \in R_1 \wedge (z, v) \in R_2\}.$$

The next lemma states the cost of a single join using a rather naïve algorithm.

2.3.1 Lemma ([RG02]). *Let $R_1, R_2 \subseteq \{1, \ldots, n\}^2$ be binary relations for some $n \in \mathbb{N}$. The relation $R_1 \bowtie R_2$ can be computed in time $O(n^3)$.*

Proof. When one represents R_1 and R_2 as boolean $n \times n$ matrices M_1 and M_2, the matrix representation for $R_1 \bowtie R_2$ can be obtained by multiplying M_1 with M_2, costing time $O(n^3)$.

The above result suffices for our purposes when we aim to show that there is a polynomial-time algorithm for a problem that involves joins. However, there are other join algorithms that are more efficient in certain practical instances. For example, we will use the following result on joining two relations later.

2.3.2 Lemma. *Let $R_1, R_2 \subseteq \{1, \ldots, n\}^2$ be binary relations for some $n \in \mathbb{N}$. The relation $R_1 \bowtie R_2$ can be computed in time $O(|R_1| \log |R_1| + |R_2| \log |R_2| + j \log |R_1 \bowtie R_2|)$ where $j = |\{(u, z, v) \mid (u, z) \in R_1$ and $(z, v) \in R_2\}|$.*

Proof. The result is obtained using a variant of the sort-merge join algorithm (see, e.g., [RG02]). Therefore, we start by sorting R_1 and R_2 on their respective join attributes which can be done in time $O(|R_1| \log |R_1| + |R_2| \log |R_2|)$. Once the relations are sorted, the algorithm proceeds similar to the sort-merge join. This means that, the algorithm determines the tuples in the result using two interleaved iterations on the relations. Since the output shall be a set of tuples rather than a multiset of tuples, the algorithm has to eliminate duplicates. In order to do this, it maintains the set of tuples that are already discovered to be in $R_1 \bowtie R_2$ in a self-balancing binary search tree (e.g., an AVL tree [SW11]). When a new candidate tuple for $R_1 \bowtie R_2$ is found, the algorithm can always determine in time $O(\log |R_1 \bowtie R_2|)$ whether the tuple has bee found before or not. Since there are j candidate tuples, this last step costs time $O(|R_1| + |R_2| + j \log |R_1 \bowtie R_2|)$ in total. □

In the worst case, the number j in Lemma 2.3.2 could be n^3. However, it is expected that j is usually small in practice and the literature emphasizes that the worst case is very unlikely [RG02].

We next review the complexity of applying the natural join on a relation more than once. For a binary relation $R \subseteq \{1, \ldots, n\}^2$ and $k \in \mathbb{N}\backslash\{0\}$, we define $R^k := R$ if $k = 1$ and $R^k := R \bowtie R^{k-1}$ otherwise. To efficiently compute R^k we use the following method known as *fast squaring*:

$$R^k = \begin{cases} R, & \text{if } k = 1, \\ R \bowtie \left(R^{\frac{k-1}{2}} \bowtie R^{\frac{k-1}{2}}\right), & \text{if } k \text{ is odd}, \\ \left(R^{\frac{k}{2}} \bowtie R^{\frac{k}{2}}\right), & \text{if } k \text{ is even}. \end{cases}$$

Using this method the following result can be obtained.

2.3.3 Lemma (see, e.g., [Ber73]). *Let R be a binary relation and $k \in \mathbb{N}$. Then, the relation R^k can be computed by performing $O(\log k)$ joins.*

2.4 Basic Complexity Results

In the following we use the complexity classes in Figure 2.3. Known inclusions for these classes are depicted by edges from bottom to top, i.e., lower classes are included in the higher ones in the figure. We assume familiarity with the mentioned classes and reductions. For a more extensive overview of complexity theory see, e.g., [AB09, Pap94].

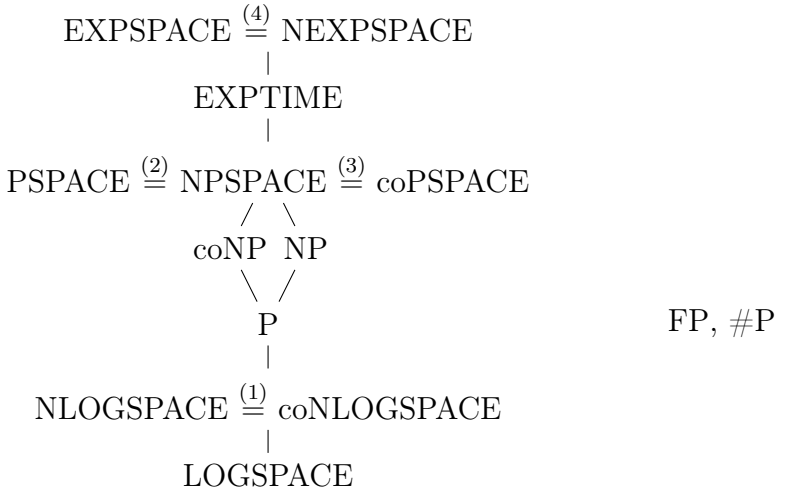


Figure 2.3: Summary of complexity classes.

In 1970, the equivalences (2) PSPACE = NPSPACE and (4) EXPSPACE = NEXPSPACE were proved. They are a consequence of Savitch's Theorem [Sav70]. Then, the equivalences (1) NLOGSPACE = coNLOGSPACE and (3) PSPACE = coPSPACE are due to the Immerman-Szelepcsényi Theorem [Imm88, Sze88].

Later, we also use that the Reachability problem is known to be NLOGSPACE-complete.

Problem (Reachability).
Given: A directed acyclic graph $G = (V, E)$ and $x, y \in V$.
Question: Is y reachable from x by a path?

2.4.1 Theorem ([Jon75]). Reachability *is NLOGSPACE-complete.*

Moreover, we consider the classes FP and #P which are defined as follows. The class *Function polynomial time (FP)* generalizes the definition of the class P from decision problems to functions.

2.4.2 Definition (see, e.g., [AB09]). The class *Function Polynomial Time (FP)* contains all functions f such that, for every x, the value $f(x)$ can be computed in deterministic polynomial time.

The class *Sharp-P (#P)* is defined as the set of counting problems for decision problems in NP.

2.4.3 Definition (see, e.g., [AB09]). The class *Sharp-P (#P)* contains all functions f such that, for a fixed NP Turing machine M, the value of $f(x)$ equals the number of accepting runs of M when given x as input.

The following problems are #P-complete under Cook reductions.

Problem (#SAT).
Given: A boolean formula φ in conjunctive normal form.
Question: How many different assignments satisfy φ?

Problem (#DNF).
Given: A boolean formula φ in disjunctive normal form.
Question: How many different assignments satisfy φ?

Problem (#SimplePaths).
Given: An s-t graph $G = (V, E, x, y)$.
Question: How many different simple paths from x to y exist in G?

2.4.4 Theorem ([Val79b]). *The problems* #SAT, #DNF*, and* #SimplePaths *are #P-complete.*

By the definition of #P it is not very surprising to see that the problem #SAT is #P-complete. However, there also exist #P-complete problems that correspond to decision problems in P see, e.g., #DNF. Already in 1979, Valiant showed the existence of this strange behavior by proving that counting the number of perfect matchings in a bipartite graph is #P-complete [Val79a]. Moreover, we remark that we always use polynomial time Turing reductions (also known as Cook reductions) to show completeness for the class #P.

For the class NP there exist many typical NP-complete problems [Kar72]. We will use the following problem in our proofs.

Problem (EvenSimplePath).
Given: An s-t graph $G = (V, E, x, y)$.
Question: Is there a simple path from x to y of even length?

2.4.5 Theorem ([LP84, MW95]). EvenSimplePath *is NP-complete.*

Regarding Theorem 2.4.5, Lapaugh and Papadimitriou proved that it is NP-complete to compute whether there exists a simple path of even length from x to y. Later, Mendelzon and Wood proved that it is NP-complete to compute whether there is a simple path from x to y such that the induced word of the path is in the language $L((aa)^*)$. Despite the fact that the result was first proved in [LP84] we also refer to [MW95] because their work is closely related to ours.

We also consider the following language theoretic problems.

Problem (Membership).
Given: A regular expression r and a word w.
Question: Is $w \in L(r)$?

Problem (Universality).
Given: A regular expression r.
Question: Is $L(r) = L(\Sigma^*)$?

2.4.6 Theorem ([MS72]). Universality *is PSPACE-complete.*

2.5 Data, Schemas, and Queries

Next, we review basics on the *Extensible Markup Language (XML)* [BPSM+08] and the *Resource Description Framework (RDF)* [CWL14]. In Section 2.5.1 our focus is on XML schema languages which are used to constrain the format of XML documents. Regarding the RDF data format, we focus on the use of regular expressions in RDF query languages. In Section 2.5.2 we review basics on RDF and the *SPARQL Protocol and RDF Query Language (SPARQL)* [HS12].

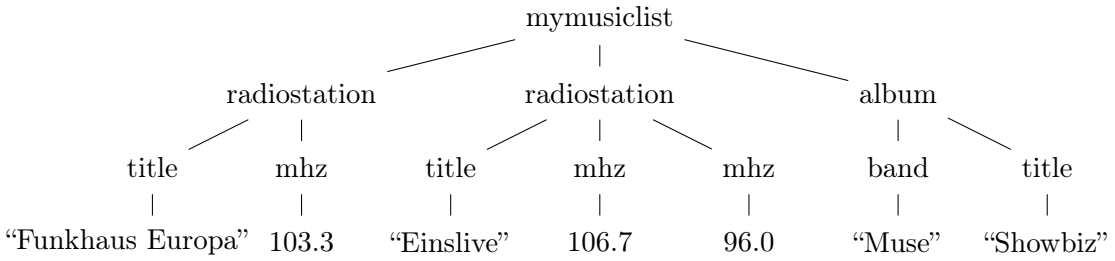
2.5.1 XML and XML Schema Languages

When processing data on the web the *Extensible Markup Language (XML)* is the W3C-recommended format for semi-structured data. The language is used to structure data by defining individualized tags which can be arranged into a tree-like form in XML documents. For our purposes it is sufficient to model XML documents as finite unranked labeled trees over a finite alphabet. For a more comprehensive overview on XML see, e.g., Abiteboul et al. [ABS99].

2.5.1 Example. In Figure 2.4(a) we depict an example of an abstract XML document which stores information about a music bibliography. The bibliography `mymusiclist` contains two elements of the type `radiostation` and one element of the type `album`. Each element is enclosed by an opening and closing tag and can

```
<mymusiclist>
   <radiostation>
      <title> "Funkhaus Europa" </title>
      <mhz> 103.3 </mhz>
   </radiostation>
   <radiostation>
      <title> "Einslive" </title>
      <mhz> 106.7 </mhz>
      <mhz> 96.0 </mhz>
   </radiostation>
   <album>
      <band> "Muse" </band>
      <title> "Showbiz" </title>
   </album>
</mymusiclist>
```

(a) Example for an XML document.



(b) The representation of the XML document in (a) as a tree.

Figure 2.4: An XML document and its abstract representation as a tree.

contain other elements. For example, elements of the type `mhz` are contained in the `radio` elements. Moreover, elements can contain values. For example, 103.3 is the value of the first `mhz` element in the XML document. In Figure 2.4(b) we illustrate the abstract tree model of the XML document from Figure 2.4(a). For simplicity, we do not distinguish between nodes that are labeled by an element name and nodes that bear a value as a label.

The allowed alphabet and the structure of the tree can be controlled by a *schema* which is specified in an *XML schema language*. Examples for XML schema languages are *Document Type Definitions (DTD)* [BPSM$^+$08], *XML Schema (XSD)* [FW04], and *Relax NG* [CM01]. The information that is stored in a schema can be highly advantageous when managing, exchanging, and querying XML data. It is known that the presence of schema information is crucial for automatic error detection in the data (which is called validation in, e.g., [BPSM$^+$08, SV02, BPV04, KM13]) and in data transformation [MSV03, MN05, MBPS05]. Schemas provide information for query optimization, processing [NS06, Woo03], and data integration [ABLM14]. Moreover, they provide a high-level overview of the structure of the data for the user. The complexity of using regular word or regular tree languages in schemas was investigated in [MN07, MNS07, MNS09, GIM$^+$13].

In the following, we consider schemas that, in their core, specify the structure of well-formed XML documents through a set of constraints that are very similar to

extended context-free grammar productions. Such constraints are usually denoted as a set of rules of the form

$$type \rightarrow content$$

where *content* is a regular expression defining the allowed content inside the element specified by *type*. For this reason, regular expressions are pivotal for schema languages.

2.5.2 Example. Next, we illustrate a possible schema for the XML document in Figure 2.4(a). For our purposes it is sufficient to represent the schema as an extended context-free grammar:

$$\begin{array}{lll} \text{mymusiclist} & \rightarrow & \text{radiostation}^* \ \text{album}^* \\ \text{radiostation} & \rightarrow & \text{title mhz}^+ \\ \text{album} & \rightarrow & (\text{band} + \text{artist}) \ \text{title year?} \end{array}$$

The schema specifies that an element *mymusiclist* contains a (possible empty) list of elements *radiostation* and, subsequently, a (possible empty) list of elements *album*. A *radiostation* has a *title* and one or more frequencies specified by elements *mhz*. Each *album* is associated with a *band* or an *artist*, a *title*, and, optionally, a *year*.

The W3C specifications for XML Schema and DTDs do not allow arbitrary regular expressions to define *content*. Instead, they require that all regular expressions in the schema are *deterministic* in some way. (We formalize this notion in Chapter 3.) This semantical constraint is contained in the specifications due to a requirement in the ISO standard for the *Standard Generalized Markup Language (SGML)* where it was introduced to ensure efficient parsing in the first place.

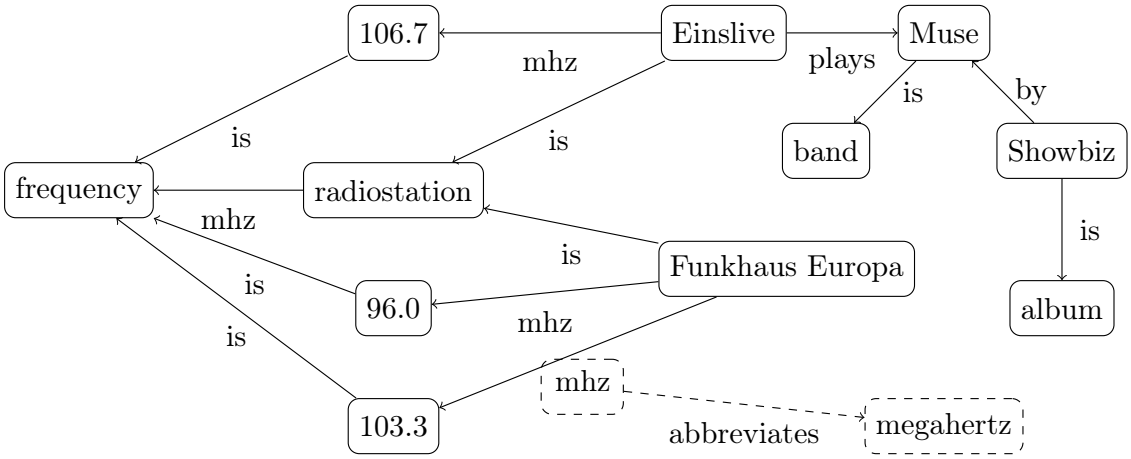
2.5.2 Linked Data, RDF, and SPARQL

Since the turn of the new millennium more and more data on the web is specified in the *Resource Description Framework (RDF)* [AP11]. In fact, nowadays, RDF is the de-facto standard for *linked data*. Here, linked data is the generic term for the idea to publish data such that it is accessible for automatic processing. More precisely, linked data on the web has the purpose to improve the way in which data on the web is readable by computers and to enable new ways of querying web data. Resources on the web are thereby specified via a unique name, i.e., a so-called *Uniform Resource Identifier (URI)* [BLFM05]. For example, every URL is a URI. Tim Berners-Lee published the following four principles that characterize linked data in general [BL09]:

- Use URIs as names for things.
- Use HTTP URIs so that people can look up those names.
- When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
- Include links to other URIs, so that they can discover more things.

```
(radiostation,mhz,frequency),          (Einslive,is,radiostation),
(frequency,is,96.0),                   (Einslive,mhz,106.7),
(frequency,is,103.3),                  (Einslive,plays,Muse),
(frequency,is,106.7),                  (Muse,is,band),
(Funkhaus Europa,is,radiostation),     (Showbiz,by,Muse),
(Funkhaus Europa,mhz,96.0),            (Showbiz,is,album),
(Funkhaus Europa,mhz,103.3),           (mhz,abbreviates,megahertz)
```

(a) The set of RDF triples specifying the graph in (a).



(b) An example RDF graph.

Figure 2.5: An RDF graph and its triple representation.

An *HTTP URI* is a URI that starts with the prefix *http* addressing a website which is readable by humans. However, in the RDF standard it is even allowed to specify resources by IRIs instead of URIs. An *Internationalized Resource Identifier (IRI)* is a generalization of URIs that allows for a wider range of words [DS05]. Thus, every URI is also an IRI but not every IRI is an URI. Moreover, we remark that no limit on the number of characters in a URI or IRI is given in the standards such that, in theory, the sets of URIs and IRIs are infinite. The exact definition of URIs and IRIs is, however, irrelevant for our purposes.

Next, we define our abstract model of RDF data. In its core, RDF data consists of a set of *RDF triples* which are of the form (*subject, predicate, object*) where, for simplicity, we assume that the components *subject*, *predicate*, and *object* range over an infinite set of IRIs. (Notice that, according to the RDF specification, subject and object can also be replaced by variables.) Furthermore, predicates can be used as subject or object of other triples. Our abstract model of this set of triples is a directed labeled graph, called *RDF graph*.

2.5.3 Example. Figure 2.5(a) contains a set of example RDF triples. The RDF graph that can be retrieved from the triples in (a) is illustrated in Figure 2.5(b). The edge (*mhz, abbreviates, megahertz*) is consistent with the RDF standard since it is allowed that the predicate *mhz* of the edge (*Funkhaus Europa, mhz, 103.3*) is the subject of the edge (*mhz, abbreviates, megahertz*).

For our purposes it is sufficient to model subjects and objects of an RDF triple as nodes in a graph and predicates of an RDF triple as edge labels. Other kind of edges (RDF triples, respectively) can always be modelled by additional nodes in the graph (see [PAG10] for details). For this reason, we do not explicitly consider edges that contain a predicate of another edge as a subject or object in the following.

The W3C-recommended language for querying RDF data is the *SPARQL Protocol and RDF Query Language (SPARQL)*. In SPARQL queries it is allowed to use regular expressions since the *property path feature* was released in the SPARQL 1.1 specification in 2010. The property path feature extends the navigational capabilities of the query language in the following way. For example, a SPARQL query of the form

`SELECT ?x,?y WHERE {?x r ?y}`,

where r is a regular expression, asks for all tuples (x, y) such that there is a path from x to y that is labeled with a word in the language $L(r)$. The regular expressions that are allowed as property paths in SPARQL queries are enhanced with additional operators that allow counting and even a restricted form of negation. Moreover, the evaluation semantics for these regular expressions are defined in a non-standard way requiring that parts of the expression have to be matched against simple paths in the graph and that the output of the above query is, according to the W3C's semantics, in fact a multiset that contains each tuple (x, y) as often as the number of paths from x to y that are labeled with a word in $L(r)$. By the requirement that certain subexpressions are matched only against simple paths, the W3C specification therefore ensures that such an answer is always finite. We study the complexity of evaluating SPARQL queries under these semantics in Chapter 5. A formal definition of these semantics can be found in Section 5.2. A collection of more detailed examples on SPARQL 1.1 queries can be found in [HS12].

3

Regular Expressions in the Context of RDF and XML

In this dissertation we study regular expressions enhanced by various additional features that are used in the context of RDF and XML. In the following we summarize the classes of regular expressions under consideration and review useful results concerning these classes. In Section 3.1, we first define classes of regular expressions that contain additional operators as syntactic sugar, e.g., counting operators, negations and wildcards. Then, in Section 3.2, we review the definition of chain regular expressions, which is a syntactically very restricted class of expressions that aims to be expressive enough in practice but tractable in general. In Section 3.3, we define deterministic regular expressions which are specified by a semantical constraint that specifies determinism for regular expressions similar as for finite automata.

3.1 Regular Expressions with Additional Operators

We define the following four additional syntactical features for regular expressions. First, we define regular expressions with *counting operators* which allow to write expressions that are exponentially more succinct in general [KT03, Gel10]. For example, the expression $a^{10,100}$ describes the language of all a-words with length at least 10 and at most 100. Moreover, it is known that testing membership for regular expressions with counting operators can be done in polynomial time [KT03] and, later, we show that these expressions can be evaluated over graphs in polynomial time as well. We are motivated to examine counting operators because counting operators are used to define expressions in XML Schema [GSMT$^+$12, GMN09, CGS09a, CGS09b] and because they are useful when querying RDF data [HS12, LM13].

Afterwards, we define two negation operators: the full-fledged *negation* ($\neg$) and a *restricted negation* (!). The motivation for the !-operator comes from the SPARQL query language where it is defined as a shortcut for a forbidden subset of symbols. At last, we define a *wildcard* symbol ($\bullet$) which can be used to specify an arbitrary symbol ($\neq \varepsilon$). The $\bullet$-symbol is allowed in SPARQL property paths.

3.1.1 Definition. Let r be an RE. The following operators are defined for the class of regular expressions:

operator	token	syntax
Counting Operators	$\#$	If $k \in \mathbb{N}$ and $\ell \in \mathbb{N} \setminus \{0\} \cup \{\infty\}$ with $k \leq \ell$ then $(r^{k,\ell})$ is a regular expression.
Negation	$\neg$	If r is a regular expression then so is $(\neg r)$.
Negated label test	!	If $\{a_1, \ldots, a_n\}$ is a non-empty, (finite) subset of Σ then $!(a_1 + \cdots + a_n)$ is a regular expression.
Wildcard	$\bullet$	The symbol $\bullet$ (where $\bullet \notin \Sigma$) is a regular expression.

To avoid confusion on the set of allowed operators for a class of regular expressions, we refer to RE for the class of (standard) regular expressions over the operators $+$, $\cdot$, and * (see, e.g., Definition 2.1.5). For the aforementioned additional operators, we refer to RE($\mathcal{X}$) for the class of regular expressions with additional features $\mathcal{X} \subseteq \{\#, \neg, \bullet, !\}$. For example, RE($\#, \neg$) denotes the set of regular expressions with counting operators and negation.

Throughout the document we consider $!(a_1 + \cdots + a_n)$ as an atomic expression. For readability, we sometimes abbreviate $r^{k,k}$ by r^k. The language $L(r)$ of a regular expression r with additional features from the set $\{\#, \neg, \bullet, !\}$ is defined by extending the semantics of REs as follows:

- $L(r^{k,\ell}) = \bigcup_{i=k}^{\ell} L(r)^i$,
- $L(\neg r) = \Sigma^* \setminus L(r)$,
- $L(!(a_1 + \cdots + a_n)) = \Sigma \setminus \{a_1, \ldots, a_n\}$, and
- $L(\bullet) = \Sigma$.

The *size* $|r|$ of r is defined analogously to Section 2.1, i.e., as the number of nodes in the syntax tree of r with additional occurrences of (k, ℓ), $\neg$, !, and $\bullet$ as nodes in the tree. Additionally, if the expression contains a counting operator k, ℓ then we add the sizes of the binary representations for every number k and ℓ to the size of r. In this case, a number $k \in \mathbb{N}$ has size $\lceil \log k \rceil$ if $k > 0$ and size one if $k = 0$.

The definition of the negated label test expression (!) seems to be quite uninteresting in the first place. Later, we also model expressions over an infinite set of symbols to cope with the infinite set of IRIs in the SPARQL specification. In these expressions it is more functional to use the negated label test.

Expressions in the class RE($\neg$) are also often called *generalized regular expressions* in the literature. Any class of expressions that does not use the Kleene star operator (*) is also called *star-free*.

3.2 Chain Regular Expressions

When a problem is intractable for the class of regular expressions, we study *chain regular expressions* in order to trace the tractability frontier of the problem. These expressions have already been used to trace the tractability frontier for the regular expression containment problem [MNS04, MNS09] and it has been shown that they can be efficiently learned [BNV07, BGNV10, BNSV10, FR13], which is especially favorable in practical applications. However, they were first examined as a model for expressions that are frequently used in practice [Cho02, BNdB04, MNS04]. The following definition of chain regular expressions comes from [MNS09].

3.2.1 Definition. A *base symbol* is a regular expression w, w^*, w^+, or $w?$ where w is a non-empty word; a *factor* is of the form e, e^*, e^+, or $e?$ where e is a disjunction of base symbols of the same kind. That is, e is of the form $(w_1+\cdots+w_n)$, $(w_1^*+\cdots+w_n^*)$, $(w_1^+ + \cdots + w_n^+)$, or $(w_1? + \cdots + w_n?)$ where $n \geq 0$ and $w_1, \ldots, w_n$ are non-empty words. A *chain regular expression (CHARE)* is $\emptyset$, ε, or a concatenation of factors.

We use the same shorthand notation for CHAREs as in [MNS09]. The shorthands we use for the different kind of factors are illustrated in Table 3.1. We denote by a and a_i arbitrary symbols in Σ and by w, w_i non-empty words in Σ^+.

Factor	Abbr.
a	a
a^*	a^*
a^+	a^+
$a?$	$a?$
w^*	w^*
w^+	w^+
$w?$	$w?$

Factor	Abbr.
$(a_1 + \cdots + a_n)$	$(+a)$
$(a_1 + \cdots + a_n)^*$	$(+a)^*$
$(a_1 + \cdots + a_n)^+$	$(+a)^+$
$(a_1 + \cdots + a_n)?$	$(+a)?$
$(a_1^* + \cdots + a_n^*)$	$(+a^*)$
$(a_1^+ + \cdots + a_n^+)$	$(+a^+)$

Factor	Abbr.
$(w_1 + \cdots + w_n)$	$(+w)$
$(w_1 + \cdots + w_n)^*$	$(+w)^*$
$(w_1 + \cdots + w_n)^+$	$(+w)^+$
$(w_1 + \cdots + w_n)?$	$(+w)?$
$(w_1^* + \cdots + w_n^*)$	$(+w^*)$
$(w_1^+ + \cdots + w_n^+)$	$(+w^+)$

Table 3.1: Possible factors in chain regular expressions and how they are denoted [Mar06].

For example, the regular expression $((abc)^*+b^*)(a+b)?(ab)^+(ac+b)^*$ is a CHARE with factors of the form $(+w^*)$, $(+a)?$, w^+, and $(+w)^*$ from left to right. The expression $(a+b)+(a^*b^*)$, however, is not a CHARE due to the nested disjunction. Notice that each kind of factor that is *not* listed in Table 3.1 can be simulated through one of the other ones. For example, a factor of the form $(a_1^+ + \cdots + a_n^+)?$ is equivalent to $(a_1^* + \cdots + a_n^*)$. For a similar reason no factor of the form w is listed.

We refer to CHARE($\mathcal{X}$) for the fragment of the class of chain regular expressions where only factors $\mathcal{X}$ are allowed. For example, the above mentioned expression is a CHARE($(+w^*)$, $(+a)?$, w^+, $(+w)^*$).

3.3 Deterministic Regular Expressions

For regular expressions the concept of determinism was introduced to supplement the ISO standard of SGML. Since the schema languages XML Schema (see, e.g., the *Unique Particle Attribution* (*UPA*)) and Document Type Definitions basically derive from SGML, their specifications also require that regular expressions which are used in XSDs and DTDs are deterministic. The theoretical foundations of these so-called *deterministic regular expressions* (or *DREs*) were developed in a seminal paper by Brüggemann-Klein and Wood [BKW98]. Since then DREs have been studied in the context of language approximations [BGMN09], learning [BGNV10], descriptional complexity [GN12, LMN12] and static analysis [CC08, CGS09b].

In the following, we review the basics of deterministic regular expressions, which are sometimes also called *one-unambiguous* expressions. Before we define them formally, we introduce the notion of *unambiguous* regular expressions to stress the differences between these two notions. In Section 3.3.1 we outline the decision procedure of Brüggemann-Klein and Wood that decides whether the language of a given minimal DFA can be defined by a deterministic regular expression. Since not every regular language can be defined by a deterministic expression, we consider the closure property for several relevant operations for languages that are definable by a deterministic expression in Section 3.3.2. In Section 3.3.3 we extend the notion of determinism for expressions using the restricted negation operator ! and a wildcard symbol $\bullet$.

For an RE r we define the *annotated expression of* r, denoted $r_\#$, as the expression r where every symbol $a \in \Sigma$ of r is replaced by the symbol a_i if a is the ith symbol of r when reading from left to right. For example, for $r = (a+b)^*a(a+b)^*$ it holds that $r_\# = (a_1 + b_2)^*a_3(a_4 + b_5)^*$. For a word $w \in L(r_\#)$ we denote by $w^\natural$ the word without the subscript numbers. Notice that we have that $w^\natural \in L(r)$ for every $w \in L(r_\#)$.

3.3.1 Definition. An RE r is *unambiguous* if there are no words $w \in L(r)$ and $u, v \in L(r_\#)$ such that $u \neq v$ but $u^\natural = v^\natural = w$.

In other words, for every word in the language of an unambiguous expression, there is one unique way how to "match" the word in the expression. For example, the expression $(ba + b)(b + ab)$ is not unambiguous since the word bab can be matched in two different ways, i.e., the words $b_1a_2b_4$ and $b_3a_5b_6$ are both in the language $L((b_1a_2 + b_3)(b_4 + a_5b_6))$. The equivalent expression $bab + baab + bb$ is unambiguous.

Every regular language can be defined by an unambiguous regular expression. Given a regular language L and an RE r with $L(r) = L$, one can obtain an unambiguous expression for L by converting r into a deterministic finite automaton and converting this automaton back into an (unambiguous) expression using the standard conversion algorithm for regular expressions (see, e.g., [HMU13] for details).

For a deterministic regular expression (also called *one-unambiguous* expression) the following holds: When reading a word from left to right every single symbol can be uniquely matched in the expression without looking ahead in the word.

3.3.2 Definition. An RE r is *deterministic* (or a *DRE*) if there are no words va_iw and va_jw' in $L(r_\#)$ with $v, w, w' \in \Sigma^*$, $a \in \Sigma$ and $i \neq j$.

Observe that every deterministic regular expression is unambiguous but not every unambiguous regular expression is deterministic. For example, the unambiguous expression $bab{+}baab{+}bb$ from above is not deterministic (or one-unambiguous) since when reading a word beginning with b we do not know to which b in the expression it should be matched (without looking ahead in the word). More precisely, the words $b_1a_2b_3$ and b_8b_9 are both in the language $L(b_1a_2b_3 + b_4a_5a_6b_7 + b_8b_9)$. An equivalent DRE for this language is $b(a(b + ab) + b)$.

However, not every regular language has an equivalent DRE, i.e., the language $L((a{+}b)^*a(a{+}b))$ is not DRE-definable [BKW98]. We say that a regular language L is *DRE-definable* if L can be defined by a DRE.

3.3.3 Theorem ([BKW98]). *The set of DRE-definable languages is a strict subclass of the set of regular languages.*

Moreover, it is known that every finite language is DRE-definable [BKW98]. Since every DRE is an RE we reuse notations introduced for regular expressions also for DREs. In particular, we define *minimal DREs* analogously for regular expressions and remark that minimal DREs are (as well as REs) not unique up to reordering of disjunctions. Take, for example, the deterministic regular expressions $(a + \varepsilon)(c + d) + b(c + \varepsilon) + \varepsilon$ and $a(c + d) + (b + \varepsilon)(c + \varepsilon) + d$.

In the remainder, we also use the following Kleene-like definition of DRE-definable languages to argue whether regular languages are DRE-definable or not.

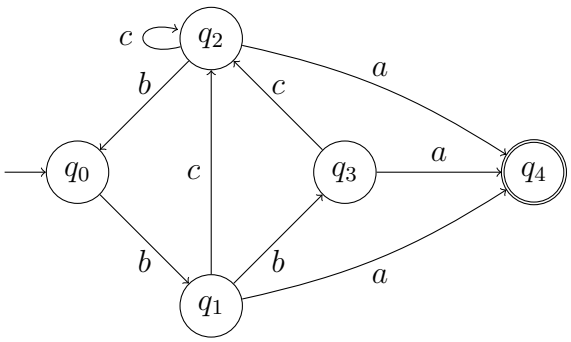
3.3.4 Theorem ([BKW98]). *Let Σ be an alphabet. The set of DRE-definable languages over Σ is the smallest set $\mathcal{L}$ of languages that satisfies the following languages:*

- *$\emptyset$, ε, and $\{a\}$, for each $a \in \Sigma$, are in $\mathcal{L}$.*
- *If $L_1, L_2 \in \mathcal{L}$ and $\text{first}(L_1) \cap \text{first}(L_2) = \emptyset$ then $L_1 \cup L_2 \in \mathcal{L}$.*
- *If $L_1, L_2 \in \mathcal{L}$, $\varepsilon \notin L_1$, and $\text{followlast}(L_1) \cap \text{first}(L_2) = \emptyset$ then $L_1 \cdot L_2 \in \mathcal{L}$.*
- *If $L_1 \in \mathcal{L}$ then $L_1 \backslash \{\varepsilon\} \in \mathcal{L}$.*
- *If $L_1 \in \mathcal{L}$ and $\text{followlast}(L_1) \cap \text{first}(L_1) = \emptyset$ then $(L_1)^* \in \mathcal{L}$.*

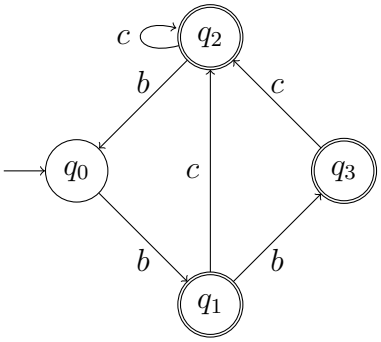
3.3.1 Recognizing DRE-definable Languages

Although it can be checked in linear time whether a regular expression is deterministic [GMS12], we will see in Chapter 4 that it is computationally harder to check whether a regular language (given as an RE, NFA, RE(#), or minimal DFA) is DRE-definable. To investigate this problem we need the theoretical basics of the class of DRE-definable languages which basically come from [BKW98][1]. In this paper, the

[1]For the interested reader this paper is the designated introduction to deterministic regular languages.



(a) An NFA with two orbits $\{q_0, q_1, q_2, q_3\}$ and $\{q_4\}$ and four gates $q_1, \ldots, q_4$.



(b) The orbit automaton N_{q_0} with an N_{q_0}-consistent symbol c.

Figure 3.1: An NFA N and the orbit automaton N_{q_0}.

main contribution is a characterization of the class of DRE-definable languages in terms of structural properties of finite automata. Using this characterization it was proved that it is decidable whether a given regular language is definable by a DRE. We review this characterization of DRE-definable languages next. To this end, we need the following terminology which originates from [BKW98].

3.3.5 Definition. Let N be an NFA and q a state in N.

- The *orbit of* q, denoted $\mathcal{O}(q)$, is the maximal strongly connected component of N that contains q. If q is the only state in $\mathcal{O}(q)$ and q has no self-loops then the orbit $\mathcal{O}(q)$ is called *trivial*.
- The state q is called a *gate* of $\mathcal{O}(q)$ if q is accepting or q has an outgoing transition (q, a, q') such that $q' \notin \mathcal{O}(q)$.

In Figure 3.1(a) we illustrate an NFA with two orbits $\{q_0, q_1, q_2, q_3\}$ and $\{q_4\}$. The states q_1, q_2, and q_3 are gates of the first orbit and the state q_4 is a gate of the trivial orbit $\{q_4\}$. Using the notion of orbits and gates we define the *orbit property*, which is central for testing DRE-definability.

3.3.6 Definition. Let N be an NFA. Then N has the *orbit property* if the following holds for every pair of gates q_1, q_2 in the same orbit:

(1) q_1 is accepting if and only if q_2 is accepting, and

(2) $q \in \delta(q_1, a)$ if and only if $q \in \delta(q_2, a)$ for every state $q \notin \mathcal{O}(q_1)$.

We say that a transition (q_1, a, q_2) is an *inter-orbit transition* if $\mathcal{O}(q_1) \neq \mathcal{O}(q_2)$, i.e., q_1 and q_2 belong to different orbits. For an NFA N the *orbit automaton of a state* q, denoted N_q, is obtained from N by removing all states that are not in $\mathcal{O}(q)$, fixing q as initial state, and the gates of $\mathcal{O}(q)$ as accepting states. The language $L(N_q)$ is called *orbit language of* q. In Figure 3.1(b) we illustrate the orbit automaton N_{q_0} of N. The following theorem characterizes the class of DRE-definable languages.

3.3.7 Theorem ([BKW98]). *Let D be a minimal DFA. The language $L(D)$ is DRE-definable if and only if D has the orbit property and every orbit language of D is DRE-definable.*

Although the last theorem gives a precise characterization of DRE-definable languages, the result is not sufficient to construct a recursive decision procedure for testing whether a language is DRE-definable. Since we cannot test whether the language of an automaton that consists of exactly one orbit is DRE-definable, we also need the following.

Let D be a DFA. A symbol $a \in \Sigma$ is called *D-consistent* if there exists a state $c(a)$ such that every accepting state q of D has a transition $(q, a, c(a))$. For example, the symbol c is N_{q_0}-consistent in the automaton of Figure 3.1(b). We sometimes also call the transition $(q, a, c(a))$ *D-consistent*. For a set S of D-consistent symbols the *S-cut* of D, denoted by D_S, is the automaton that is obtained from D by removing the transitions $(q, a, c(a))$ for every accepting state q and symbol $a \in S$. The following theorem provides another characterization of DRE-definable languages.

3.3.8 Theorem ([BKW98]). *Let D be a minimal DFA and let S be a set of D-consistent symbols. Then, $L(D)$ is DRE-definable if and only if D_S has the orbit property and every orbit language of D_S is DRE-definable. Moreover, if D consists of a single, nontrivial orbit and $L(D)$ is DRE-definable then there is at least one D-consistent symbol.*

We also review the following result by Brüggemann-Klein and Wood.

3.3.9 Lemma ([BKW98]). *Let D be a minimal DFA and S be the set of D-consistent symbols.*

(1) If D_S has the orbit property then $(D_S)_q$ is minimal for each state q in D.

(2) If p and q are states in the same orbit of D_S then $L((D_S)_p)$ is DRE-definable if and only if $L((D_S)_q)$ is DRE-definable.

Point (1) of the above lemma is immediate from combining Lemmas 5.9 and 5.10 from [BKW98]. Point (2) is immediate from the fact that DRE-definable regular languages are closed under derivatives [BKW98]. Notice that D_S does not have to be a minimal DFA in general. In particular, it can have states that are not reachable from the initial state.

Finally, these results lead to a recursive test that decides whether the language of a minimal DFA is DRE-definable. We present this test in Algorithm 1. In the algorithm it is ensured by Lemma 3.3.9 that the DFA for the recursive call in line 11 is always minimal. Correctness of the algorithm can be obtained by combining Theorems 3.3.7 and 3.3.8. Whenever we refer to *the* BKW-Algorithm in the remainder we consider Algorithm 1.

3.3.10 Theorem ([BKW98]). *Given a minimal DFA D it is decidable whether $L(D)$ is DRE-definable in time $O(|D|^2)$.*

Algorithm 1 The BKW-Algorithm [BKW98].

Algorithm BKW
Input: Minimal DFA $D = (Q, \Sigma, \delta, q_0, F)$
Output: *true* if $L(D)$ is DRE-definable, else *false*
$S \leftarrow$ the maximal set of D-consistent symbols
if D has only one trivial orbit **then return** *true*
if D has precisely one orbit and $S = \emptyset$ **then return** *false*
compute the orbits of D_S
if D_S does not have the orbit property **then return** *false*
for each orbit $\mathcal{O}$ in D_S **do**
    choose a state q in $\mathcal{O}$
    if not BKW($(D_S)_q$) **then return** *false*
return *true*

Moreover, Brüggemann-Klein and Wood showed how to build an equivalent DRE from a minimal DFA for a DRE-definable language. The constructed DREs are exponential in the size of the DFA in the worst case.

3.3.11 Theorem ([BKW98]). *Given a minimal DFA D, an equivalent DRE for $L(D)$ can be constructed in exponential time if $L(D)$ is DRE-definable. The constructed DRE is of size $O(|\Sigma|^{|\delta|})$.*

In [BGMN09], Bex et al. discussed a variant of the BKW-Algorithm which constructs smaller expressions than the original result above. Since the produced expressions are still exponentially larger than the input automaton in the worst case, we refer to the result in [BKW98] when necessary. Moreover, we remark that the algorithm in [BGMN09] contains a minor imprecision. The suggested optimization $((S + \varepsilon) \cdot \text{BKW}(A_S))^*$ for line 7 (in their paper) is not correct. Even under the obtained constraints the language of the constructed expression can become inequivalent to the input automaton which is due to the fact that the accepting states of A_S do not have to be equal to the states $c(a)$ that are reached by a symbol $a \in S$.

3.3.2 Closure Properties of DRE-definable languages

Next, we review the closure property of DRE-definable languages for several language-theoretic operations. In particular, we consider the union, intersection, difference, concatenation, star, and reversal operation. For each operation we consider the closure for languages over arbitrary and unary alphabets. Notice that several of these operations are relevant in XML schema management [GIM+13, MNS10]. We summarize the known closure properties of DRE-definable languages in Table 3.2.

DRE-definable languages are not closed under complement [GN08, GN12], concatenation [BKW98], and union [BKW98] even when the alphabet of the language is unary. Moreover, DRE-definable languages over arbitrary alphabets are not closed under intersection [Los10, CHM11], reversal [LMN12], and Kleene star [BKW98].

	$\lvert\Sigma\rvert = 1$	$\lvert\Sigma\rvert \geq 1$
$\setminus$	no	no
$\cdot$	no	no
$\cup$	no	no
$\cap$	yes	no
rev	yes	no
*	yes	no
$w\setminus$	yes	yes

Table 3.2: Closure properties of DRE-definable languages. By "$\setminus$" we denote the complement $\Sigma^*\setminus L$ for a given language L. By "rev" we denote the reversal operation and by "$w\setminus$" the Brzozowski derivative with respect to words w over Σ.

DRE-definable languages over unary alphabets are closed for these three operations [LMN12]. In addition, we remark that DRE-definable languages are closed under taking the Brzozowski derivative with respect to words [BKW98]. That is, for every DRE-definable language L over an alphabet Σ and word $w \in \Sigma^*$, the language $w\setminus L$ is DRE-definable.

3.3.3 Determinism for Variants of Regular Expressions

In the following we introduce the notion of determinism for the class RE(!, •). Recall that, by ! we denote the restricted negation (for a finite set of symbols) and by • we denote the wildcard symbol (see Definition 3.1.1). We formally define determinism for the class RE(!, •) using Glushkov automata (see [BEGO71, Glu61]) instead of a semantical definition like in Definition 3.3.2.

Let r be a regular expression in RE(!, •) over an alphabet Σ_r. By num(r) we denote the *numbered regular expression* obtained from r by replacing each subexpression of the form $!(a_1 + \cdots + a_n)$, •, or $a \in \Sigma$ (that is not in the scope of an !-operator) with a unique number, increasing from left to right. For example, for $r = a\ !(a) \bullet (a + bc)^*\bullet\ !(a + b)$ we have num(r) = 1 2 3 (4 + 5 6)*7 8. More precisely, num(r) can be obtained by traversing the syntax tree of r depth-first left-to-right and replacing each atomic expression by unique increasing numbers. By denum$_r$ we denote the mapping that maps each number i to the subexpression it replaced in r. For the above example, we have that denum$_r(1) = a$, denum$_r(2) =!(a)$, denum$_r(3) = \bullet$, et cetera. Notice that r_m can be seen as a regular expression over a finite alphabet $\Sigma' \subseteq \mathbb{N}$ where $|\Sigma'|$ is the number of leaves in the syntax tree of r.

3.3.12 Definition. Let r be an RE(!, •) and r_m its numbered expression. The *Glushkov automaton* G_r of r is the tuple $(Q_r, \Sigma_r, \delta_r, q_0, F_r)$ where the following holds:

- $Q_r = \{q_0\} \uplus \Sigma'$ is a finite set of states. That is, Q_r contains an initial state and one state for each position i in the numbered expression r_m.
- If $\varepsilon \in L(r)$ then the set of accepting states is $F_r = \text{last}(r_m) \uplus \{q_0\}$; otherwise, $F_r = \text{last}(r_m)$.

– The transition function δ_r contains the following transitions for every $a \in \Sigma_r$ and $i \in Q_r$:

  - $\delta_r(q_0, a) = \{i \in \text{first}(r_m) \mid \text{denum}(i) = a, \text{denum}(i) = \bullet,$ or $\text{denum}(i) = !(a_1 + \cdots + a_\ell)$ with $a \notin \{a_1, \ldots, a_\ell\}\}$, and
  - $\delta_r(i, a) = \{j \in \text{follow}(r_m, i) \mid \text{denum}(j) = a, \text{denum}(j) = \bullet,$ or $\text{denum}(j) = !(a_1 + \cdots + a_\ell)$ with $a \notin \{a_1, \ldots, a_\ell\}\}$.

It is known that, for a regular expression r, an equivalent Glushkov automaton can be constructed in polynomial time in $|r|$ [BK93]. The following is a direct corollary of this result.

3.3.13 Corollary. *For every RE($!, \bullet$) r the Glushkov automaton G_r of r can be constructed in polynomial time in $|r|$. Moreover, it holds that $L(r) = L(G_r)$.*

The definition of Glushkov automata reveals that determinism in regular expressions is very similar to determinism in finite automata, i.e., an RE r is deterministic if and only if the Glushkov automaton of r is a DFA. Moreover, there exists a similar connection for unambiguity in regular expressions and finite automata, i.e., an RE r is unambiguous if and only if the Glushkov automaton of r is a UFA. We exploit this connection to define the notions of determinism and unambiguity for RE($!, \bullet$) expressions.

3.3.14 Definition. Let r be an RE($!, \bullet$) then

– r is *deterministic* (or a *DRE*($!, \bullet$)) if G_r is a DFA, and

– r is *unambiguous* if G_r is a UFA.

Determinism has also been studied for the class RE(#), i.e., the class of regular expressions with counting operators [KT07, GGM12, Hov09]. For such expressions determinism seems to bring up even more challenging research questions. In fact, the class RE(#) even leads to two different but natural notions of determinism, i.e., *weak* and *strong determinism*. Weak determinism for regular expressions with counting operators is defined analogously to determinism for regular expressions without counting operators. Opposed to that strong determinism also requires that the use of counting operators and the Kleene-star operator has to be deterministic. For example, the expression $(a^{1,2})^{3,4}$ is not strongly deterministic because when reading an a one does not know which counting operator is "raised". However, these notions are even less understood than the notion of determinism for standard regular expressions. For example, it is still unknown if it is decidable whether the language of a regular expression is definable by a weakly deterministic expression. In the following, we focus on determinism for the classes RE and RE($!, \bullet$) and investigate their complexity for several computational problems. All lower bounds, however, can be trivially transferred to weakly and strongly deterministic regular expressions (with counting operators).

4

Regular Expressions in XML Schema Languages

In two of the most prevalent schema languages for XML, namely Document Type Definitions (DTDs) and XML Schema (XSDs), it is not allowed to use arbitrary regular expressions. Instead, all used regular expressions have to be deterministic to ensure compatibility with SGML where they were introduced to guarantee efficient parsing. In this chapter we examine the complexity of theoretical problems for deterministic regular expressions which become relevant when processing DTDs and XSDs. Although DRE-definable languages have been around for quite some time, they are not yet well-understood. This motivates us to study their foundational properties.

Since not every regular language can be represented by a deterministic regular expression, it is natural to examine whether a regular language is definable by a DRE or not. Although it can be checked in linear time whether a regular expression is deterministic [GMS12], we will see that it is computationally harder to check whether a regular language is DRE-definable. In Section 4.1, we consider this problem for several representations of regular languages, i.e., we consider the problem for input REs, NFAs, RE(#)s, and minimal DFAs. (Notice that in XML Schema one is allowed to write regular expressions with counting operators.) We have already seen that the problem is decidable for each variant due to [BKW98]. We focus on the exact computational complexity of these problems. We can settle the complexity for the variants regarding NFAs, REs, and RE(#)s, though for minimal DFAs the exact complexity remains open.

In Sections 4.2 and 4.3, we examine the relative descriptional complexity of DRE-definable languages. That is, we study whether the representation of regular languages as minimal DREs is less succinct than alternative (more general) representations, e.g., minimal DFAs and REs.

Previously, Brüggemann-Klein and Wood showed that there exists an algorithm that, given a DRE-definable language as a minimal DFA, constructs an equivalent DRE that is at most exponentially larger than the minimal DFA [BKW98]. However, in [BKW98] the main focus was on proving that the problem is decidable and it is known that the constructed DREs are not necessary minimal. Nonetheless, we show that there cannot exist a significantly better procedure for constructing deterministic regular expressions in Section 4.2. More precisely, we prove that an exponential blow-up cannot be avoided when translating a DFA or an RE into an equivalent DRE in general.

Finally, we examine the descriptional complexity of several language-theoretic operations on DRE-definable languages in Section 4.3. In particular, we investigate the boolean operations ($\setminus$, $\cap$, and $\cup$), the reversal, and concatenation operation. We have already seen in Section 3.3.2 that DRE-definable languages are not closed under any of these operations. Therefore, there basically exist two alternatives to define the worst-case descriptional complexity blow-up when applying one of the aforementioned operations on two DREs. The first alternative allows to write the result as an RE in general (since the resulting language may not be DRE-definable) and studies the worst-case complexity of minimal REs for the resulting language. This setting was already examined in [GN08, GN12]. The second alternative examines the descriptional complexity of the result if and only if the resulting language is still DRE-definable. We will study the descriptional complexity of DRE-definable languages according to the second alternative. As far as we know, we are the first to consider this setting and we remark that the results from [GN08, GN12] cannot naïvely be transferred to work in this setting. We start our studies by reviewing basic results on the state complexity of boolean operations on DFAs for DRE-definable languages. Afterwards, we examine the descriptional complexity of DREs that are the result of applying an operation on two input DREs and prove that an exponential blow-up cannot be avoided for each of the above operations in general.

4.1 The Complexity of the DRE-Definability Problem

In this section we study the complexity of deciding whether a regular language is DRE-definable, i.e., we examine the complexity of the following decision problem:

Problem (DRE-Definability(X)).
Given: A regular language L represented as X.
Question: Is L DRE-definable?

We consider the problem for several variants depending on the input. For each variant, we choose the parameter of the problem X to be one of the following representations: an NFA, RE, RE(#), or a minimal DFA. To refer to the particular variants, we put the respective representation between braces at the end. For example, DRE-Definability(RE) is the problem: Given a regular expression r, is $L(r)$ DRE-definable?

input	lower bound	upper bound
min. DFAs	NLOGSPACE (Th. 4.1.25, [CDLM13])	P [BKW98] / (NLOGSPACE [LBC14])
NFAs	PSPACE [BGMN09]	PSPACE (Th. 4.1.23, [CDLM13])
REs	PSPACE [BGMN09]	PSPACE (Th. 4.1.23, [CDLM13])
RE(#)s	EXPSPACE (Th. 4.1.24, [CDLM13])	EXPSPACE (Th. 4.1.24, [CDLM13])

Table 4.1: Complexity of the DRE-Definability problem.

We summarize all known results about the DRE-Definability problem in Table 4.1. The DRE-Definability problem was first studied by Brüggemann-Klein and Wood nearly two decades ago. It is known that the problem can be decided in polynomial time in the size of a given minimal DFA [BKW98]. Furthermore, it is known that DRE-Definability for NFAs and REs is PSPACE-hard [BGMN09].

In [CDLM13] we obtained the results that will be presented in the remainder of this section. Our main result is a PSPACE algorithm for the problem DRE-Definability(NFA). By the existence of such an algorithm the PSPACE upper bound for DRE-Definability(RE) and the EXPSPACE upper bound for DRE-Definability(RE(#)) are direct consequences.

We use Section 4.1.1 to 4.1.4 to present the PSPACE algorithm for NFAs. Towards the result we need a more detailed analysis of the BKW-Algorithm (see, e.g., Algorithm 1). To this end, we define, in Section 4.1.1, *level automata* which describe the detailed structure of the input automata for every recursion step of the algorithm. Using the notion of level automata we can prove certain properties of the automata during a run of the algorithm in the subsequent sections. In Section 4.1.2, we prove an upper bound on the recursion depth of the algorithm that is polynomial in the size of a given NFA for the input language. (We remark that the upper bound is also polynomial in the size of a minimal NFA for the input language.) Moreover, this bound is crucial for understanding why there exists a PSPACE algorithm for DRE-Definability(NFA). In Section 4.1.3, we analyze the different causes of why the algorithm returns that the input language is not DRE-definable. To this end, we define three *violations* that uniquely characterize in which cases the algorithm fails for a certain input. Finally, in Section 4.1.4, we construct an algorithm that decides whether there exists a violation for the input automaton at some point in the recursion, i.e., whether the algorithm outputs that the input language is not DRE-definable. The algorithm thereby adapts Algorithm 1 and applies it on the minimal DFA that is computed on-the-fly from the input NFA. Using the polynomial upper bound on the recursion depth we prove that the algorithm runs in PSPACE by a long mutual induction. Since PSPACE is closed under complement the result holds in the end.

At last, we examine the complexity of the problems DRE-Definability(RE(#)) and DRE-Definability(*min*DFA) in Section 4.1.5. We prove that the problem DRE-Definability(RE(#)) is EXPSPACE-complete and that the problem DRE-Definability(*min*DFA) is NLOGSPACE-hard. Lu et al. [LBC14] showed that DRE-Definability is in NLOGSPACE when the input is a minimal DFA over an alphabet that has at most logarithmic size in the number of states in the DFA.

4.1.1 Level-Automata

In this section our goal is to define a set of automata that describe the detailed structure of the input DFA for every recursive call (in line 11) during a run of the BKW-Algorithm (see Algorithm 1). Using this notion we can analyze the BKW-Algorithm in more detail afterwards.

For the remainder of this section let D be a minimal DFA. Whenever we refer to a set S of D-consistent symbols we assume that the set is maximal for D, i.e., there does not exist a D-consistent symbol a that is not in S. Henceforth, we always refer to *the* set of D-consistent symbols. It is important to keep in mind that our automata always contain only useful states in the following.

We now define, for a given minimal DFA D, a set of *level automata* which describe the structure of the orbit of some state q at the moment of the recursive call (in line 11) of the BKW-Algorithm when it is called with D as the input automaton. Using these automata we can examine how, for a state q of D, the orbit of q evolves during the recursion. To this end, observe that we always delete two kinds of transitions in every iteration of Algorithm 1: the D-consistent transitions (which we delete to obtain D_S from D) and the inter-orbit transitions in D_S (which we delete to obtain $(D_S)_q$). Moreover, in every round of the algorithm either of the sets can be empty but not both together as otherwise the algorithm fails.

4.1.1 Definition. For a state q of a minimal DFA D and $k \in \mathbb{N}$ the *level k automaton of D for the state q*, denoted $\mathrm{lev}_k(D, q)$, is inductively defined as follows:

- $\mathrm{lev}_0(D, q) = D$.
- Let S be the maximal set of D-consistent symbols. Then

$$\mathrm{lev}_1(D, q) = \begin{cases} (D_S)_q & \text{if } D \text{ has more than one orbit and} \\ & \qquad D_S \text{ has the orbit property,} \\ (D_S)_q & \text{if } S \neq \emptyset \text{ and } D_S \text{ has the orbit property,} \\ \emptyset & \text{otherwise.} \end{cases}$$

- For $k > 1$, let $B := \mathrm{lev}_{k-1}(D, q)$ and S_{k-1} be the maximal set of B-consistent symbols. Then

$$\mathrm{lev}_k(D, q) = \begin{cases} (B_{S_{k-1}})_q & \text{if } S_{k-1} \neq \emptyset \text{ and } B_{S_{k-1}} \text{ has the orbit property,} \\ \emptyset & \text{otherwise.} \end{cases}$$

The above definition complies exactly with the construction in Algorithm 1 if state q is chosen every time in line 10. Notice that, the top level recursion of the BKW-Algorithm is slightly different from the others: the input DFA D of the top level can have multiple orbits, whereas this is not the case for deeper recursive levels. Thus, the definition of $\mathrm{lev}_1(D, q)$ is different to the definition of level automata for larger k. Moreover, $\mathrm{lev}_k(D, q)$ is always minimal according to Lemma 3.3.9.

4.1.2 Example. We now illustrate the notion of level automata for an example. Consider the following DFA D from Figure 4.1.

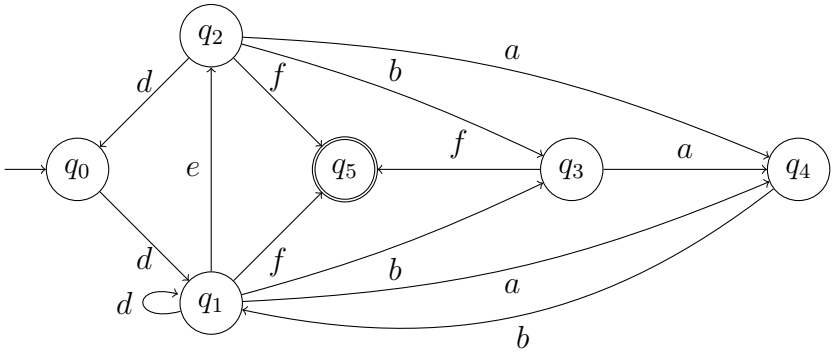


Figure 4.1: The automaton D, i.e., $\mathrm{lev}_0(D, q_0)$, $S = \emptyset$.

By definition, $\mathrm{lev}_0(D, q_0)$ is the automaton D itself. In order to build the level automaton for the next iteration of Algorithm 1, observe that D has two orbits: one orbit contains the states $q_1, \ldots, q_4$ and the other orbit contains only q_5. The set of D-consistent symbols S is empty since q_5 has no outgoing transitions, i.e., D_S equals D. Moreover, D_S has the orbit property since all transitions that leave $\mathcal{O}(q_0)$ are f-labeled and go to state q_5. As such, $\mathrm{lev}_1(D, q_0)$ equals $(D_\emptyset)_{q_0}$ which is the orbit automaton of q_0 in D. We illustrate $\mathrm{lev}_1(D, q_0)$ in Figure 4.2.

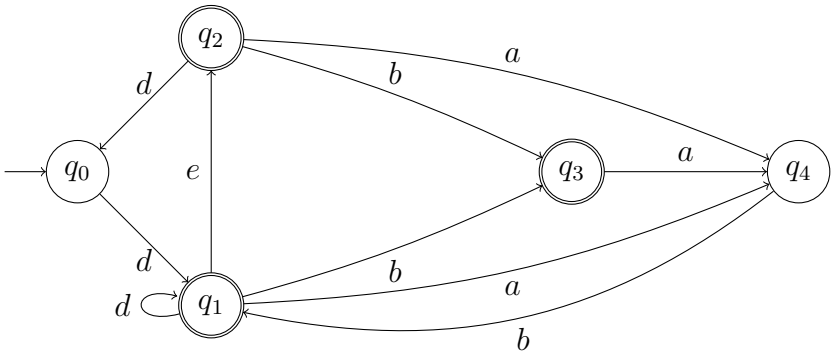


Figure 4.2: The automaton $\mathrm{lev}_1(D, q_0)$ with $\mathrm{lev}_1(D, q_0)$-consistent symbols $S_1 = \{a\}$.

For the second iteration of the BKW-Algorithm the level automaton is built as follows. The maximal set of $\mathrm{lev}_1(D, q_0)$-consistent symbols is $S_1 = \{a\}$. Since the set is not empty we obtain the S_1-cut of $\mathrm{lev}_1(D, q_0)$ (see Figure 4.3). Moreover, we can observe that the automaton in Figure 4.3 has the orbit property because it contains only one orbit. Thus, the BKW-Algorithm proceeds with the next recursive call. By the definition of level automata, we get that $\mathrm{lev}_2(D, q_0)$ is the orbit automaton of q_0 in the S_1-cut of $\mathrm{lev}_1(D, q_0)$ (that is, $\mathrm{lev}_2(D, q_0) = (\mathrm{lev}_1(D, q_0)_{S_1})_{q_0}$). The automaton is illustrated in Figure 4.4.

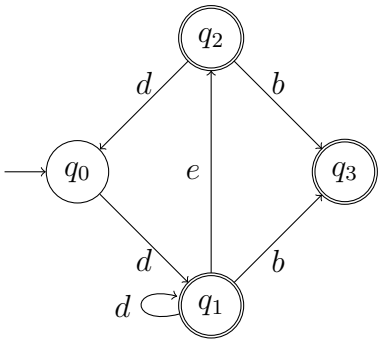


Figure 4.3: The S_1-cut of $\mathrm{lev}_1(D, q_0)$.

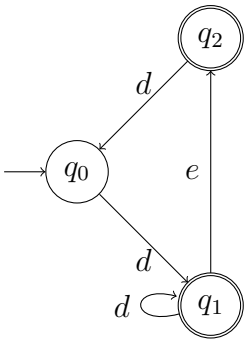


Figure 4.4: $\mathrm{lev}_2(D, q_0), S_2 = \emptyset$.

Finally, we observe that $\mathrm{lev}_2(D, q_0)$ has only one orbit but no $\mathrm{lev}_2(D, q_0)$-consistent symbols. In accordance with the BKW-Algorithm, this means that $L(D)$ is not DRE-definable. By the definition of level automata, we have that $\mathrm{lev}_3(D, q_0) = \emptyset$.

We conclude the section by proving that the definition of level automata is a valid characterization of DRE-definable languages.

4.1.3 Lemma. *Let D be a minimal DFA. Then the following are equivalent:*

(1) $L(D)$ is DRE-definable,

(2) for every state q of D and $k \in \mathbb{N}$, $L(lev_k(D, q))$ is DRE-definable,

(3) for every state q of D and $k \in \mathbb{N}$, $L(lev_k(D, q))$ is DRE-definable and $lev_k(D, q)_{S_k}$ has the orbit-property.

Proof. The implications (3) $\Rightarrow$ (2) and (2) $\Rightarrow$ (1) are obvious. It remains to show the implication (1) $\Rightarrow$ (3). Assume that there exist a state q of D and $k \in \mathbb{N}$ such that $\mathrm{lev}_k(D, q)$ is not DRE-definable or $\mathrm{lev}_k(D, q)_{S_k}$ does not have the orbit-property where S_k is the set of $\mathrm{lev}_k(D, q)$-consistent symbols. By (2) in Lemma 3.3.9, we know that the choice of a state in line 10 of the BKW-Algorithm is arbitrary. Consider a run of the BKW-Algorithm on D where, at each recursion level $i < k$, state q is chosen in line 10 when the current orbit $\mathcal{O}$ contains q. Then, at level k, Algorithm 1 is called on $\mathrm{lev}_k(D, q)$ and returns *false* by definition, i.e., $L(D)$ is not DRE-definable. □

4.1.2 A Bound on the Recursion Depth of the BKW-Algorithm

Using the notion of level automata we are now able to prove a polynomial upper bound on the maximal recursion depth of the BKW-Algorithm in terms of the size of an input NFA. Moreover, this bound will be central when assembling the PSPACE-algorithm for NFAs. We first observe that once a state becomes a gate in the BKW-Algorithm its outgoing transitions disappear in deeper recursion levels.

4.1.4 Lemma. *Let D be a minimal DFA and q be a gate in $lev_k(D,q)$ for some $k > 0$. Then either $lev_{k+1}(D,q) = \emptyset$ or q has strictly less outgoing transitions in $lev_{k+1}(D,q)$. In the latter case, q is also a gate in $lev_{k+1}(D,q)$.*

Proof. We know that $\mathrm{lev}_k(D,q)$ has precisely one orbit because $k > 0$. Thus, there exist no inter-orbit transitions in $\mathrm{lev}_k(D,q)$. Since q is a gate by assumption, q has to be accepting. Assume that the set of $\mathrm{lev}_k(D,q)$-consistent symbols S_k is not empty and that $\mathrm{lev}_k(D,q)_{S_k}$ fulfills the orbit property. By definition it holds that $\mathrm{lev}_{k+1}(D,q) = (\mathrm{lev}_k(D,q)_{S_k})_q$, which is obtained from $\mathrm{lev}_k(D,q)$ by removing all transitions that leave an accepting state and are labeled with a symbol in S_k first and taking the orbit automaton of q afterwards. Since q is accepting and $S_k \neq \emptyset$ it follows that there exists at least one q-outgoing $\mathrm{lev}_k(D,q)$-consistent transition which is not present in $\mathrm{lev}_k(D,q)_{S_k}$. This transition is also not present in $\mathrm{lev}_{k+1}(D,q)$ and, therefore, q has strictly less outgoing transitions on this level. If there does not exist a $\mathrm{lev}_k(D,q)$-consistent symbol or $\mathrm{lev}_k(D,q)_{S_k}$ does not fulfill the orbit property then $\mathrm{lev}_{k+1}(D,q) = \emptyset$ by definition. □

Now we are able to provide an upper bound on the number of recursion levels that is needed until a gate is deleted in the BKW-Algorithm. We thereby use the fact that in a minimal DFA every state has at most $|\Sigma|$ outgoing transitions.

4.1.5 Lemma. *Let D be a minimal DFA and q be a gate in $lev_k(D,q)$ for some $k \geq 0$. Let $n = |\Sigma| + 1$ if $k = 0$ and $n = |\Sigma|$ otherwise. Then either q is a trivial orbit in $lev_{k+n}(D,q)$ or $lev_{k+n}(D,q) = \emptyset$.*

Proof. Choose n as desired in the lemma and assume that $\mathrm{lev}_{k+n}(D,q) \neq \emptyset$. By definition, q exists and is a gate in $\mathrm{lev}_{k+\ell}(D,q)$ for every $0 \leq \ell \leq n$. Since q has at most $|\Sigma|$ outgoing transitions in $\mathrm{lev}_k(D,q)$ the lemma statement holds by Lemma 4.1.4. □

Using Lemma 4.1.5 we can also give an upper bound on the number of recursion levels that are needed until a state p in D becomes a gate in the BKW-Algorithm.

4.1.6 Lemma. *Let D be a minimal DFA and p be a state of $lev_k(D,p)$ for some $k \in \mathbb{N}$. Let ℓ be the length of a shortest path from p to a gate in $lev_k(D,p)$. Then either $lev_{k+|\Sigma|\cdot\ell+1}(D,p) = \emptyset$ or p is a gate in $lev_{k+|\Sigma|\cdot\ell+1}(D,p)$.*

Proof. Let ℓ_j be the length of a shortest path from p to a gate in $\mathrm{lev}_j(D,p)$. If $\ell_j = 0$ then p is a gate. If $\ell_j > 1$ then we show that within $|\Sigma|$ recursion levels ($|\Sigma| + 1$ levels if $j = 0$) this value decreases strictly. Let q be a gate such that there is a path

$$p = p_0 \to p_1 \to \cdots \to p_{\ell_j - 1} \to p_{\ell_j} = q$$

of length ℓ_j from p to q in $\mathrm{lev}_j(D,p)$.

Notice that ℓ_j cannot increase as long as no transition on the path is removed. (If q is a gate at some level then it remains a gate at the next level). Assume now that a transition $p_i \to p_{i+1}$ is removed from $\mathrm{lev}_{j+m}(D,p)$ for some $m \in \mathbb{N}$. By definition of level automata, the transition is either $\mathrm{lev}_{j+m}(D,p)$-consistent or an

inter-orbit transition. In both cases p_i is a gate in $\mathrm{lev}_{j+m}(D,p)$ and thus $\ell_{j+m} < \ell_j$. Therefore, ℓ_j cannot increase when considering ascending levels.

From Lemma 4.1.5 we know that after at most $n = |\Sigma|$ recursion levels ($|\Sigma|+1$ if we start from level $j = 0$) the orbit automaton $\mathrm{lev}_{j+n}(D,q)$ is either empty or trivial. But this means that the transition $p_{\ell_j - 1} \to q$ is an inter-orbit transition in $\mathrm{lev}_{j+n}(D,q)$, i.e., p_{ℓ_j-1} is a gate. Thus, $\ell_{j+n} < \ell_j$ which proves that ℓ_j strictly decreases within $|\Sigma|$ recursion levels ($|\Sigma|+1$ if $j = 0$). In summary we know that after at most $j = \ell \cdot |\Sigma| + 1$ recursion levels either $\mathrm{lev}_{k+j}(D,p)$ is empty or $\ell_j = 0$, which means p is a gate in $\mathrm{lev}_{k+j}(D,p)$. □

We also need the following observation about NFAs and their minimal DFAs which states that paths to accepting states in minimal DFAs are always short when compared to NFAs. Recall that we denote by $[N]$ the minimal DFA for an NFA N.

4.1.7 Lemma. *Let N be an NFA of size n. Then, for every state in the minimal DFA $[N]$ of N, there is a path leading to some accepting state of length at most $n-1$.*

Proof. We prove the assumption by examining the states in the power set automaton $\mathcal{P}(N)$ (see Definition 2.1.4). Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and $p = \{q_1, \ldots, q_k\}$ be a non-empty subset of Q (that is, a state in $\mathcal{P}(N)$). Remember that we only consider NFAs where all states are useful. Thus, for every state $q \in Q$, there exists a path to some accepting state $q^f \in F$ of length at most $n-1$. In particular, this means that there is such a path from q_1 to an accepting state $q_1^f \in F$. Then, by definition of $\mathcal{P}(N)$, there exists a path from p to an accepting state p^f in $\mathcal{P}(N)$ that contains q_1^f of length at most $n-1$. Let w be a word such that $\delta^*_{\mathcal{P}(N)}(p,w) = p^f$ and $|w| = n-1$. Then, it also holds that $\delta^*_{[N]}([p],w) = [p^f]$, which concludes the proof. □

Using Lemmas 4.1.5, 4.1.6 and 4.1.7 we provide an upper bound on the number of recursion levels that a state can be present in the BKW-Algorithm.

4.1.8 Lemma. *Let N be an NFA of size n. Then, $lev_{n\cdot|\Sigma|+2}([N],p) = \emptyset$ for every state p of $[N]$.*

Proof. Let p be a state in $[N]$. By Lemma 4.1.7 there exists a path from p to some accepting state q in $[N]$ of length at most $n-1$. Since q is accepting, q is a gate in $[N]$. Thus, the length of a shortest path from p to a gate in $[N]$ is smaller or equal to $n-1$. Remember that $[N] = \mathrm{lev}_0([N],p)$. By Lemma 4.1.6 we know that p is a gate in $\mathrm{lev}_{(n-1)\cdot|\Sigma|+1}([N],p)$ or $\mathrm{lev}_{(n-1)\cdot|\Sigma|+1}([N],p)$ is empty. In the latter case the lemma is shown. In the first case p is a trivial orbit in $\mathrm{lev}_{n\cdot|\Sigma|+1}([N],p)$ by Lemma 4.1.5. Then it holds that $\mathrm{lev}_{n\cdot|\Sigma|+2}([N],p) = \emptyset$, which concludes the proof. □

Finally, we prove that the maximal recursion depth of the BKW-Algorithm is polynomially bounded in the size of an NFA for the input language.

4.1.9 Theorem. *Let N be an NFA with size n. The recursion depth of Algorithm 1 on $[N]$ is at most $n \cdot |\Sigma| + 2$.*

Notice that, by Theorem 4.1.9, the recursion depth of the BKW-Algorithm is also polynomially bounded in the size of a minimal NFA for the language.

4.1.3 Consistency Violations

In the following we analyze the possible causes of failure for the BKW-Algorithm. To this end, we identify three properties such that the BKW-Algorithm fails if and only if one of them holds for some orbit automaton at some level k.

When examining Algorithm 1 one can immediately see that there are two situations in which it can reject at some point in the recursion: (in line 6) when the automaton consists only of one orbit that has no D-consistent symbols or (in line 8) when the S-cut of the automaton does not have the orbit property. The latter means that there exist two gates of the same orbit in the S-cut such that either they do not have the same transitions to the outside or one of them is accepting while the other one is not. We now formalize these different types of violations and prove afterwards that the BKW-Algorithm fails for some input if and only if one of these violations is found at some point in the recursion.

4.1.10 Definition. Let D be a minimal DFA D and S the set of D-consistent symbols. Then, D has an

- *out-consistency violation* if there exist gates q_1 and q_2 in the same orbit $\mathcal{O}$ of D_S and there exists a state q outside $\mathcal{O}$ such that there is a transition $q_1 \xrightarrow{a} q$ and no transition $q_2 \xrightarrow{a} q$,
- *acceptance consistency violation* if there exist gates q_1 and q_2 in the same orbit of D_S such that q_1 is accepting and q_2 is not, and
- *orbit consistency violation* if there exists an accepting state q_1 such that, for every symbol a, there exists another accepting state q_2 in $\mathcal{O}(q_1)$ in D such that for every state q at most one of the transitions $q_1 \xrightarrow{a} q$ and $q_2 \xrightarrow{a} q$ exists. (Notice that $\delta(q_1, a)$ may be empty because D contains only useful states.)

We say that a DFA D *has a violation* if and only if it has at least one of the above violations.

Notice that similar to the BKW-Algorithm the first two violations focus on D_S and the last one on D. We show next that these violations are a valid characterization of DRE-definable languages.

4.1.11 Theorem. *Let D be a minimal DFA. Then, $L(D)$ is not DRE-definable if and only if there exist a state q of D and a number $k \in \mathbb{N}$ such that $lev_k(D, q)$ has a violation.*

Proof. We prove the direction from right to left first. Therefore, we distinguish three cases depending on the violation that occurs in $\mathrm{lev}_k(D, q)$.

If for some state q of D and a number $k \in \mathbb{N}$ the automaton $\mathrm{lev}_k(D, q)_{S_k}$ has an out-consistency violation or an acceptance consistency violation then $\mathrm{lev}_k(D, q)_{S_k}$ does not fulfill the orbit property. By Lemma 4.1.3 it holds $L(D)$ is not DRE-definable.

If $\text{lev}_k(D, q)$ has an orbit consistency violation then there does not exist a $\text{lev}_k(D, q)$-consistent symbol. In the case that $k \geq 1$ we also know that $\text{lev}_k(D, q)$ has only one orbit. Thus, $L(\text{lev}_k(D, q))$ is not DRE-definable by Theorem 3.3.8. It follows that $L(D)$ is not DRE-definable by Lemma 4.1.3. In the case that $k = 0$ the automaton $\text{lev}_1(D, q)$ is the orbit automaton of q in D. This automaton consists of a single orbit whose states are the ones in the orbit of q in D. (Since there does not exist a D-consistent symbol by assumption we have that $D_S = D$.) Furthermore, there still cannot exist a $\text{lev}_1(D, q)$-consistent symbol because all accepting states of the orbit of q in D are still accepting in $\text{lev}_1(D, q)$. Therefore, $L(\text{lev}_1(D, q))$ is not DRE-definable by Theorem 3.3.8 and, thus, the language $L(D)$ is not DRE-definable by Lemma 4.1.3.

It remains to prove the direction from left to right. Therefore, assume that $L(D)$ is not DRE-definable. By Lemma 4.1.3 there exist a state q and a number k such that $L(\text{lev}_k(D, q))$ is not DRE-definable, i.e., the BKW-Algorithm fails when given $\text{lev}_k(D, q)$ as input. If we assume that it fails in line 6 then there does not exist a $\text{lev}_k(D, q)$-consistent symbol. Thus, an orbit consistency violation occurs in $\text{lev}_k(D, q)$. If we assume that it fails in line 8 then $\text{lev}_k(D, q)_{S_k}$ does not have the orbit property. Thus, either an out-consistency violation occurs in $\text{lev}_k(D, q)_{S_k}$ or an acceptance consistency violation occurs in $\text{lev}_k(D, q)_{S_k}$. This concludes the proof. □

4.1.4 DRE-Definability for REs and NFAs

We are now ready to prove that DRE-Definability for NFAs and REs is in PSPACE. Therefore, we construct an algorithm that exploits Theorem 4.1.11 in the following way. Given an NFA N, it searches for a level k and a state p in the minimal DFA $[N]$ such that $\text{lev}_k([N], p)$ has a violation. Since PSPACE is closed under complement the result holds.
Notice that $[N]$ can be exponentially larger than N in general and, therefore, that we cannot simply compute $[N]$ in polynomial space in the size of N. To overcome this difficulty we use the following two ideas:

(1) Use the fact that the maximal recursion depth of Algorithm 1 on $[N]$ is polynomial in the size of the NFA N (see Theorem 4.1.9).

(2) Adapt Algorithm 1 using Theorem 4.1.11 and apply it on the minimal DFA $[N]$ which is constructed on-the-fly from the NFA.

In the following we explain how the algorithm can detect whether there is a violation in the minimal DFA $[N]$ on-the-fly. To this end, we fix the following notation. By $N = (Q_N, \Sigma, \delta_N, q_N^0, F_N)$ we always denote an NFA. By $[N]$ we denote the minimal DFA for $L(N)$. Since the algorithm computes $[N]$ on-the-fly we often identify the states in $[N]$ only by a set of states in Q_N. For a set of states $q \subseteq Q_N$ we denote by $[q]$ the corresponding state in the minimal DFA $[N]$. More precisely, $[q]$ is the state that can be reached by the following set of words in the minimal DFA: $\{w \mid \exists t \in q \text{ such that } \delta_N^*(t, w) \cap F_N \neq \emptyset\}$, i.e., the Myhill-Nerode equivalence class of q. (Notice that the automaton $[N]$ can be obtained by merging states of the

power set automaton $\mathcal{P}(N)$ [HMU13].) Whenever we talk about $\mathrm{lev}_k([N],[q])_{S_k}$ the set S_k is the set of $\mathrm{lev}_k([N],[q])$-consistent symbols.

The key result of this section (which is obtained in Lemma 4.1.22) proves that we can detect whether there is a violation for $[N]$ in level k in polynomial space in k and the size of N. To this end, we consider the following precise problems:

- Out-Cons-Violation: Given an NFA N and a number $k \in \mathbb{N}$.
  Is there a $q \subseteq Q_N$ such that $\mathrm{lev}_k([N],[q])_{S_k}$ has an out-consistency violation?
- Acc-Cons-Violation: Given an NFA N and a number $k \in \mathbb{N}$.
  Is there a $q \subseteq Q_N$ such that $\mathrm{lev}_k([N],[q])_{S_k}$ has an acceptance consistency violation?
- Orbit-Cons-Violation: Given an NFA N and a number $k \in \mathbb{N}$.
  Is there a $q \subseteq Q_N$ such that $\mathrm{lev}_k([N],[q])$ has an orbit consistency violation?

To decide the above problems we study the complexity of the following subproblems first. For an NFA N, non-empty sets $p, q \subseteq Q_N$, $a \in \Sigma$, and a number $k \in \mathbb{N}$ we define:

- BkwEdge: Given (N, p, q, a, k).
  Is $[p] \xrightarrow{a} [q]$ a transition in $\mathrm{lev}_k([N],[p])$?
- BkwReachability: Given (N, p, q, k).
  Is $[q]$ reachable from $[p]$ in $\mathrm{lev}_k([N],[p])$?
- SameOrbit: Given (N, p, q, k).
  Are $[p]$ and $[q]$ in the same orbit of $\mathrm{lev}_k([N],[p])$?
- InterOrbit: Given (N, p, k).
  Is there an inter-orbit transition $[p] \xrightarrow{a} [q]$ for some label a and q in $\mathrm{lev}_k([N],[p])$?
- BkwAcceptance: Given (N, p, k).
  Is $[p]$ accepting in $\mathrm{lev}_k([N],[p])$?
- IsGate: Given (N, p, k).
  Is $[p]$ a gate in $\mathrm{lev}_k([N],[p])$?

Notice that SameOrbit and InterOrbit are only non-trivial if $k = 0$. (Level k automata for $k > 0$ always have exactly one orbit.) Moreover, we consider for each of the above problems X a variation called X-Cut which decides, given the same input, whether the problem X is *true* for the automaton $\mathrm{lev}_k([N],[p])_{S_k}$ (instead of $\mathrm{lev}_k([N],[p])$). Whenever we say that a problem X (or X-Cut) *for N at level k* can be decided, we mean that it can be decided for the NFA N and arbitrary sets of states $p, q \subseteq Q_N$, $a \in \Sigma$, and k.

Our proof that DRE-Definability(NFA) is in PSPACE is a careful mutual induction on the above problems. We briefly outline the proof in the following. First, we show that BkwEdge, BkwEdge-Cut, and BkwAcceptance for N

at level 0 can be decided (see Lemmas 4.1.12 and 4.1.13). Second, we prove in Lemmas 4.1.14 to 4.1.18 a set of implications that are of the form: *If problem X for N at level k can be decided then problem Y for N at level k (or level $k+1$) can also be decided.* In Lemmas 4.1.19 to 4.1.21 we combine the previous lemmas to prove that each of the above problems for N at level k can be decided. Finally, we prove in Lemma 4.1.22 that we can detect whether there is a violation for $[N]$ at level k. In Theorem 4.1.23 we state that DRE-Definability(NFA) is in PSPACE.

The basis of the entire mutual induction relies on being able to test whether some state is accepting in $[N]$, whether a certain transition is present in $[N]$, and whether a certain transition is present in the S-cut $[N]_S$. Before we can decide these questions, we need that it can be decided whether two states $[p]$ and $[q]$ for $p, q \subseteq Q_N$ are equivalent in $[N]$.

4.1.12 Lemma. *Given N and $p, q \subseteq Q_N$, it can be tested whether $[p] = [q]$ in space $O(|N|^2)$.*

Proof. We show that a nondeterministic Turing machine can test in nondeterministic space $O(|N|)$ whether $[p] \neq [q]$. The statement then holds by Savitch's Theorem and the Immerman-Szelepscényi Theorem (see, e.g., Section 2.4).

Notice that we will consider the power set automaton of N *with* its useless states in this proof because we want to state a procedure that is correct even if p or q are not states of $\mathcal{P}(N)$. To this end, let $\mathcal{P}(N)' = (Q_{\mathcal{P}(N)'}, \Sigma, \delta_{\mathcal{P}(N)'}, \{q_N^0\}, Q_{\mathcal{P}(N)'})$ be the power set automaton of N (with all useless states). That is, $Q_{\mathcal{P}(N)'}$ is the power set of Q_N and $\delta_{\mathcal{P}(N)'}(q, a) = \{s \mid \exists t \in q \text{ such that } s \in \delta_N(t, a)\}$.

The nondeterministic Turing machine checks whether there exists a Σ-word w such that exactly one of the two states $\delta^*_{\mathcal{P}(N)'}(p, w)$ and $\delta^*_{\mathcal{P}(N)'}(q, w)$ is accepting. If such a word w exists, it implies that the two states p and q are not in the same Myhill-Nerode equivalence class. Towards the test, the Turing machine checks first whether exactly one of the two states p or q is accepting in $\mathcal{P}(N)'$. If this is the case then it accepts immediately. (This situation corresponds to $w = \varepsilon$ above.) Otherwise, it guesses the word $w = a_1 \cdots a_k$ symbol by symbol and stores, for every $i = 1, \ldots, k$, the current states $\delta^*_{\mathcal{P}(N)'}(p, a_1 \cdots a_{i-1})$ and $\delta^*_{\mathcal{P}(N)'}(q, a_1 \cdots a_{i-1})$ on the tape. The machine updates these two states after every new guess of a symbol $a_i \in \Sigma$. Since a state in $\mathcal{P}(N)'$ is a set of states of N the test requires space $O(|N|)$. □

4.1.13 Lemma. BkwEdge, BkwAcceptance, *and* BkwEdge-Cut *for N at level 0 can be decided in space $O(|N|^2)$.*

Proof. We first show how to check, for a given non-empty set $p \subseteq Q_N$, whether $[p]$ is a state in $[N]$, i.e., whether it is useful. Since N only has useful states there exists a path from $[p]$ to some accepting state in $[N]$. Thus, it is enough to check whether there is a path from $[\{q_N^0\}]$ to $[p]$ which can be done by a nondeterministic algorithm working in space $O(|N|)$. The algorithm guesses a word symbol by symbol, simulates $\mathcal{P}(N)$ on-the-fly starting from $\{q_N^0\}$, and tests at each step whether the reached state q is equivalent to p, i.e., if $[p] = [q]$. By Savitch's Theorem and Lemma 4.1.12, this can be done by a deterministic algorithm working in space $O(|N|^2)$.

We now turn our attention to BkwEdge when given $(N, p, q, a, 0)$ as input. We have to decide whether the transition $[p] \xrightarrow{a} [q]$ is present in $[N]$. Since $[N]$ only has useful states we have to show that the following holds in $[N]$:

- $[p]$ and $[q]$ are states in $[N]$, and
- $[\delta_{\mathcal{P}(N)}(p, a)] = [q]$ where $\mathcal{P}(N)$ is the power set automaton of N.

The former can be decided in $O(|N|^2)$ as we have seen above. For the latter notice that, given p and a, we can compute $\delta_{\mathcal{P}(N)}(p, a)$ in space $O(|N|)$. According to Lemma 4.1.12 we can then decide if $[\delta_{\mathcal{P}(N)}(p, a)] = [q]$ in space $O(|N|^2)$.

We now show that BkwAcceptance for N at level 0 can be decided in space $O(|N|^2)$ when given $(N, p, 0)$ as input. We know that $[p]$ is accepting in $\mathrm{lev}_0([N], [p])$ if and only if $[p]$ is a (useful) state in $[N]$ and $p \cap F_N \neq \emptyset$. Using the same arguments as above, the former can be done in space $O(|N|^2)$ and the latter in space $O(|N|)$.

Finally, we focus on BkwEdge-Cut when given $(N, p, q, a, 0)$ as input. Notice that $\mathrm{lev}_0([N], [p])_S = [N]_S$ where S is the set of $[N]$-consistent symbols. We have to decide whether the transition $[p] \xrightarrow{a} [q]$ is present in $[N]_S$ and is not deleted in the S-cut $[N]_S$. We can test whether $[p] \xrightarrow{a} [q]$ is a transition in $[N]$ in space $O(|N|^2)$ by the test for BkwEdge from above. If $[p] \xrightarrow{a} [q]$ is a transition in $[N]$ then we test whether $[p]$ is accepting, which can be done in space $O(|N|^2)$ (as we have seen above). If $[p]$ is accepting then $[p] \xrightarrow{a} [q]$ is a transition in $[N]_S$ if and only if a is not $[N]$-consistent. To decide this we iterate over all states $p' \subseteq Q_N$ of $\mathcal{P}(N)$, test whether $[p']$ is accepting in $[N]$, and whether $[p']$ does not have an a-transition to $[q]$. More precisely, we test, for every state p' of $\mathcal{P}(N)$, whether BkwAcceptance for $[p']$ returns *true* and BkwEdge for $(N, [p'], [q], a, 0)$ returns *false*. As we have seen above BkwEdge and BkwAcceptance can be decided in space $O(|N|^2)$ in this case. This concludes the proof. □

In the following we show that if BkwEdge can be decided at a certain level then also BkwReachability, SameOrbit, and InterOrbit can be decided at this level. This statement also holds for the Cut-variants of these problems.

4.1.14 Lemma. *Assume that* BkwEdge *for N at level k can be decided in space $f(k, |N|)$. Then,*

- BkwReachability *for N at level k can be decided in space $f(k, |N|) + O(|N|^2)$,*
- SameOrbit *for N at level k can be decided in space $f(k, |N|) + O(|N|^2)$, and*
- InterOrbit *for N at level k can be decided in space $f(k, |N|) + O(|N|^2)$.*

Analogously, if BkwEdge-Cut *for N at level k can be decided in space $f(k, |N|)$ then* BkwReachability-Cut, SameOrbit-Cut, *and* InterOrbit-Cut *for N at level k can be decided in space $f(k, |N|) + O(|N|^2)$.*

Proof. We prove the statement for BkwReachability, SameOrbit, and InterOrbit. For the Cut-variants the statement can be proved analogously using BkwEdge-Cut instead of BkwEdge.

We show first how to decide BkwReachability when given (N, p, q, k) as input. The proof is analogous to the one of Savitch's Theorem (which shows that graph reachability is in space $O(\log^2 n)$). The proof of Savitch's Theorem can be sketched as follows. Let G be a graph, x and y be nodes in G, and ℓ be a number. The predicate $\textsc{Path}(G, x, y, \ell)$ denotes whether there is a path from x to y in G of length at most ℓ. It can be computed in space $O(\log^2 |G|)$ by a recursive algorithm that searches for a mid-point on the path. In our case we ask whether there exists a path from $[p]$ to $[q]$ in $\text{lev}_k([N], [p])$. To answer this question we use a very similar procedure as in the proof of Savitch's Theorem. The only difference is that when we need to test whether there is a transition we use the procedure BkwEdge for N at level k. Since the size of $[N]$ is $O(2^{|N|})$ this needs space $f(k, |N|) + O(|N|^2)$.

Next, we show how to decide SameOrbit for N at level k when given (N, p, q, k) as input. We know that $[p]$ and $[q]$ are in the same orbit if and only if BkwReachability is *true* for (N, p, q, k) and for (N, q, p, k), i.e., $[q]$ is reachable from $[p]$ in $\text{lev}_k([N], [p])$ and vice versa. Thus, if BkwReachability for N at level k can be decided in space $f(k, |N|) + O(|N|^2)$ then SameOrbit for N at level k can be decided in space $f(k, |N|) + O(|N|^2)$.

Finally, we show how to decide InterOrbit for N at level k when given (N, p, k) as input. The problem can be decided by enumerating all $a \in \Sigma$ and testing whether $[p]$ and $[\delta_{\mathcal{P}(N)}(p, a)]$ are not in the same orbit, i.e., checking whether SameOrbit returns *false* for these states. Since SameOrbit can be decided in space $f(k, |N|) + O(|N|^2)$ this test requires space $f(k, |N|) + O(|N|^2 + |N| + \log |\Sigma|) = f(k, |N|) + O(|N|^2)$ in total. □

Next, we show that if BkwEdge and BkwAcceptance can be decided at a certain level then also BkwEdge-cut can be decided at this level.

4.1.15 Lemma. *Assume that* BkwEdge *and* BkwAcceptance *for N at level k can be decided in space $f(k, |N|)$ and that $\text{lev}_k([N], [p])$ has no violation. Then* BkwEdge-Cut *for N at level k can be decided in space $f(k, |N|) + O(|N|)$.*

Proof. Let the input for BkwEdge-Cut at level k be (N, p, q, a, k). Then there is a transition $[p] \xrightarrow{a} [q]$ in $\text{lev}_k([N], [p])_{S_k}$ if and only if all of the following hold:

- there is a transition $[p] \xrightarrow{a} [q]$ in $\text{lev}_k([N], [p])$,
- $[p] \xrightarrow{a} [q]$ is not deleted in the S_k-cut, and
- $\text{lev}_k([N], [p])$ has no violation.

By assumption we know that $\text{lev}_k([N], [p])$ has no violation and that we can test whether there is a transition $[p] \xrightarrow{a} [q]$ in $\text{lev}_k([N], [p])$ in space $f(k, |N|)$. In order to check whether the transition is deleted in the S_k-cut we test whether $[p]$ is accepting in $\text{lev}_k([N], [p])$. By assumption this can also be done in space $f(k, |N|)$. Afterwards, we check whether there is a state $p' \subseteq Q_N$ in $[N]$ such that $[p']$ is accepting and does not have an a-transition to $[q]$ in $\text{lev}_k([N], [p])$. Again, both can be decided in space $f(k, |N|)$ by assumption. In total we need space $O(|N|)$ to store p, q, p' and a and space $f(k, |N|)$ to decide BkwEdge and BkwAcceptance. This concludes the proof. □

In Lemmas 4.1.16 to 4.1.18 we prove that we can compute the structure of the automaton at level $k+1$ under the assumption that the automaton at level k is already computed. In this way, the next lemmas formalize a single induction step of our overall proof for DRE-Definability(NFA) is in PSPACE.

4.1.16 Lemma. *Assume that* BkwEdge-Cut *and* BkwAcceptance *for N at level k can be decided in space $f(k, |N|)$ and that $lev_k([N], [p])$ has no violation. Then,* BkwAcceptance *for N at level $k+1$ can be decided in space $f(k, |N|) + O(|N|^2)$.*

Proof. Let $(N, p, k+1)$ be the input for BkwAcceptance. Then $[p]$ is an accepting state in $\mathrm{lev}_{k+1}([N], [p])$ if and only if $\mathrm{lev}_k([N], [p])_{S_k}$ has the orbit property and either

- $[p]$ is accepting in $\mathrm{lev}_k([N], [p])$, or
- $[p]$ has an outgoing inter-orbit transition in $\mathrm{lev}_k([N], [p])_{S_k}$.

Since $\mathrm{lev}_k([N], [p])$ has no violation we know that $\mathrm{lev}_k([N], [p])_{S_k}$ has the orbit property. By Lemma 4.1.14 we can test whether $[p]$ has an outgoing inter-orbit transition in $\mathrm{lev}_k([N], [p])_{S_k}$ in space $f(k, |N|) + O(|N|^2)$. □

4.1.17 Lemma. *Assume that* BkwEdge-Cut *for N at level k can be decided in space $f(k, |N|)$ and that $lev_k([N], [p])$ has no violation. Then,* BkwEdge *for N at level $k+1$ can be decided in space $f(k, |N|) + O(|N|^2)$.*

Proof. Let $(N, p, q, a, k+1)$ be the input for BkwEdge. Then there is a transition $[p] \xrightarrow{a} [q]$ in $\mathrm{lev}_{k+1}([N], [p])$ if and only if all of the following hold:

- there is a transition $[p] \xrightarrow{a} [q]$ in $\mathrm{lev}_k([N], [p])_{S_k}$,
- $[p]$ and $[q]$ are in the same orbit in $\mathrm{lev}_k([N], [p])_{S_k}$, and
- $\mathrm{lev}_k([N], [p])$ has no violation.

By assumption we know that $\mathrm{lev}_k([N], [p])$ has no violation and that we can test whether there is a transition $[p] \xrightarrow{a} [q]$ in $\mathrm{lev}_k([N], [p])_{S_k}$ in space $f(k, |N|)$. By Lemma 4.1.14 we can check whether $[p]$ and $[q]$ are in the same orbit in $\mathrm{lev}_k([N], [p])_{S_k}$ in space $f(k, |N|) + O(|N|^2)$. □

4.1.18 Lemma. *Assume that* BkwEdge *and* BkwAcceptance *for N at level k can be decided in space $f(k, |N|)$ and that $lev_k([N], [p])$ has no violation. Then,* BkwEdge *and* BkwAcceptance *for N at level $k+1$ can be decided in space $f(k, |N|) + O(|N|^2)$.*

Proof. Since BkwEdge and BkwAcceptance for N at level k can be decided in space $f(k, |N|)$ by assumption, BkwEdge-Cut for N at level k can be decided in space $f(k, |N|) + O(|N|)$ by Lemma 4.1.15. Then BkwEdge-Cut and BkwAcceptance for N at level k can be decided in space $f(k, |N|)$ such that BkwAcceptance for N at level $k+1$ can be decided in space $f(k, |N|) + O(|N|^2)$ by Lemma 4.1.16. Finally, since BkwEdge-Cut for N at level k can be decided in space $f(k, |N|)$, BkwEdge for N at level $k+1$ can be decided in space $f(k, |N|) + O(|N|^2)$ by Lemma 4.1.17. □

We now combine the previous lemmas to show that BkwEdge and BkwAcceptance for N (see Lemma 4.1.19) and IsGate-Cut for N (see Lemma 4.1.20) can be decided for an arbitrary level k and in polynomial space in k and the size of N. For technical reasons we need the assumption that all level i ($0 \leq i \leq k-1$) automata $\mathrm{lev}_i([N],[p])$ have no violation. This is because, otherwise, the level k automaton would be empty.

4.1.19 Lemma. *Assume that for $0 \leq i \leq k-1$ all automata $lev_i([N],[p])$ have no violation. Then,* BkwEdge *and* BkwAcceptance *for N at level k can be decided in space $O((k+1)|N|^2)$.*

Proof. To prove the desired upper bound we show that there is a constant $c > 0$ such that BkwEdge and BkwAcceptance for N at level k can be decided using at most space $c(k+1)|N|^2$. By Lemma 4.1.13 we know that there is a constant c_1 such that BkwEdge and BkwAcceptance for N at level 0 can be decided in space $c_1|N|^2$. Similarly, let c_2 be the constant from Lemma 4.1.17 such that BkwEdge and BkwAcceptance for N at level $k+1$ can be decided in space $f(k,|N|)+c_2|N|^2$. Notice that c_2 does not depend on k. We take $c = \max\{c_1, c_2\}$.

The proof is by induction on k. For the base case, $k = 0$, the lemma statement holds by Lemma 4.1.13. Assume that the lemma is true for k, i.e., BkwEdge and BkwAcceptance can be decided for N at level k in space $c(k+1)|N|^2$. By Lemma 4.1.17 BkwEdge and BkwAcceptance for N at level $k+1$ can be decided in space $c(k+1)|N|^2 + c_2|N|^2 \leq c(k+2)|N|^2$. This concludes the proof. □

4.1.20 Lemma. *Assume that, for $0 \leq i \leq k-1$, all automata $lev_i([N],[p])$ have no violation. Then,* IsGate-Cut *for N at level k can be decided in space $O((k+1)|N|^2)$.*

Proof. Let (N,p,k) be the input for IsGate-Cut. Then the state $[p]$ is a gate in $\mathrm{lev}_k([N],[p])_{S_k}$ if and only if $\mathrm{lev}_k([N],[p])$ has no violation and at least one of the following holds:

- $[p]$ is accepting in $\mathrm{lev}_k([N],[p])_{S_k}$, or
- $[p]$ has an outgoing inter-orbit transition in $\mathrm{lev}_k([N],[p])_{S_k}$.

By assumption $\mathrm{lev}_k([N],[p])$ has no violation. By Lemma 4.1.19 BkwAcceptance and BkwEdge for N at level k can be decided in space $O((k+1)|N|^2)$. Thus, InterOrbit for N at level k can be decided in space $O((k+1)|N|^2)$ by Lemma 4.1.14. □

We are now ready to show that we can decide all the necessary subproblems of a level k automaton for arbitrary $k \in \mathbb{N}$ in polynomial space. However, notice that we only need the Cut-variant of the problem IsGate.

4.1.21 Lemma. *Assume that, for $0 \leq i \leq k-1$, all automata $lev_i([N],[p])$ have no violation. Then,* BkwEdge, BkwReachability, SameOrbit, InterOrbit, BkwAcceptance *and* BkwEdge-Cut, BkwReachability-Cut, SameOrbit-Cut, InterOrbit-Cut, *and* IsGate-Cut *for N at level k can be decided in space $O((k+1)|N|^2)$.*

Proof. By Lemma 4.1.19 BkwEdge and BkwAcceptance for N at level k can be decided in space $O((k+1)|N|^2)$. For IsGate-Cut the statement holds by Lemma 4.1.20. Thus we have that, for Lemmas 4.1.14 and 4.1.15, $f(k,|N|) \in O((k+1)|N|^2)$. Therefore, the assumption holds for BkwReachability, SameOrbit, InterOrbit, BkwReachability-Cut, SameOrbit-Cut, InterOrbit-Cut, and BkwEdge-Cut. □

By Lemma 4.1.21 we can decide on-the-fly which transitions are present and which states are accepting in a level k automaton (still under the assumption that no violations occur in levels smaller than k). Since we can compute the entire structure of a level k automaton via these subproblems, we can now also decide whether there is a violation at level k.

4.1.22 Lemma. *Assume that, for $0 \leq i \leq k-1$, all automata $lev_i([N],[p])$ have no violation. Then,* Out-Cons-Violation, Acc-Cons-Violation *and* Orbit-Cons-Violation *for N at level k can be decided in space $O((k+1)|N|^2)$.*

Proof. Let N be the input for Out-Cons-Violation at level k. By definition an out-consistency violation occurs at level k of N if and only if there exist $p, q \subseteq Q_N$ such that the following holds:

- all automata $\mathrm{lev}_i([N],[p])$ for $0 \leq i \leq k-1$ have no violation,
- $[p]$ and $[q]$ are gates in $\mathrm{lev}_k([N],[p])_{S_k}$,
- $[p]$ and $[q]$ are in the same orbit of $\mathrm{lev}_k([N],[p])_{S_k}$, and
- there exist a symbol $a \in \Sigma$ and $[q']$ outside the orbit of $[p]$ such that $[p] \xrightarrow{a} [q']$ but $[q] \overset{a}{\not\to} [q']$ in $\mathrm{lev}_k([N],[p])_{S_k}$.

By assumption all $\mathrm{lev}_i([N],[p])$ for $0 \leq i \leq k-1$ have no violation. Then, IsGate-Cut and SameOrbit-Cut for N at level k can be decided in space $O((k+1)|N|^2)$ by Lemma 4.1.21. The last point can then be decided by enumerating all $a \in \Sigma$ and states $q' \subseteq Q_N$ where for each pair a and q' it is checked whether $[p]$ and $[q']$ are in different orbits and $[p] \xrightarrow{a} [q']$ exists but $[q] \xrightarrow{a} [q']$ does not exist. By Lemma 4.1.21 this can be done in space $O((k+1)|N|^2)$, which concludes the proof for Out-Cons-Violation.

Now, let N be the input for Acc-Cons-Violation at level k. By definition, an Acc-Cons-Violation occurs at level k of N if and only if there exist $p, q \subseteq Q_N$ such that the following holds:

- all automata $\mathrm{lev}_i([N],[p])$ for $0 \leq i \leq k-1$ have no violation,
- $[p]$ and $[q]$ are in the same orbit of $\mathrm{lev}_k([N],[p])_{S_k}$,
- $[p]$ and $[q]$ are gates of $\mathrm{lev}_k([N],[p])_{S_k}$, and
- exactly one of the states $[p]$ and $[q]$ is accepting in $\mathrm{lev}_k([N],[p])_{S_k}$.

By assumption all $\text{lev}_i([N],[p])$ for $0 \leq i \leq k-1$ have no violation. By Lemmas 4.1.21 and 4.1.20 SameOrbit-Cut, IsGate-Cut and BkwAcceptance at level k for N can be decided in $O((k+1)|N|^2)$, which concludes the proof to Acc-Cons-Violation.

Next, let N be the input for Orbit-Cons-Violation at level k. We know an Orbit-Cons-Violation occurs at level k of N if and only if there exists a set $q_1 \subseteq Q_N$ such that the following holds:

- all automata $\text{lev}_i([N],[q_1])$ for $0 \leq i \leq k-1$ have no violation,
- $[q_1]$ is accepting in $\text{lev}_k([N],[q_1])$, and
- for every symbol a, there exists an accepting state $[q_2] \in \mathcal{O}([q_1])$ such that, for every state $[q] \in [N]$, at most one of the transitions $[q_1] \xrightarrow{a} [q]$ and $[q_2] \xrightarrow{a} [q]$ exists in $\text{lev}_k([N],[q_1])$.

By assumption all $\text{lev}_i([N],[q_1])$ for $0 \leq i \leq k-1$ have no violation. By Lemma 4.1.21 SameOrbit, BkwAcceptance and BkwEdge can be decided in space $O((k+1)|N|^2)$. In addition, space $O(|N|)$ is needed to store $[q_1]$, $[q_2]$, $[q]$, and a. This concludes the proof for Orbit-Cons-Violation. □

Finally, we have all the ingredients to prove our main result.

4.1.23 Theorem. DRE-Definability(*NFA*) *and* DRE-Definability(*RE*) *are PSPACE-complete.*

Proof. It is known that DRE-Definability(RE) is PSPACE-hard which was first proved by a reduction from corridor tiling (see, e.g., [vEB97]) in [BGMN09]. A much shorter and less complex proof of this result is given in the thesis of Groz [Gro12]. We illustrate this proof in the following. Groz reduces from Universality for regular expressions. For a regular expression e the reduction constructs the expression $r = \Sigma^*\#(a+b)^*a(a+b) + e\#\Sigma^*$. For this expression, it holds that $L(r)$ is DRE-definable if and only if $L(e) = \Sigma^*$. (Recall that $L((a+b)^*a(a+b))$ is not DRE-definable.) Since an RE can be translated to an equivalent NFA in polynomial time, the lower bound directly holds for NFAs.

We now prove that DRE-Definability for an NFA N is in space $O(|N|^4)$. We assume that $|\Sigma| \leq |N|$ without loss of generality. By Theorem 4.1.11 we know that $L(N)$ is not DRE-definable if and only if a violation occurs at some level k for N. By Theorem 4.1.9 the recursion depth of Algorithm 1 is at most $|N|^2+2$, i.e., only levels $k \in \{0,\ldots,|N|^2+2\}$ are relevant for the problem.

Our PSPACE algorithm checks for violations starting from level 0 on moving to higher levels up to $|N|^2+2$. For every single level k violations can be detected in space $O((k+1)|N|^2)$ by Lemma 4.1.22. Notice that we can apply the above lemma because we know that all smaller levels are checked beforehand and, therefore, do not contain any violation. In summary, DRE-Definability for an NFA N can be tested in space $O((k+1)|N|^2)$ for $k = |N|^2+2$, i.e., in $O(|N|^4)$. The upper bound directly transfers to REs. □

An alternative algorithm for DRE-Definability(NFA) that was obtained independently needs only quadratic space and was presented in [LBC14].

4.1.5 DRE-Definability for RE(#)s and minimal DFAs

We now settle the complexity for the problem DRE-Definability(RE(#)). Afterwards, we prove an NLOGSPACE lower bound for DRE-Definability(*min*DFA).

4.1.24 Theorem. DRE-Definability(RE(#)) *is EXPSPACE-complete.*

Proof. The EXPSPACE upper bound is immediate from Theorem 4.1.23 and the fact that RE(#)s can always be translated to REs that are only exponentially larger by unfolding the counting operators.

We prove the lower bound by a reduction from Universality for RE(#). This problem is known to be EXPSPACE-complete which was proved by Meyer and Stockmeyer. In [MS72] they showed that Universality for regular expressions with squaring (i.e., regular expressions that only allow counters $(2, 2)$) is EXPSPACE-complete. The reduction for RE(#) is analogous to the reduction for RE in [Gro12] (see also Theorem 4.1.23) which shows that DRE-Definability(RE) is PSPACE-hard. More precisely, it constructs, given an RE(#) e, the expression $r = \Sigma^*\#(a + b)^*a(a + b) + e\#\Sigma^*$ where # is a symbol that is not used in the language of e. To prove correctness of the reduction recall that $L((a + b)^*a(a + b))$ is not DRE-definable. Thus, $L(r)$ is DRE-definable if and only if $L(e) = \Sigma^*$. □

As mentioned before, DRE-Definability can be solved in polynomial time when the input is a minimal DFA [BKW98]. In [LBC14] it was shown that DRE-Definability is in NLOGSPACE when the input is a minimal DFA over an alphabet that is of size at most logarithmic in the number of states in the DFA. We prove that the problem is NLOGSPACE-hard. The precise complexity for arbitrary regular languages remains open.

4.1.25 Theorem. DRE-Definability(*minDFA*) *is NLOGSPACE-hard.*

Proof. The proof is by a LOGSPACE reduction from the complement of the Reachability problem for directed acyclic graphs, which is known to be NLOGSPACE-complete [Jon75]. For our reduction we use that Reachability for directed acyclic graphs is already NLOGSPACE-complete when every node in the graph has outdegree at most two. We restrict ourselves to such instances because we aim to construct a reduction over an alphabet of fixed size. Moreover, the reduction relies on the fact that finite languages are always DRE-definable [BKW98]. Since NLOGSPACE coincides with coNLOGSPACE this proves the assumption.

Let $G = (V, E)$ be an acyclic graph where every node in V has at most outdegree two and let x and y be nodes in V. The Reachability problem decides whether y is reachable from x by a directed path. In our reduction we construct a minimal DFA D such that $L(D)$ is DRE-definable if and only if node y is reachable from node x in G.

The minimal DFA D is defined as the tuple $(Q, \Sigma, \delta, q_0, \{q_{|V|}\})$ where $Q = V \uplus \{q_0, q_1, \ldots, q_{|V|}\}$ is the finite set of states (i.e., each node in V is associated with a state in Q and, additionally, D has new states $q_0, q_1, \ldots, q_{|V|}$), q_0 is the initial state, $q_{|V|}$ is the only accepting state, and $\Sigma = \{a, b, c, d, e\}$ is the finite alphabet. Moreover, let $V = \{v_1, \ldots, v_n\}$ be the nodes of G. Then D contains the following transitions:

- $\delta(v_i, a) = v_j^1$ and $\delta(v_i, b) = v_j^2$ for each node $v_i \in V$ and edges $(v_i, v_j^1) \in E$ and $(v_i, v_j^2) \in E$ (if v_i has only one outgoing edge then D has only one a-labeled v_i-outgoing transition, and if v_i has no outgoing edges then D has no v_i-outgoing transitions),
- $\delta(q_0, a) = v_i$ and $\delta(v_i, c) = q_i$ for each node v_i,
- $\delta(q_i, d) = q_{i+1}$ for each node q_i with $0 < i < |V|$, and
- $\delta(y, e) = x$.

This concludes our reduction. Notice that it is computable in LOGSPACE. Next, we prove that the reduction is correct. Therefore, we prove that the following holds for the minimal DFA D:

(1) D is a minimal DFA, and

(2) $L(D)$ is DRE-definable if and only if y is not reachable from x in G.

We prove (1) first. By definition D is a DFA. To see that D is minimal observe that, for every pair q and q' of states in D, it holds that either $\delta(q, cd^i) = q_{|V|}$ and $\delta(q', cd^i) \neq q_{|V|}$, or $\delta(q, d^i) = q_{|V|}$ and $\delta(q', d^i) \neq q_{|V|}$, for some $i \in \{0, \ldots, |V|\}$. Thus, there do not exist two states that are in the same Myhill-Nerode equivalent class of D. This proves that D is minimal.

Concerning (2) we prove first that the implication from right to left holds. Assume that there is no path from x to y in G. Towards a contradiction, assume also that there exists a cycle in D. The cycle cannot contain one of the states $q_0, \ldots, q_{|V|}$ because q_0 has no incoming transitions, every state q_i reaches only states q_j with $j > i$, and $q_{|V|}$ has no outgoing transitions. Hence, the cycle can only use transitions that originate from the edges of G and the transition from y to x. Moreover, the cycle has to contain the transition from y to x since the graph G is acyclic (and the structure of the graph is maintained in D). Thus, the rest of the cycle forms a path from x to y, which contradicts our assumption. It follows that D has to be acyclic, which means that $L(D)$ has to be finite. Since every finite language is DRE-definable, the implication from right to left holds.

Second, we show that the implication from left to right holds. Towards a contradiction, assume that $L(D)$ is DRE-definable but y is reachable from x in G. Then, x and y are in the same orbit of D. This orbit cannot contain any of the states $q_1, \ldots, q_{|V|}$ because these states cannot reach a state v_i in D. Therefore, the transitions (x, c, q_i) and (y, c, q_j) with $i \neq j$ are inter-orbit transitions and, hence, x and y are gates of the same orbit. However, these transitions are not out-consistent which implies that D does not fulfill the orbit property. By Theorem 3.3.7, $L(D)$ is not DRE-definable. This contradicts our assumption and concludes the proof. □

Notice that, in contrast to our proof in [CDLM13], we provide here a reduction that constructs a minimal DFA that works over an alphabet of fixed size (instead of an alphabet of size $|Q|^2$).

4.2 Descriptional Complexity of Deterministic Regular Expressions

In this section we consider the relative descriptional complexity of REs, DREs and DFAs. It is well-known that an exponential blow-up cannot be avoided when translating REs to minimal DFAs in the worst case [HMU13]. Ehrenfeucht and Zeiger [EZ76] proved that there also are minimal DFAs that are exponentially more succinct than every equivalent RE. In [GH08, GJ08] the authors exhibited certain characteristics of automata which make equivalent regular expressions large. However, these results cannot naïvely be transferred to DREs because the languages that are used to prove lower bounds on the size of REs (see, e.g., [EZ76, GH08, GJ08]) are not definable by DREs. Therefore, we have to find new languages and techniques to prove similar bounds for DRE-definable languages. An overview of our results on the descriptional complexity of DRE-definable languages is shown in Table 4.2.

RE	DRE	DFA	case exists?	reference
$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	yes	Obs. 4.2.1
$\Theta(n)$	$2^{\Omega(n)}$	$2^{\Omega(n)}$	yes	Cor.4.2.6
$2^{\Omega(n)}$	$2^{\Omega(n)}$	$\Theta(n)$	?	
$\Theta(n)$	$2^{\Omega(n)}$	$\Theta(n)$	yes	Cor.4.2.7

Table 4.2: Relative descriptional complexity for DRE-definable languages [LMN12]. In the table, n is the size of a minimal representation for a given language. If a case exists, then there exist infinitely many DRE-definable languages such that the minimal REs, DREs, and DFAs for these languages exhibit the specified complexity.

Since every DRE is an RE, we know that every minimal RE for a language L is smaller or equal to a minimal DRE for L. Moreover, Brüggemann-Klein and Wood showed that, given a DRE r, one can construct a DFA D for $L(r)$ with size $O(|\Sigma||r|)$ [BK93]. Therefore, the table contains all substantial cases that ought to be considered. However, one special case that is not in the table is whether there are minimal REs that are double-exponentially more succinct than their equivalent minimal DREs. More precisely, this means that for such REs there would exist an exponential blow-up when translating them to a minimal DFA and another exponential blow-up when translating the DFA to a minimal DRE. Whether there exist DRE-definable languages that exhibit such a behavior is still an open problem.

Before we prove the results from the table above, we briefly examine the relative descriptional complexity of REs, DFAs, and DREs for finite languages. Since every finite language is DRE-definable (see, e.g., [BGMN09]), all known lower bounds on the descriptional complexity for DFAs and REs for finite languages transfer to DREs (see Table 4.3).

RE	DRE	DFA	case exists?	reference
$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	yes	Obs. 4.2.1
$\Theta(n)$	$2^{\Omega(n)}$	$2^{\Omega(n)}$	yes	[Man73, BKW98]
$2^{\Omega(n)}$	$2^{\Omega(n)}$	$\Theta(n)$	no	[EKSW04]
$\Theta(n)$	$2^{\Omega(n)}$	$\Theta(n)$	yes	Th.4.2.4
$n^{\Omega(\log n)}$	$n^{\Omega(\log n)}$	$\Theta(n)$	yes	[GJ08]

Table 4.3: Relative descriptional complexity for finite languages. In the table, n is the size of a minimal representation for a given language. If a case exists, then there exist infinitely many finite languages such that the minimal REs, DREs, and DFAs for these languages exhibit the specified complexity.

In Sections 4.2.1 and 4.2.2, we present the proofs for the results from Tables 4.2 and 4.3. Before, we start with a trivial observation which shows that there are languages that do not cause any significant blow-up between the different representations.

4.2.1 Observation. There exists a family of finite languages $(L_n)_{n\in\mathbb{N}}$ and a family of infinite languages $(L'_n)_{n\in\mathbb{N}}$ such that, for each $n \in \mathbb{N}$, the minimal DFAs, minimal REs, and minimal DREs for L_n and L'_n have size $\Theta(n)$.

To see that the observation is true consider, for each $n \in \mathbb{N}$, the finite language $L(a^n)$ and the infinite language $L((a^n)^*)$.

4.2.1 DREs for Finite Languages

We briefly review known results on the descriptional complexity of finite languages. In 1973, Mandl showed that there is a family of finite languages such that a minimal RE for a language in this family is exponentially more succinct than the equivalent minimal DFA. Brüggemann-Klein and Wood showed that, for these languages, a minimal RE is also exponentially more succinct than an equivalent minimal DRE.

4.2.2 Theorem ([Man73, BKW98]). *Let $L_n = L((a+b)^{0,n}a(a+b)^n)$ for every $n \in \mathbb{N}$. Then, every minimal RE for L_n has size $\Theta(n)$, the minimal DFA for L_n has size $2^{\Omega(n)}$, and every minimal DRE for L_n has size $2^{\Omega(n)}$.*

Ellul et al. [EKSW04] showed that, for every finite language with a DFA (or even NFA) D of size n, there exists an RE for $L(D)$ of size $n^{O(\log n)}$. Gruber and Johannsen showed that this upper bound is also tight [GJ08].

4.2.3 Theorem ([EKSW04, GJ08]).

- *Let D be a DFA of size n and let $L(D)$ be finite. Then, there exists an RE r for $L(D)$ such that $|r| \leq n^{O(\log n)}$.*
- *There exists a family of finite languages $(L_n)_{n\in\mathbb{N}}$ such that, for each $n \in \mathbb{N}$, the minimal DFA for L_n has $\Theta(n)$ states but every minimal RE for L_n has size $n^{\Theta(\log n)}$.*

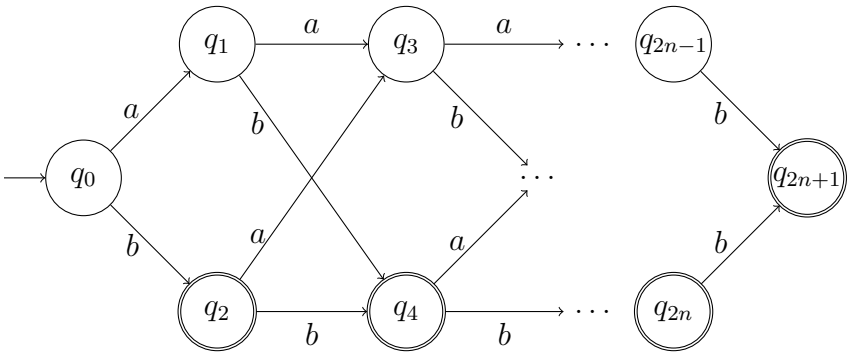


Figure 4.5: Minimal DFA for the language L_n.

We prove that there are infinitely many finite languages $(L_n)_{n\in\mathbb{N}}$ such that every minimal RE and the minimal DFA for L_n are exponentially more succinct than a minimal DRE for L_n. The following theorem was not published before and solves an open case in [LMN12].

4.2.4 Theorem. *There exists a family of finite languages $(L_n)_{n\in\mathbb{N}}$ such that, for each $n \in \mathbb{N}$, every minimal RE for L_n has size $\Theta(n)$, the minimal DFA for L_n has size $\Theta(n)$, and every minimal DRE for L_n has size $2^{\Omega(n)}$.*

Proof. To prove the assumption we consider the family $(L_n)_{n\in\mathbb{N}}$ where

$$L_n = \{(a+b)^{0,n} \cdot b \mid n \geq 1\}, \text{ for every } n \in \mathbb{N}.$$

For every n, the regular expression $(a+b)^{0,n}\cdot b$ is equivalent to the regular expression $(a+b+\varepsilon)\cdots(a+b+\varepsilon)\cdot b$ where the subexpression $(a+b+\varepsilon)$ appears n times. Observe that the latter expression has size $6n+1$. Moreover, every regular expression that is equivalent to L_n has to be at least of size n since L_n is finite and contains a word of length n. The minimal DFA for L_n is shown in Figure 4.5 and has size $4n$.

Assume that r_n is a minimal DRE for L_n. We show that r_n has at least size 2^n. The proof is by induction on n. For the base case, $n = 0$, the assumption holds because $L_0 = L(b)$ and, thus, $|r_0| \geq 1$.

Now, assume that r_{n-1} has at least size 2^{n-1}. For the induction step, we show that

$$r_n = a \cdot r_{n-1} + b \cdot (\varepsilon + r_{n-1}).$$

Towards a contradiction, assume that r_n has a concatenation operation as topmost operation in its syntax tree, i.e., $r_n = s_1 \cdot s_2$ for some DREs s_1 and s_2. Then, we distinguish two cases depending on whether $\varepsilon \in L(s_1)$ or not.

If $\varepsilon \notin L(s_1)$ then $\text{first}(s_1) = \{a, b\}$. Since $b \in L_n$ it follows that $\varepsilon \in L(s_2)$ and, thus, that every word in $L(s_1)$ ends with b by definition of L_n. Moreover, we know that $s_2 \neq \varepsilon$ because r_n is minimal. Let ub be a longest word in $L(s_1)$ and vb be a longest word in $L(s_2)$ such that $ubvb \in L_n$. By the structure of L_n it follows that also $uavb \in L_n$. Since ua and vb are of maximal length for s_1 and s_2, respectively, we have that $ua \in L(s_1)$. It follows that also $ua \in L(r_n)$ (because $\varepsilon \in L(s_2)$) which contradicts the assumption that $L(r_n) = L_n$.

If $\varepsilon \in L(s_1)$ then it holds that $\varepsilon \notin L(s_2)$. Since $b \in L(r_n)$ and r_n is a DRE it follows that $\text{first}(s_1) = \{a\}$ and $\text{first}(s_2) = \{b\}$. Let bw be a longest word in $L(r_n)$, then we know that $bw \in L(s_2)$. Since $s_1 \neq \varepsilon$ (because r_n is minimal) bw is not a longest word in L_n, which contradicts our assumption. This proves that r_n is not a concatenation but has to be a disjunction $s_1 + s_2$. (Recall that the expression r_n cannot be an expression $(s_1)^*$ because $\varepsilon \notin L_n$.)

Since r_n is a DRE and $\varepsilon \notin L_n$, the expression r_n has to be of the form $a \cdot s_1 + b \cdot s_2$ for some DREs s_1 and s_2. Moreover, we have that $a\backslash L_n = L_{n-1}$ such that $s_1 = r_{n-1}$. This proves that $r_n = a \cdot r_{n-1} + b \cdot s_2$.

It remains to prove that, for $L(s_2) = b\backslash L_n = L(r_{n-1} + \varepsilon)$, every minimal DRE for the language $L_n \cup \{\varepsilon\}$ is of the form $r_n^\varepsilon = r_n + \varepsilon$ with $L(r_n) = L_n$.

Towards a contradiction, assume that r_n^ε has a concatenation operation as topmost operation in its syntax tree, i.e., $r_n^\varepsilon = s_1 \cdot s_2$. Then, we know that $\varepsilon \in L(s_1)$ and $\varepsilon \in L(s_2)$. Thus, every word ($\neq \varepsilon$) in $L(s_1)$ and every word ($\neq \varepsilon$) in $L(s_2)$ has to end with b. Let ub be a longest word in $L(s_1)$ and vb be a longest word in $L(s_2)$ such that $ubvb \in L(r_n^\varepsilon)$. By the structure of $L_n \cup \{\varepsilon\}$ it follows that also $uavb \in L(r)$ such that $ua \in L(s_1)$ and $ua \in L(r_n^\varepsilon)$. This contradicts the assumption that $L(r_n^\varepsilon) = L_n \cup \{\varepsilon\}$.

Therefore, r_n^ε has to be a disjunction $s_1 + s_2$. (The expression r_n^ε cannot be an expression $(s_1)^*$ because $L_n \cup \{\varepsilon\}$ is finite and $L_n \neq \{\varepsilon\}$.) It remains to prove that s_1 or s_2 equals ε. Towards a contradiction, assume that $\text{first}(s_1) = \{a\}$ and $\text{first}(s_2) = \{b\}$ without loss of generality. (Recall that $\text{first}(s_1) \cap \text{first}(s_2) \neq \emptyset$ is not allowed since r_n^ε is a DRE.) Furthermore, we know that $\varepsilon \in L(s_1)$ or $\varepsilon \in L(s_2)$. Without loss of generality, let $\varepsilon \in L(s_1)$. This directly contradicts our assumption since it is not possible to write a DRE for $L(s_1)$ with $\text{first}(s_1) = \{a\}$ and $\varepsilon \in L(s_1)$ such that s_1 is not a disjunction $s_1' + \varepsilon$. This concludes the proof. □

4.2.2 DREs for Infinite Languages

It is well-known that an exponential blow-up is unavoidable when translating between REs and DFAs for infinite regular languages in the worst case.

4.2.5 Theorem ([HMU13, EZ76]).

- *For each $n \in \mathbb{N}$ the minimal DFA for $L((a+b)^*a(a+b)^n)$ has size $2^{\Theta(n)}$.*
- *There is a family of infinite regular languages $(L_n)_{n\in\mathbb{N}}$ such that, for each $n \in \mathbb{N}$, the minimal DFA for L_n has size $\Theta(n^2)$ and every minimal RE for L_n has size $2^{\Omega(n)}$.*

To prove that there exists an unavoidable exponential blow-up when translating a minimal RE for an infinite DRE-definable language to a minimal DFA in the worst case, we extend the finite languages of Theorem 4.2.2 to infinite languages.

4.2.6 Corollary. *Let $L_n = L((a+b)^{0,n}a(a+b)^n\#^*)$ for every $n \in \mathbb{N}$. Then, every minimal RE for L_n has size $\Theta(n)$, the minimal DFA for L_n has size $2^{\Omega(n)}$, and every minimal DRE for L_n has size $2^{\Omega(n)}$.*

To prove that there exists an unavoidable exponential blow-up when translating the minimal DFA for an infinite DRE-definable language to a minimal DRE in the worst case, we extend the finite languages of Theorem 4.2.4 to infinite languages. In this way, we provide a much shorter and less complex proof of this result than compared to the proof in [Los10, LMN12].

4.2.7 Corollary. *Let $L_n = \{(a+b)^{0,n} \cdot b \cdot \#^* \mid n \geq 1\}$. For each $n \in \mathbb{N}$, every minimal RE for L_n has size $\Theta(n)$, the minimal DFA for L_n has size $\Theta(n)$, and every minimal DRE for the language has size $2^{\Omega(n)}$.*

Our initial motivation to study the descriptional complexity of DRE-definable languages was an unproven claim in [BKW98] which states that, for expressions of the form $\Sigma^* w$ where $w \in \Sigma^*$, every equivalent DRE is at least exponential in the length of w. This claim was proved in the master thesis of the author [Los10] and its proof is published in [LMN12]. Proving the claim turned out to be rather non-trivial. Since such languages have polynomial-size REs and DFAs, we needed to develop new techniques for proving lower bounds on the size of DREs. In [LMN12] we presented a general technique to show lower bounds on the descriptional complexity of DREs. The main idea of this technique is to identify concatenations of a minimal DRE in a minimal DFA. To this end, we search for so-called *bottleneck states* which are states that every accepting run needs to visit. Using these bottlenecks, the following results have been obtained in [LMN12]. Let $L_n^{\text{bottle}} = L((b + ab + \cdots + a^n b)^* a^n)$ for every $n \in \mathbb{N}$.

4.2.8 Lemma. *For each $n > 0$ there exists a minimal DRE for the language L_n^{bottle} that contains at least 2^n concatenations.*

4.2.9 Theorem. *For each $n \in \mathbb{N}$ every minimal RE for L_n^{bottle} has size $\Theta(n)$, the minimal DFA for L_n^{bottle} has size $\Theta(n)$, and every minimal DRE for L_n^{bottle} has size $2^{\Omega(n)}$.*

Although the proof in [LMN12] is much more complex than the proof of Theorem 4.2.4, we believe that both proofs are valuable because they both provided additional insights into different aspects of the descriptional complexity of DRE-definable languages.

4.3 Descriptional Complexity of Operations on DRE-Definable Languages

In the following we study several language-theoretic operations on DRE-definable languages. Our study is motivated by the use of DREs in XML schema languages where the obtained knowledge about the descriptional complexity of operations on DREs can be useful for, e.g., merging schemas or schema optimization.

In Section 4.3.1, we briefly review results on the state complexity of DFAs that are the result of applying a boolean operation on DRE-definable languages. Since

DREs can always be efficiently translated to a DFA the state complexity for DFAs provides lower bounds on the descriptional complexity for DREs as well. Previously, the state complexity of boolean operations on DFAs has been studied in [KW80, Man73, PS02, SY97, Yu01]. In [PS02] the focus has been on unary languages. The results in [Yu01] that involve only finite languages can be directly transferred to DFAs for DRE-definable languages.

In Section 4.3.2, we study the descriptional complexity of several language-theoretic operations on DREs, i.e., we study the complement, intersection, union, reversal, and concatenation operation on DREs. Our results show that the restriction to deterministic regular expressions does not significantly decrease the descriptional complexity in comparison to the class of all regular expressions. Previously, the descriptional complexity of the complement and intersection operation for REs has been independently examined by Gelade and Neven [GN08, GN12] and Gruber and Holzer [GH09]. They show that, in the worst case, the size of a minimal RE for the intersection of a fixed number of REs is exponentially larger than the input and that the size of a minimal RE for the complement of an RE is double-exponentially larger than the input. Moreover, they prove that these bounds are tight. Gelade and Neven also study these operations on DREs. They prove that, in the worst case, the size of a minimal RE for the intersection of a fixed number of DREs can be exponentially larger than the input and that the complement of a DRE can always be written as an RE that is only polynomially larger than the input DRE. However, in their proofs the languages of the resulting REs are not DRE-definable. (Recall that DRE-definable languages are not closed under any boolean operation see, e.g., Section 3.3.2). The concatenation and reversal operation on regular languages has been studied in [CCSY01, JJS05, Jir08, SWY04, YZS94]. In [YZS94] the authors also examine languages over unary alphabets.

4.3.1 Boolean Operations on DFAs

Next, we review known results on the state complexity of minimal DFAs that define a DRE-definable language and are the result of applying an operation on some minimal DFAs for DRE-definable languages. We consider this problem for DFAs over arbitrary and unary alphabets. In each case we also consider a single use of a boolean operation as well as its k-fold application. The results are summarized in Tables 4.4 and 4.5.

	$\lvert\Sigma\rvert = 1$		$\lvert\Sigma\rvert \geq 1$	
	1	k	1	k
$\setminus$	$\Theta(m)$ [HMU13]	—	$\Theta(m)$ [HMU13]	—
$\cap$	$\Theta(\min\{m_1, m_2\})$ [Yu01]	$\Theta(\min\{m_0, ..., m_k\})$ [Yu01]	$\Theta(m_1 m_2)$ [Yu01]	$2^{\Omega(k)}$ [Los10, LMN12]
$\cup$	$\Theta(\max\{m_1, m_2\})$ [Yu01]	$\Theta(\max\{m_0, ..., m_k\})$ [Yu01]	$\Theta(m_1 m_2)$ [Yu01]	$2^{\Omega(k)}$ [Los10, LMN12]

Table 4.4: State complexity of minimal DFAs for finite languages. For every operation, 1, 2 or $k+1$ minimal DFAs with $m, m_0, \ldots, m_k$ states are given as input. The table entries specify the number of states of the minimal DFA for the resulting language in the worst case. By "$\setminus$" we denote the complement $\Sigma^* \setminus L$ for a given language L.

	$\|\Sigma\| = 1$		$\|\Sigma\| \geq 1$	
	1	k	1	k
$\setminus$	$\Theta(m)$ [HMU13]	—	$\Theta(m)$ [HMU13]	—
$\cap$	$\Theta(m_1 m_2)$ [Los10, LMN12]	$k^{\Omega(k)}$ [Los10, LMN12]	$\Theta(m_1 m_2)$ [Los10, LMN12]	$k^{\Omega(k)}$[Los10, LMN12]
$\cup$	$\Theta(\max\{m_1 m_2\})$[Los10, LMN12]	$\Theta(\max\{m_0, ..., m_k\})$ [LMN12]	$\Theta(m_1 m_2)$ [Los10, LMN12]	$2^{\Omega(k)}$[Los10, LMN12]

Table 4.5: State complexity of minimal DFAs for infinite DRE-definable languages. For every operation, $1, 2$ or $k+1$ minimal DFAs with $m, m_0, \ldots, m_k$ states are given as input. The table entries specify the number of states of the minimal DFA for the resulting language in the worst case. By "$\setminus$" we denote the complement $\Sigma^* \setminus L$ for a given language L.

It is well-known that there is no blow-up for the complement operation on DFAs [HMU13]. This result holds for finite as well as for infinite languages.

Regarding the results for finite languages in [Yu01] notice the following: For the union and intersection of two finite languages Yu proved an $m_1 m_2$ upper and lower bound. Nevertheless, only the result for the upper bound has been stated in the paper since in [Yu01] the focus was on the exact state complexity, which depends on the following problem: Given two DFAs for finite languages, is it possible that the minimal DFA for the union (intersection, respectively) has exactly $m_1 m_2$ states? To see that this is in fact not possible observe that even the product automaton of two DFAs for finite languages cannot have exactly $m_1 m_2$ states. For example, let s be the initial state of one automaton and q be a non-initial state of the other automaton, then the state (s, q) can neither be part of the product for the union nor for the intersection of the two automata. As far as we know, the exact state complexity of the union or intersection of two finite languages is still open. In [Los10, LMN12] we proved an exponential lower bound for the k-fold application of the intersection and union operation on finite languages over an arbitrary alphabet.

For infinite DRE-definable languages the state complexity for boolean operations on DFAs is not lower than for all regular languages in general (see Table 4.5). We only found one case in which the state complexity is strictly lower than for regular languages, namely when considering DFAs for the union of DRE-definable languages over unary alphabets.

4.3.2 Operations on DREs

In this section we investigate the descriptional complexity of DREs that are itself the result of applying an operation on some DREs. We study the complement, intersection, union, reversal, and concatenation operation on DREs. An overview of our results is illustrated in Table 4.6. The results in Table 4.6 hold for DRE-definable languages over alphabets with at least two symbols. The results without a reference have not been published before. Another result that we present in the following is that, for DRE-definable languages over unary alphabets, the descriptional complexity of DREs for operations is basically the same as the state complexity for their equivalent minimal DFAs.

To this end, we prove that, for a minimal DFA D for a DRE-definable language over a unary alphabet, there always exists an equivalent DRE that has linear size in $|D|$. For the proof we exploit that DFAs for languages over unary alphabets have

	input (one or two) DREs of size $O(n)$
$\setminus$	$2^{\Omega(n)}$ (Th. 4.3.5)
$\cap$	$2^{\Omega(n)}$ (Th. 4.3.6)
$\cup$	$2^{\Omega(n)}$ (Th. 4.3.7)
rev	$2^{\Omega(n)}$ [Los10, LMN12]
$\cdot$	$2^{\Omega(n)}$ [Los10, LMN12]

Table 4.6: Descriptional complexity of several operations on DREs over an alphabet Σ with $|\Sigma| \geq 2$. The table entries specify the size of a minimal DRE for the resulting language in the worst case. By "$\setminus$" we denote the complement $\Sigma^* \setminus L$ for a given language L. and by "rev" the reversal operation.

a very restricted form. The following notation has been introduced in Pighizzini and Shallit [PS02]: A DFA with state set $Q = \{q_0, \ldots, q_{n+m}\}$ where q_0 is the initial state is a *chain followed by a cycle* if its transition function is of the form $\delta(q_0, a) = \{q_1\}$, $\ldots$, $\delta(q_{n-1}, a) = \{q_n\}$, $\delta(q_n, a) = q_{n+1}$, $\ldots$, $\delta(q_{n+m-1}, a) = q_{n+m}$, $\delta(q_{n+m}, a) = q_n$, where $q_i \neq q_j$ for $i \neq j$. Moreover, at least one of the states in $\{q_n, \ldots, q_{n+m}\}$ is an accepting state. We refer to the states $q_n, \ldots, q_{n+m}$ as the *cycle states* of this DFA. (Notice that Pighizzini and Shallit referred to a *tail* instead of a *chain*.)

4.3.1 Lemma ([PS02]). *Every minimal DFA for an infinite regular language over a unary alphabet is a chain followed by a cycle.*

The next result can be obtained by applying Theorem 3.3.7 on Lemma 4.3.1.

4.3.2 Corollary. *An infinite regular language L over a unary alphabet is DRE-definable if and only if it has exactly one accepting cycle state.*

In the following we show that, for every DRE-definable language L over a unary alphabet, the size of a minimal DRE for L is linear in the size of the minimal DFA for L.

4.3.3 Theorem. *Let L be a DRE-definable language over a unary alphabet and D be a minimal DFA for L with m states. Then, there exists a minimal DRE r for L such that r is of size $O(m)$.*

Proof. Let L be a DRE-definable language over the alphabet $\Sigma = \{a\}$ and let D be the minimal DFA for L with m states. If L is equal to $L(\varepsilon)$ or $L(\emptyset)$ then the assumption holds. Otherwise, we distinguish two cases depending on whether L is finite or infinite.

If L is finite then D is a chain with some accepting states. Let $Q = \{q_0, \ldots, q_{m-1}\}$ be the states of D and $F \subseteq Q$ be the set of accepting states. We now that $q_{m-1} \in F$ since D is a minimal DFA and contains only useful states. We construct a DRE r for $L(D)$ as follows:

$$r = r_0 \text{ where } r_j = \begin{cases} a \cdot r_{j+1} & \text{if } q_j \notin F \text{ and } j < m-1, \\ (a \cdot r_{j+1} + \varepsilon) & \text{if } q_j \in F \text{ and } j < m-1, \\ \varepsilon & \text{if } j = m-1. \end{cases}$$

If L is infinite then D is a chain and a cycle where exactly one cycle state is accepting. Let $Q = \{q_0, \ldots, q_{m-1}\}$ be the states of D such that $q_n, \ldots, q_{m-1}$ are the cycle states of D. Let q_i be the accepting state in the cycle of D. (Notice that D can have more accepting states in the chain.) We construct a DRE r for $L(D)$ as follows:

$$r = r_0 \text{ where } r_j = \begin{cases} a \cdot r_{j+1} & \text{if } q_j \notin F \text{ and } j < i, \\ (a \cdot r_{j+1} + \varepsilon) & \text{if } q_j \in F \text{ and } j < i, \\ (a^{m-n})^* & \text{if } j = i. \end{cases}$$

By a straightforward induction it holds that r is a DRE for L of size $O(m)$. □

In the remainder of this section we present the proofs of the results that are illustrated in Table 4.6. For the reversal operation the result can be obtained using the language $L((a+b)^{0,n}a(a+b)^n)$ for $n \in \mathbb{N}$ [Los10, LMN12]. For the concatenation operation the result can be obtained using the languages $L_n^1 = (a+b)^{0,n}$ and $L_n^2 = a(a+b)^n$ for $n \in \mathbb{N}$ [Los10, LMN12].

Before we can prove the exponential blow-up for the complement operation, we need the following auxiliary result on minimal DREs. Recall that a regular language L is *prefix-free* if and only if, for every word $v \in L$, there does not exist a word $z \in \Sigma^*$ such that $v \cdot z \in L$.

4.3.4 Lemma. *Let $L_a = L \cdot \{a\}$ be a prefix-free DRE-definable language. Then, there exists a minimal DRE for L_a that is either a or of the form $r \cdot a$ for some regular expression r.*

Proof. The proof is by structural induction on a minimal DRE s for L_a. For the base case, $s = a$, the assumption holds. (Notice that L_a cannot be $L(\emptyset)$ or $L(\varepsilon)$ by definition.)

Let the assumption hold for DREs s_1 and s_2. The expression s cannot be a star expression $(s_1)^*$ since L_a does not contain ε.

Assume that $s = s_1 + s_2$. By the structure of L_a we know that $L(s_1) = L_1 \cdot \{a\}$ and $L(s_2) = L_2 \cdot \{a\}$ for some some languages L_1 and L_2 such that $L_1 \cdot \{a\}$ and $L_2 \cdot \{a\}$ are prefix-free DRE-definable languages. By induction hypothesis it follows that, for every $i = \{1, 2\}$, there exists a minimal DRE for $L(s_i)$ of the form a or $r_i \cdot a$. This implies that s is of the form $(a+a)$, $(r_1 \cdot a + r_2 \cdot a)$, or $(r_i \cdot a + a)$ where $i \in \{1, 2\}$. In each case, there exists an expression for $L(s)$ that is of the required form and of the same size or more succinct, i.e., a, $(r_1 + r_2) \cdot a$, or $(r_i + \varepsilon) \cdot a$, respectively.

Assume that $s = s_1 \cdot s_2$. We know that $\varepsilon \notin L(s_2)$ because L_a is prefix-free. Thus, $L(s_2)$ is of the form $L_1 \cdot \{a\}$ for some language L_1 such that $L_1 \cdot \{a\}$ is DRE-definable and prefix-free. By induction hypothesis there is a minimal DRE for $L(s_2)$ that is of the form a or $r_1 \cdot a$. Since s is a DRE we know that $\text{followlast}(L(s_1)) \cap \text{first}(L(s_2)) = \emptyset$. Thus, there exists a minimal DRE for L_a that is of the form $s_1 a$ or $s_1 r_1 a$, which concludes the proof. □

Now, we are ready to prove that complementing a DRE can cause an unavoidable exponential blow-up (when the result is represented as a minimal DRE).

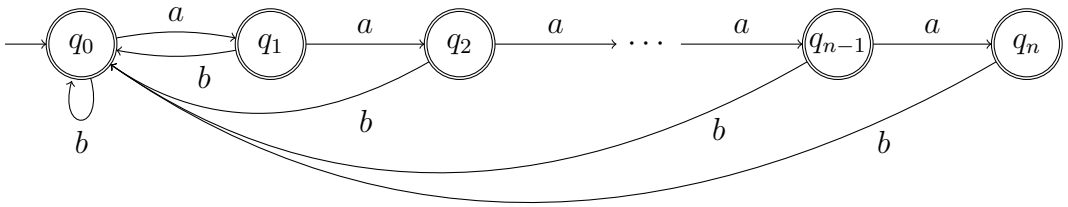


Figure 4.6: Minimal DFA for the language L_n^C from Theorem 4.3.5.

4.3.5 Theorem. *There exists a family of DRE-definable languages $(L_n)_{n\in\mathbb{N}}$ such that, for each $n \geq 1$, every minimal DRE for L_n has size $O(n)$ and every minimal DRE for $\overline{L_n} = \Sigma^* \setminus L_n$ has size $2^{\Omega(n)}$.*

Proof. We prove the assumption by showing that, for each $n \geq 1$, the language L_n^C (see Figure 4.6) has a minimal DRE of size $O(n)$ and that the size of every minimal DRE for the language $\overline{L_n^C} = \Sigma^* \setminus L_n^C$ is at least 2^n. Intuitively, the language L_n^C contains all words over the alphabet $\Sigma = \{a, b\}$ that do not contain the subword a^n. The regular expression

$$r_n = \underbrace{(\varepsilon + a(\varepsilon + a(\cdots)))}_{n \text{ times}} \cdot (b \cdot \underbrace{(\varepsilon + a(\varepsilon + a(\cdots)))}_{n \text{ times}})^*$$

is a DRE for L_n^C that has size $O(n)$.

Now, let L_n^{bottle} be the language from Theorem 4.2.9 and observe that

$$\overline{L_n^C} = L_n^{\text{bottle}} \cdot L(a \cdot (a + b)^*).$$

We show that every minimal DRE for $\overline{L_n^C}$ is of the form $\overline{r_n} \cdot a(a+b)^*$ where $L(\overline{r_n}) = L_n^{\text{bottle}}$. Then, the assumption holds by Theorem 4.2.9.

Let $\overline{r}$ be a minimal DRE for $\overline{L_n^C}$. We show first that $\overline{r}$ is a concatenation. Since there exists a word of length two in $\overline{L_n^C}$, the expression $\overline{r}$ is not atomic (i.e., a Σ-symbol, ε, or $\emptyset$). Since $\varepsilon \notin \overline{L_n^C}$, the expression $\overline{r}$ is not a star expression.

Towards a contradiction, assume that $\overline{r}$ is a disjunction $r_1 + r_2$. Since $\varepsilon \notin \overline{L_n^C}$ and $\overline{r}$ is a DRE we know that $\text{first}(r_1) = \{a\}$ and $\text{first}(r_2) = \{b\}$ without loss of generality. Using the same arguments as above r_1 and r_2 cannot be atomic or star expressions. Since $\varepsilon \notin \overline{L_n^C}$, $\overline{r}$ is a DRE, and $|\,\text{first}(r_1)| = \text{first}(r_2)| = 1$, the expressions r_1 and r_2 cannot be disjunctions themselves. Thus, r_1 and r_2 have to be concatenations. Moreover, r_1 and r_2 end with a star expression because they are DREs and for every word $w \in L(\overline{r})$ there is a word $wv \in L(\overline{r})$ with $v \neq \varepsilon$. Hence, they are of the form $r_1' \cdot (r_1'')^*$ and $r_2' \cdot (r_2'')^*$, respectively.

Towards a contradiction, assume that $(r_1'')^*$ is not equivalent to $(a + b)^*$. Then, consider words $w \in L(r_1)$ and $v \notin L((r_1'')^*)$. Observe that $wv \in \overline{L_n^C}$ because $w \in L(\overline{r})$ and $wv' \in \overline{L_n^C}$ for every $v' \in \Sigma^*$. However, $wv \notin L(\overline{r})$ because $\overline{r}$ is a DRE. This contradicts the assumption that $\overline{r}$ is a DRE for $\overline{L_n^C}$. Thus, $\overline{r} = r_1' \cdot (a+b)^* + r_2' \cdot (a+b)^*$ which directly contradicts that $\overline{r}$ is minimal. Therefore, $\overline{r}$ is a concatenation.

Finally, it remains to prove that $\overline{r}$ cannot be represented more succinct than in $\overline{r_n} \cdot a(a+b)^*$. Using the same arguments as above we get that every minimal DRE for $\overline{r}$ is of the form $r' \cdot (a+b)^*$. Then, every minimal DRE is of the form $\overline{r_n} \cdot a(a+b)^*$ by Lemma 4.3.4, which concludes the proof. □

Next, we prove that the intersection and union operation on DREs can cause an unavoidable exponential blow-up (when the result is represented as a minimal DRE).

4.3.6 Theorem. *There exist families of DRE-definable languages $(L_n^1)_{n\in\mathbb{N}}$ and $(L_n^2)_{n\in\mathbb{N}}$ such that, for each $n \geq 1$, minimal DREs for L_n^1 and L_n^2 have size $O(n)$ and every minimal DRE for $L_n^1 \cap L_n^2$ has size $2^{\Omega(n)}$.*

Proof. To prove the assumption, we show that the language

$$L_n = \{(a+b)^{0,n} \cdot b\} \text{ with } n \geq 1$$

can be expressed as the intersection of two DREs of size $O(n)$. By Theorem 4.2.4 we know that the size of every minimal DRE for L_n is at least 2^n. Consider the languages

$$L_n^1 = L((a^*b)^*) \text{ and } L_n^2 = L((a+b)^{1,n+1}).$$

The expression $(a^*b)^*$ is a DRE for L_n^1 of size 5. L_n^2 is finite and, therefore, DRE-definable. Furthermore, $(a+b)(\varepsilon + (a+b)(\varepsilon + \cdots))$ is a DRE for L_n^2 of size $O(n)$.

It remains to show that $L_n = L_n^1 \cap L_n^2$. We first prove that $L_n \subseteq L_n^1 \cap L_n^2$. Let w be a word in L_n. Then, we know that w is a word over $\{a, b\}$ that always ends with b and has at most length $n+1$. Thus, w is in L_n^1 and L_n^2.

Second, we prove that $L_n \supseteq L_n^1 \cap L_n^2$. Let w be a word in $L_n^1 \cap L_n^2$. Then, w has at least length one and at most $n+1$ because $w \in L_n^2$. Moreover, w ends with b because $w \in L_n^1$. Thus, $w \in L_n$ which proves the assumption. □

4.3.7 Theorem. *There exist families of DRE-definable languages $(L_n^1)_{n\in\mathbb{N}}$ and $(L_n^2)_{n\in\mathbb{N}}$ such that, for each $n \geq 1$, minimal DREs for L_n^1 and L_n^2 have size $O(n)$ and every minimal DRE for $L_n^1 \cup L_n^2$ has size $2^{\Omega(n)}$.*

Proof. To prove the assumption we show that the language L_n^{inf} with $n \geq 1$ from Figure 4.7 can be expressed as the union of two DREs of size $O(n)$ and that the size of every minimal DRE for L_n^{inf} is at least 2^n. Consider the languages

$$L_n^1 = L((a^*b)^*) \text{ and } L_n^2 = L((a+b)^{n+2}(a+b)^*).$$

The expression $(a^*b)^*$ is a DRE for L_n^1 of size 5. L_n^2 is finite and, therefore, DRE-definable. Furthermore, a DRE for L_n^2 that has size $O(n)$ can be obtained from the expression $(a+b)^{n+2}(a+b)^*$ by unfolding the counters.

Next, we show that $L_n^{\text{inf}} = L_n^1 \cup L_n^2$. First, we prove that $L_n^{\text{inf}} \subseteq L_n^1 \cup L_n^2$. Let w be a word in L_n^{inf}. If $|w|$ is smaller or equal to $n+1$ then w ends with b and, therefore, $w \in L_n^1$; otherwise $w \in L_n^2$.

Second, we prove that $L_n^{\text{inf}} \supseteq L_n^1 \cup L_n^2$. Let w be a word in $L_n^1 \cup L_n^2$. If w ends with b then $w \in L_n^{\text{inf}}$. If w ends with an a then $|w|$ is greater than $n+1$. Thus, $w \in L_n^{\text{inf}}$.

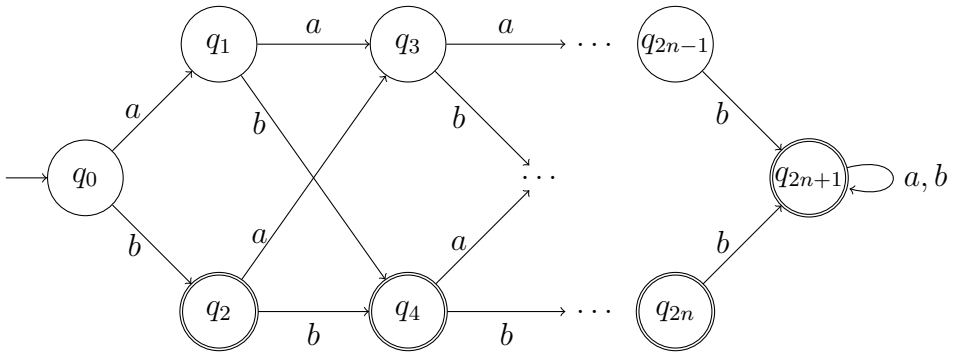


Figure 4.7: Minimal DFA for the language L_n^{inf} from Theorem 4.3.7.

It remains to prove that every minimal DRE for the language L_n^{inf} has at least size 2^n. Analogously to the the proof of Theorem 4.3.5 every minimal DRE for L_n^{inf} is of the form $r \cdot (a + b)^*$ such that $L(r) = L_n$ where L_n is the language from Theorem 4.2.4. This concludes the proof. □

5

Querying RDF Data using SPARQL Property Path Expressions

In this chapter we examine regular expressions used as a query language for graph-structured data. Such query languages have been investigated in the database community for more than a decade, often under the name of regular path queries or general path queries [AQM+97, BDHS96, CM90, CMW87, FFLS00, Yan90, LY02]. Various problems for regular path queries have been studied, such as optimization [AV99], query rewriting [CGLV02], query answering using views [CGLV00b], and query containment [CGLV00a, DT01, FLS98]. Recently, there has been a renewed interest in path queries on graphs, for example, on path queries using expressions with data value comparisons [LV12, LTV13, KRV14] or using certain XPath dialects [LMV13].

We examine variants of regular expressions that are used in the SPARQL query language. According to the working draft from January 2012, SPARQL 1.1 queries (see Section 9 in [HS12]) can contain *property paths* which are defined as regular expressions including counting operators (#), restricted negation operators (!), and wildcard symbols (•). Property paths without counting operators have been studied in [PAG10, ABE09, ACP12]. As far as we know we are the first to study the complexity of full SPARQL property paths (including counting operators). Most closely related to the results in this chapter is [ACP12] which complements them in several aspects. A detailed discussion on the results in [ACP12] can be found in Section 5.7. Further results on the complexity of SPARQL query evaluation can be found in [PAG09, SML10, AP11, LRV13].

In the following we study two kinds of semantics for property path expressions: *simple walk semantics* and *regular path semantics*. Simple walk semantics is our formalization of the W3C's intended semantics for property paths, whereas regular

path semantics is our alternative suggestion for SPARQL semantics. We will see that regular path semantics exhibits a better computational complexity than simple walk semantics for several problems.

In our abstract framework we model SPARQL property paths as RE($\#, !, \bullet$) expressions over an infinite set of symbols and RDF graphs as directed labeled s-t graphs. In particular, we investigate the complexity of the following three problems that are central for SPARQL query processing. Notice that it depends on the chosen semantics (i.e., simple walk or regular path semantics) whether a path "matches" a regular expression.

Problem (Evaluation).
Given: A graph (V, E, x, y) and a regular expression r.
Question: Is there a path from x to y that matches r?

Problem (Counting).
Given: A graph (V, E, x, y), a regular expression r, and a number max in unary.
Question: How many different paths of length at most max between x and y match r?

Problem (Finiteness).
Given: A graph (V, E, x, y) and a regular expression r.
Question: Are there only finitely many different paths from x to y that match r?

As it may be clear that Evaluation is pivotal for SPARQL query evaluation, it is important to consider Counting and Finiteness because of the SPARQL counting requirement. This requirement demands that all correct answers to a query have to be counted. To get a fair comparison between simple walk and regular path semantics we also study the complexity of Finiteness. (Later, we will see that the result for a query under simple walk semantics is always finite. For a query under regular path semantics, however, this is not the case.)

Moreover, we parameterize the problems with the kind of regular expressions or automata that we consider. For example, when we talk about Evaluation for RE($\#, \neg$), we mean the Evaluation problem where the input is a graph (V, E, x, y) and an expression r in RE($\#, \neg$).

Before we study the complexity of these problems, we examine a cumbersome detail about regular expressions in SPARQL, i.e., the fact that the expressions are built over the theoretically infinite set of IRIs (see Section 2.5.2). In Section 5.1, we formally define such expressions and introduce slightly different automata that can handle words over a potentially infinite set of symbols. We mark these automata with a superscript w, e.g., NFA^w, DFA^w, and UFA^w. However, we remark that all results that are presented in this chapter do not depend on an infinite set of symbols and can be proved for finite alphabets analogously. In Section 5.2, we define the two above mentioned semantics for evaluating regular expressions over graphs. Therefore, we examine particular parts of the SPARQL specification that contain the definition of the W3C intended simple walk semantics.

In Section 5.3, we study the EVALUATION problem. As an introduction to the problem we illustrate a practical study of SPARQL query engines and their efficiency. The lack of efficiency of these engines motivated us to study the semantics of SPARQL queries in more detail. In contrast to the deficient results of the studied SPARQL engines we prove that EVALUATION under regular path semantics remains tractable under combined query evaluation complexity, even when counting operators are added to regular expressions. Our algorithm is cubic in the size of the graph and will be presented in Section 5.3.1. We prove in Section 5.3.2 that EVALUATION under regular path semantics cannot remain tractable when full-fledged negation is allowed. In Section 5.3.3, we illustrate that EVALUATION under simple walk semantics is already NP-complete for the regular expression $(aa)^*$ which holds directly by results in [LP84, MW95]. Moreover, notice that this result makes the problem NP-complete even under data complexity. We also identify a fragment of expressions for which EVALUATION under simple walk semantics remains in P, though for this fragment simple walk semantics coincides with regular path semantics.

The picture is even more drastic for the COUNTING problem. For regular path semantics we provide a detailed study of the tractability frontier. COUNTING can be solved in polynomial time when the considered expressions can be translated to a DFA^w in polynomial time. As such, we know that COUNTING is in P for *unambiguous* expressions (see Section 3.3). However, even for expressions with a very limited amount of nondeterminism beyond unambiguity, COUNTING is #P-complete. For simple walk semantics the situation goes from bad to worse. COUNTING under simple walk semantics is already #P-complete for the regular expression a^*. Basically, this shows that, as soon as the Kleene star operator is available, COUNTING is #P-complete under simple walk semantics. Tractable fragments for COUNTING under simple walk semantics we found only in the set of fragments for that simple walk semantics coincides with regular path semantics. The COUNTING problem is closely related to two problems studied in the literature: (1) counting the number of words of a given length in the language of a regular expression and (2) counting the number of paths in a graph that satisfy certain constraints. We chose to represent the number *max* in unary because this was the case in previous related work on (1) and (2) (e.g., [KSM95, AJ93, Val79b]). Furthermore, such an encoding only strengthens our hardness results. We remark that our polynomial-time results for COUNTING can be transferred to the setting where the number *max* is given in binary (Theorems 5.4.6 and 5.3.8).

An overview of our results in combined complexity is presented in Table 5.1. One result, which is not in the table but may be of independent interest, is on the word membership problem for regular expressions with counting operators and negation. In Theorem 5.3.4 we prove that this problem is in P. Another result that is not in the table is that the complexity of EVALUATION under regular path semantics remains the same if we extend our expressions by a *nesting* operator, thereby obtaining a variant of *nested regular expressions* (see, e.g., [PAG10]). This result is presented in Section 5.6.

Our results in data complexity are presented in Table 5.2. For regular path semantics all considered problems that are #P-hard for combined complexity are

Problem	Fragment	Regular path semantics	Simple walk semantics
EVALUATION	CHARE($(+a)^*$,$(+a)^+$, $(+w)$,$(+w)?$)	in P	**in P** (5.3.8)
	star-free RE	in P	**in P** (5.3.9)
	$(aa)^*$	in P	NP [LP84, MW95]
	RE, RE($\#, !, \bullet$)	**in P** (5.3.2)	**NP** (5.3.7)
	RE($\neg$)	**non-elem.** (5.3.6)	—
	RE($\#, !, \neg, \bullet$)	**non-elem.** (5.3.6)	—
COUNTING	DFA^w, UFA^w	**in FP** (5.4.6)	—
	CHARE(a, $(+a)$)	**in FP**	**in FP** (5.4.8)
	a^+, a^*	**in FP**	#P (5.4.8, [Val79b])
	Det-RE	**in FP**	**#P**
	CHARE($a, a?$)	**#P** (5.4.6,5.4.7)	**#P** (5.4.8)
	CHARE(a, a^*), CHARE($a, (+a^+)$), CHARE($a, (+a)^+$), CHARE(a, w^+), CHARE($(+a), a^+$), RE	**#P** (5.4.6,5.4.7)	**#P**
	RE($\#, !, \bullet$)	**#P** (5.4.7)	**#P** (5.4.8)
	RE($\#, !, \neg, \bullet$)	**#P** (5.4.7)	—
FINITENESS	RE, RE($\#, !, \bullet$)	**in P** (5.5.1)	—
	RE($\neg$), RE($\#, !, \neg, \bullet$)	**non-elem.** (5.5.3)	—

Table 5.1: An overview of our results in combined complexity. The results printed in bold are new and published in [LM13]. All complexities are completeness results, unless stated otherwise. The entries marked by "—" signify that the question is either trivial or not defined. We annotated new results with the relevant theorem numbers. If no such number is provided, it means that the result directly follows from other entries in the table.

tractable under data complexity, i.e., when the query is fixed we can always translate it to a DFA^w and perform the algorithm for DFA^{w}s. For simple walk semantics all NP-hardness or #P-hardness results also hold under data complexity, except for the CHARE($a, a?$) fragment. Indeed, if a CHARE($a, a?$) expression is fixed then we can translate it to a DFA^w and perform the algorithm for COUNTING under regular path semantics. (For this fragment simple walk semantics coincides with regular path semantics.) Thus, the difference between regular path semantics and simple walk semantics is rather severe under data complexity.

Finally, we remark that Arenas et al. independently provided similar results on the intractability of SPARQL 1.1 queries in [ACP12]. Together we obtained strong arguments to adapt the current SPARQL query specification. In Section 5.7, we discuss the progress that has been made in the SPARQL specification since then.

Problem	Fragment	Regular path semantics	Simple walk semantics
EVALUATION	CHARE($(+a)^*$,$(+a)^+$, $(+w)$,$(+w)?$)	in P	in P
	star-free RE	in P	in P
	$(aa)^*$	in P	NP [LP84, MW95]
	RE, RE$(\#, !, \neg, \bullet)$	in P	NP
COUNTING	DFA^w, UFA^w, NFA^w	in FP	—
	CHARE$(a, (+a))$, CHARE$(a, a?)$	in FP	in FP
	a^+, a^*	in FP	#P [Val79b]
	RE, RE$(\#, !, \neg, \bullet)$	in FP	#P
FINITENESS	RE, RE$(\#, !, \neg, \bullet)$	in P	—

Table 5.2: An overview of our results when interpreted under data complexity [LM13].

5.1 Modelling RDF data using IRIs

When querying RDF data some data points in the graph refer to an *Internationalized Resource Identifier* (IRI). An IRI is a unique resource name which is specified by the user (see Section 2.5.2 for details). As such IRIs are not fixed and can be added to the considered data at any time. Therefore, the languages that are defined by SPARQL property paths are created over a possibly infinite but countable set of symbols (i.e., IRIs). We denote this set as Δ. Although the results in this chapter do not depend on an infinite set of symbols, we choose to model alphabet and queries according to the RDF specification. To this end, we have to adapt the definitions of regular expressions and automata.

Let Δ be an infinite set of symbols. We define *regular expressions over* Δ analogous to Section 2.1 and Chapter 3 using Δ instead of Σ. For simplicity, we overload notation in this chapter and refer to different classes of regular expressions via the shortcuts that are defined in Chapter 3 (for example, we refer to RE$(\#)$ and RE$(\#, !, \bullet)$). However, all regular expressions that are considered in this chapter are defined over a (possibly) infinite set of symbols.

According to the working draft from January 2012, SPARQL 1.1 queries can contain the following regular expressions (see Section 9.1 in [HS12]).

5.1.1 Definition. Let Δ be an infinite set of symbols. The set of *SPARQL regular expressions* (or *SPARQL property paths*) is the set RE$(\#, !, \bullet)$ over Δ.

Notice that an infinite set of symbols makes the definition of the restricted negation operator (!) interesting in the first place. Moreover, we assume that we can test for equality between elements of Δ in constant time such that we can still test membership of a word and a regular expression in linear time.

Next, we adapt our notion of finite word automata to finite automata that read Δ-words. The automata behave very similarly to standard finite automata (as they have been introduced in Section 2.1) but they can make use of a wildcard symbol "$\circ$"

to deal with the infinite set of labels. To handle this new symbol the transition function of these automata is defined over a finite subset Σ of Δ and over the symbol $\circ$ (where we assume that $\circ$ is not a symbol in Σ).

5.1.2 Definition. A *nondeterministic finite automaton with wildcards* NFA^w N is a tuple $(Q, \Sigma, \Delta, \delta, q_0, F)$ where Q is a finite set of states, $\Sigma \subseteq \Delta$ is a finite alphabet, Δ is a (possibly) infinite set of input symbols, δ is the transition function with signature $Q \times (\Sigma \uplus \{\circ\}) \to \mathcal{P}(Q)$, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states.

The semantics of an NFA^w N is defined as follows. A *run of N on a Δ-word* $w = a_1 \cdots a_n$ is a sequence $r = q_0 q_1 \cdots q_n$ such that, for every $i = 1, \ldots, n$,

- $q_i \in \delta(q_{i-1}, a_i)$ if $a_i \in \Sigma$, and
- $q_i \in \delta(q_{i-1}, \circ)$ if $a_i \notin \Sigma$.

If $q_n \in F$ then the run is *accepting*. A Δ-word w is *accepted by N* if there exists an accepting run of N on w. By $L(N)$ we denote the *language of N*, i.e., the set of all Δ-words accepted by N.

5.1.3 Remark. When the NFA^w is in a state q and reads a symbol $a \in \Delta$, we distinguish two cases. If $a \in \Sigma$ then the transitions labeled with Σ-symbols can be followed. If $a \notin \Sigma$ then the outgoing transitions labeled with the $\circ$-symbol can be followed to the target state. Thus, the $\circ$-label in outgoing transitions is used to deal with every symbol that is not specified by the finite alphabet Σ. Notice that the semantics of the $\circ$-label in automata is therefore different from the $\bullet$-symbol in regular expressions. We choose to model the semantics of automata in this way because of two reasons: first, we are able to define a natural notion of determinism for automata reading Δ-words and, second, our definition of $\circ$ makes it easy to handle subexpressions of the form $!(a_1 + \cdots + a_n)$ in automata. Opposed to that one could also implement these goals by defining the semantics of $\circ$ to be "all symbols for which the current state has no other outgoing transition". In our opinion, the current definition is more transparent since the semantics of every $\circ$-transition is defined equally across the whole automaton. In addition, every RE($\bullet$) can still be translated to an equivalent NFA^w in polynomial time.

Analogously to our definition of NFA^{w}s we adapt the notions of unambiguity and determinism for automata.

5.1.4 Definition. An *unambiguous finite automaton with wildcards* (or a UFA^w) is an NFA^w N where, for every Δ-word $w \in L(N)$, there exists exactly one accepting run of N on w.

5.1.5 Definition. A *deterministic finite automaton with wildcards* (or a DFA^w) is an NFA^w where δ is a (partial) function with signature $Q \times (\Sigma \uplus \{\circ\}) \to Q$.

Finally, we adapt the definition of Glushkov automata for RE(!, •) expressions (see Definition 3.3.12). Using this definition we are able to formally define unambiguity and determinism for regular expressions RE(!, •) over an infinite set of symbols Δ. For an RE(!, •) r we define its numbered expression analogous to Section 3.3.3. For example, for $r = a\ !(a) \bullet (a + bc)^* \bullet\ !(a + b)$ we have $\text{num}(r) = 1\ 2\ 3\ (4\ +5\ 6)^*7\ 8$. Recall that by denum_r we denote the mapping that maps each number i to the subexpression it replaced in r. For the above example, we have that $\text{denum}_r(1) = a$, $\text{denum}_r(2) =!(a)$, $\text{denum}_r(3) = \bullet$, et cetera.

5.1.6 Definition. Let r be an RE(!, •) over Δ, r_m its numbered expression, and $G_r = (Q_r, \Sigma_r, \Delta, \delta_r, q_0, F_r)$ be the Glushkov automaton of r. The *Glushkov automaton with wildcards* G_r^ω of r is the tuple $(Q_r, \Sigma_r, \Delta, \delta_r^\omega, q_0, F_r)$ where δ_r^ω contains all transitions of δ_r and, additionally, the transitions

- $\delta_r(q_0, \circ) = \{i \in \text{first}(r_m) \mid \text{denum}(i) = \bullet \text{ or } \text{denum}(i) = !(a_1 + \cdots + a_\ell) \text{ for some } a_1, \ldots, a_\ell\}$, and
- $\delta_r(i, \circ) = \{j \in \text{follow}(r_m, i) \mid \text{denum}(j) = \bullet \text{ or } \text{denum}(j) = !(a_1 + \cdots + a_\ell) \text{ for some } a_1, \ldots, a_\ell\}$.

Similarly to Corollary 3.3.13 for finite alphabets, the following proposition holds for SPARQL regular expressions over a possibly infinite set of symbols.

5.1.7 Proposition. *Let Δ be an infinite set of symbols. For every RE(!, •) r over Δ the Glushkov automaton with wildcards G_r^ω of r can be constructed in polynomial time in $|r|$. Moreover, it holds that $L(r) = L(G_r^\omega)$.*

Finally, we use the notions of unambiguity and determinism for RE(!, •) over Δ analogously to Definition 3.3.14. That is, we say that r is *unambiguous* if G_r^ω is a UFA^w and r is *deterministic* (or a $DRE(!, \bullet)$) if G_r^ω is a DFA^w.

5.2 Alternative Semantics for Property Paths

In this section we formally define the semantics for SPARQL property path expressions. More precisely, we define two different semantics for SPARQL property path expressions which we examine independently in the following. The first semantics, called *regular path semantics*, is analogous to the semantics of regular expressions that is given in Section 3.1. Since SPARQL expressions are evaluated over paths p in a graph we interpret the semantics over path labels $\text{lab}(p)$ rather than over words. However, we only consider finite paths and, therefore, every label $\text{lab}(p)$ can be seen as a Δ-word.

5.2.1 Definition. Let r be a regular expression, N be a finite automaton, and p be a path in a directed labeled graph. Then,

- *p matches r under regular path semantics* if $\text{lab}(p) \in L(r)$, and
- *p matches N under regular path semantics* if $\text{lab}(p) \in L(N)$.

The second semantics is called *simple walk semantics*. This semantics is our formalization of the semantics that are defined for property paths in the SPARQL working draft of January 2012 [HS12]. In the SPARQL algebra (see Section 18.4 of [HS12]) the semantics of subexpressions of the form r^* and r^+ is defined through the operators **ZeroOrMorePath** and **OneOrMorePath**, respectively. These operators are specified by the following definitions which come from [HS12] and read as follows:

> **ZeroOrMorePath**: An arbitrary length path $P = (X\ (\mathbf{path})^*\ Y)$ is all solutions from X to Y by repeated use of **path** such that any nodes in the graph are traversed once only. ZeroOrMorePath includes X.

> **OneOrMorePath**: An arbitrary length path $P = (X\ (\text{path})^+\ Y)$ is all solutions from X to Y by repeated use of path such that any nodes in the graph are traversed once only. This does not include X, unless repeated evaluation of the path from X returns to X.

In these definitions X and Y are RDF terms (or variables) of the query, i.e., they bind to nodes of the graph in the answer. By **path** they denote an arbitrary subexpression. It is assumed that the semantics of **path** is already known to the user. Since the precise intentions of these definitions are veiled to us, we try to formalize our interpretations of them (and the SPARQL specification) in the following. On the one hand, the definition of **ZeroOrMorePath** seems to require that the subexpression **path** is matched only by paths from X to Y in the input graph that contain each node at most once. That is, subexpressions are allowed to be matched only by simple paths (see Definition 2.2.1). On the other hand, in the definition of **OneOrMorePath** it seems that subexpressions are allowed to be matched by paths that return to X at the end. That means, subexpressions are allowed to be matched by simple paths and simple cycles (see Definition 2.2.1). For this reason, it is unclear to us whether the informal definition of the W3C allows simple cycles or not. However, examples in the working draft suggest that simple cycles are allowed. Therefore, we choose to answer this question positive and consider simple walks in the presentation of our proofs. (Remember that a simple walk is either a simple path or a simple cycle.) In particular, we assume that the following constraint holds for the evaluation of property paths under the SPARQL 1.1 specification.

5.2.2 Note. Property paths that are evaluated under the SPARQL 1.1 specification are required to fulfill the *simple walk requirement*, that is, subexpressions of the form r^* and r^+ should be matched by simple walks in RDF graphs.

In the following we formally define *simple walk semantics* as an abstraction of the definitions of the W3C working draft [HS12] from January 2012. Therefore, recall that we did not define r^+ as an abbreviation of the expression rr^* in Section 2.1 since r^+ and rr^* have different semantics in the following.

5.2.3 Definition. Let $p = v_0[a_1]v_1[a_2]v_2 \cdots v_{n-1}[a_n]v_n$ be a path in a graph and r be an RE$(\#, !, \bullet)$ over Δ. Then p *matches* r *under simple walk semantics* if r and p fulfill the following properties:

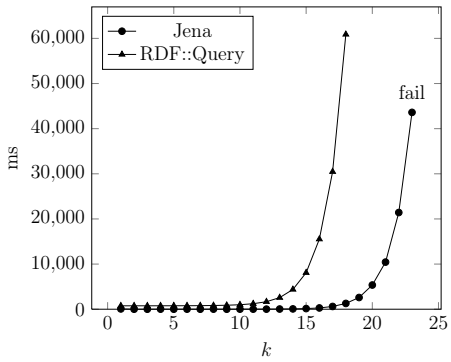
structure of r	properties of a path p that matches r
$r = \emptyset$, $r = \varepsilon$, $r = \bullet$, $r = a, a \in \Delta$, $r = !(a_1 + \cdots + a_n)$	$\text{lab}(p) \in L(r)$
$r = s^*$, $r = s^+$	$\text{lab}(p) \in L(r)$ and p is a simple walk
$r = s?$	$p = v_0$ or p matches s under simple walk semantics
$r = s_1 \cdot s_2$	$p = p_1 \cdot p_2$ and path p_i matches s_i under simple walk semantics for every $i = 1, 2$.
$r = s_1 + s_2$	p matches s_1 or s_2 under simple walk semantics
$r = s^{k,\ell}, \ell \neq \infty$	$p = p_1 \cdots p_m$ with $k \leq m \leq \ell$ and path p_i matches s under simple walk semantics for every $i = 1, \ldots, m$. (Notice that if $\ell = 0$ then p has length zero.)
$r = s^{k,\infty}$	$p = p_1 \cdot p_2$ where p_1 matches $s^{k,k}$ under simple walk semantics and p_2 matches s^* under simple walk semantics

If it is clear from the context we only say that a path *matches* a regular expression and omit a reference to the concrete semantics. We added the case $r = \emptyset$ for compatibility with regular expressions. To provide some intuition for simple walk semantics we remark the following. Notice that we no longer have that a^* is equivalent to a^*a^* under simple walk semantics. We neither have that $a^{1,\infty}$ is equivalent to a^+ nor that aa^* is equivalent to a^+ under simple walk semantics. However, aa^* is still equivalent to $a^{1,\infty}$ under simple path semantics. For counting operators, like in an expression $(a+b)^{5,7}$, simple walk semantics is defined in exactly the same way as regular path semantics. Moreover, observe that we do not say that a path matches a finite automaton under simple walk semantics since it is unclear how simple walk semantics transfers to automata.

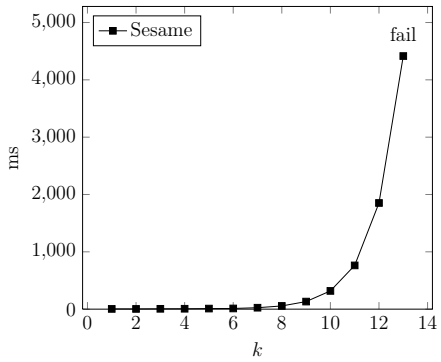
Finally, we remark that all complexity results presented in this chapter also hold if one considers simple paths instead of simple walks.

5.3 The Complexity of the Evaluation Problem for Property Paths

Before we study the complexity of SPARQL 1.1 queries in a formal framework, we illustrate the results of a practical study on the efficiency of SPARQL engines for property path evaluation. We conducted our study in November 2011 and evaluated the most prevalent SPARQL query engines that support property paths at that time: the Jena Semantic Web Framework [Apa11], Sesame [KFB08], RDF::Query [Wil], and Corese 3.0 [Cor12].



(a) Evaluation time for Jena and RDF::Query.



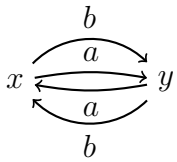
(b) Evaluation time for Sesame.

Figure 5.1: Time taken by Jena, Sesame and RDF::Query for evaluating the expression $(a + b)^{1,k}$ for increasing values of k on a graph with two nodes and four edges [LM13].

The four frameworks were tested with the query

```
ASK WHERE { :x (a|b){1,k} :y }
```

for increasing values of $k \in \mathbb{N}$ on the graph



consisting of two nodes and four labeled edges. An ASK-query in SPARQL returns a boolean value that is *true* if and only if there exists at least one answer for the query in the graph. Answering this query therefore corresponds to solving the EVALUATION problem on the above graph for the expression $(a + b)^{1,k}$. Moreover, notice that, for every $k \in \mathbb{N}$, the answer to this query is *true* and that the query result under regular path semantics is same as under simple walk semantics.

The performance of three of the four systems is depicted in Figure 5.1. The results are obtained from an evaluation on a desktop PC with 2 GB of RAM. For the Jena and Sesame framework the points in the graph depict all the points for that we could obtain data. When we increased the number k by one, the systems ran out of memory and failed (as shown in the figure). Our conclusion from our measurements is that all three systems seem to exhibit a double-exponential behavior: from a certain point, whenever we increase the number k by one (which does not mean that one more bit is needed to represent it), the processing time doubles. The fourth SPARQL query engine, namely Corese 3.0, evaluated queries of the above form very quickly. However, when we asked the engine to answer the query

```
ASK WHERE {x ((a|b)/(a|b)){1,k} y},
```

which basically searches for the existence of even length paths, its time consumption was as bad as for the other three systems. In contrast to the other three systems, Corese did not run out of memory so quickly.

In a related study by Arenas, Conca, and Pérez [ACP12] a similar double-exponential behavior is also observed for `SELECT` queries using property paths. In general `SELECT` queries are more difficult to answer than `ASK` queries because `SELECT` queries should output all tuples that witness the query, whereas an `ASK` query simply asks if there exists such a tuple or not. However, the queries for which [ACP12] observed an double-exponential behavior did not exploit counting operators. As such, the experiments here and in [ACP12] seem to complement each other.

In Section 5.3.1, we show that the double-exponential behavior that we observed in practice can be improved to polynomial-time combined complexity.

5.3.1 Evaluation under Regular Path Semantics

We present a polynomial-time algorithm for the Evaluation problem for SPARQL regular expressions under regular path semantics. Previously, it has already been shown that Evaluation is in P for standard regular expressions without counting operators (see, e.g., [MW95, AV99, ABE09]). In this case, the problem is very similar to testing intersection non-emptiness of two finite word automata. In order to see this, one has to interpret the input graph G with given nodes x and y as a finite nondeterministic automaton N_G. In this automaton the nodes of G are considered as states and the edges as transitions. Accordingly, x is the initial state and y an accepting state in N_G. The expression r is converted to an NFA N_r in polynomial time by using standard methods. Finally, it holds that there is a path from x to y in G that matches r if and only if the intersection of $L(N_G)$ and $L(N_r)$ is not empty, which can be tested in polynomial time.

In this section we prove that Evaluation is also in P when the expressions are allowed to use counting operators, wildcards, and restricted negation. To this end, we construct a polynomial-time algorithm for Evaluation of RE$(\#, !, \bullet)$-expressions which follows a dynamic programming approach. The connection between dynamic programming and regular expressions goes back at least to 1956 when Kleene used recursive formulas to extract a regular expression from a DFA [Kle56]. In addition, dynamic programming is also used to test membership for regular expressions see, e.g., [HMU13]. In [KT03], Kilpeläinen and Tuhkonen adapted this approach to test membership for RE$(\#)$ expressions. However, the algorithm from Kilpeläinen and Tuhkonen does not naïvely extend to graphs. On graphs their algorithm would require exponential time in the expression since it exploits the fact that the length of the *longest match* of the expression on the word cannot exceed the length of the word. For example, the regular expression a^{42} can only match a word if the word contains 42 a's. This assumption no longer holds for graphs.

Our algorithm for Evaluation gets an RE$(\#, !, \bullet)$ r as input and computes bottom-up, for every subexpression s in the syntax tree of r, the binary relation $\text{eval}(s) \subseteq V \times V$ such that

$$(u, v) \in \text{eval}(s) \text{ if and only if}$$
$$\text{there exists a path from } u \text{ to } v \text{ in } G \text{ that matches } s. \qquad (*)$$

We demonstrate our approach on an example first.

5.3.1 Example. We illustrate a run of the dynamic programming evaluation algorithm for the regular expression $r = (b + c)^* b^{3,5}$ and the graph G from Figure 5.2.

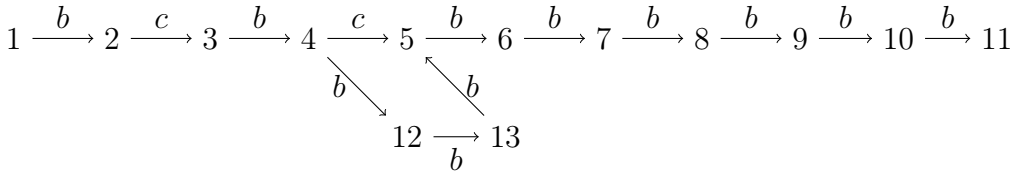


Figure 5.2: An edge-labeled graph G.

Figure 5.3 depicts how to evaluate the regular expression r on the graph G using dynamic programming. Therefore, we compute the relation eval(r) via one bottom-up pass in the syntax tree of r. During the computation each node v of the syntax tree gets annotated with the binary relation eval(s) where s is the subexpression that is associated with the node v.

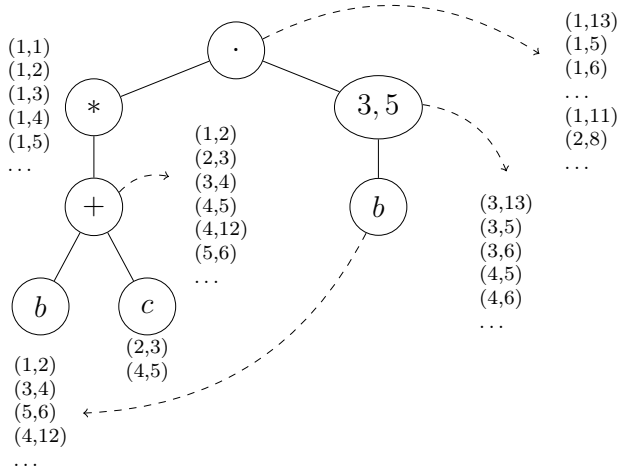


Figure 5.3: Illustration of the dynamic programming approach to evaluate r on G.

Finally, the relation eval(r) is computed at the root node of the syntax tree and contains all pairs (x, y) such that there is a path from x to y that matches r.

In the following we prove that this approach is correct and can be implemented to run in polynomial time. We present the result in two versions (see Theorem 5.3.2 and Corollary 5.3.3), i.e., both results present a polynomial-time result, though the result in Theorem 5.3.2 is not very practical. The biggest bottleneck in the proof of the result is the join procedure for two relations of size n, which has a worst-case time complexity of n^3 (see Lemma 2.3.1). Moreover, if we represent the relations as matrices then the procedure has a best-case complexity of $\Omega(n^2)$, even if the resulting relation is much smaller than n. We therefore look at an alternative algorithm that replaces the matrix multiplications by sort-merge joins (see Lemma 2.3.2). In this way we obtain in Corollary 5.3.3 an algorithm that has a worst-case time complexity of n^3 where the expected runtime is usually small in practical instances. Moreover, the literature emphasizes that the worst case is very unlikely [RG02].

type of expression	computation of eval
$\text{eval}(\emptyset)$	$\emptyset$
$\text{eval}(\varepsilon)$	$\{(u,u) \mid u \in V\}$
$\text{eval}(\bullet)$	$\{(u,v) \mid \exists a \in \Delta \text{ with } (u,a,v) \in E\}$
$\text{eval}(a)$, for $a \in \Delta$	$\{(u,v) \mid (u,a,v) \in E\}$
$\text{eval}(!(a_1 + \cdots + a_n))$	$\{(u,v) \mid \exists a \in \Delta \backslash \{a_1, \ldots, a_n\} \text{ with } (u,a,v) \in E\}$
$\text{eval}(s_1 + s_2)$	$\text{eval}(s_1) \cup \text{eval}(s_2)$
$\text{eval}(s_1 \cdot s_2)$	$\text{eval}(s_1) \bowtie \text{eval}(s_2)$
$\text{eval}(s?)$	$\text{eval}(s) \cup \text{eval}(\varepsilon)$
$\text{eval}(s^+)$	the transitive closure of $\text{eval}(s)$
$\text{eval}(s^*)$	the reflexive and transitive closure of $\text{eval}(s)$
$\text{eval}(s^{k,\infty})$	$\text{eval}(s^k) \bowtie \text{eval}(s^*)$
$\text{eval}(s^{k,\ell})$, for $\ell \neq \infty$	$\text{eval}(s)^k \bowtie \text{eval}(s?)^{\ell-k}$

Table 5.3: Inductive definition of the relation eval.

5.3.2 Theorem. *Let G be a graph and r be an RE(#, !, •). Evaluation for r and G under regular path semantics is in time $O(|r| \cdot |V|^3)$.*

Proof. We first formally define our dynamic programming algorithm for the evaluation of expressions in RE(#, !, •). Afterwards, we show that the algorithm is correct and discuss its complexity. To simplify notation we identify nodes from the syntax tree of r with their corresponding subexpressions. The main idea of our algorithm is to traverse the syntax tree of r bottom-up and compute, for every node with associated subexpression s, a binary relation $\text{eval}(s) \subseteq V \times V$ such that

$(u,v) \in \text{eval}(s)$ if and only if
there exists a path from u to v in G that matches s. (*)

The relation is computed by joining already computed relations for the subexpressions of the child-nodes while going up in the syntax tree. Thereby, the type of join that is computed depends on the associated subexpression of the node. All possible cases for the computation are depicted in Table 5.3. Finally, if the input for Evaluation is an edge-labeled graph $G = (V, E, x, y)$ and an RE(#, !, •) r then the algorithm returns the answer *true* if and only if $(x,y) \in \text{eval}(r)$.

Next, we show that our algorithm for Evaluation is correct and can be implemented to run in polynomial time. Therefore, we prove that the computation of the sets $\text{eval}(s)$ is correct in terms of (*) and show that they can be computed in polynomial time in the graph and the expression. The correctness proof is a straightforward induction on the structure of the expression r. More precisely, we show that the following invariant (I) holds for every relation $\text{eval}(s)$ that is computed:

For each subexpression s of r, we have
$(u,v) \in \text{eval}(s) \Leftrightarrow \exists$ path p in G from u to v such that $\text{lab}(p) \in L(s)$. (I)

In particular, the correctness of the invariant implies the correctness of the algorithm.

The base cases, i.e., the cases where $s = \emptyset$, $s = \varepsilon$, $s = a$ for some $a \in \Delta$, $s = \bullet$, or $s = !(a_1 + \cdots + a_n)$ are immediate.

For the induction step let s be a node in the syntax tree of r such that (I) holds for all of its children. We distinguish several cases depending on the structure of s.

If $s = s_1 + s_2$ then we compute $\text{eval}(s) = \text{eval}(s_1) \cup \text{eval}(s_2)$. Let (u, v) be an arbitrary tuple in $V \times V$. First, we show that if (u, v) in $\text{eval}(s)$ then there exists a path p from u to v in G such that $\text{lab}(p) \in L(s)$. By construction of $\text{eval}(s)$ we know that $(u, v) \in \text{eval}(s_1)$ or $(u, v) \in \text{eval}(s_2)$. By induction hypothesis we know that $\text{eval}(s_1)$ and $\text{eval}(s_2)$ are computed correctly. Therefore, there exists a path p from u to v that matches s_1 or s_2. By definition of s this implies that p also matches s. Second, we show that if there exists a path p from u to v in G with $\text{lab}(p) \in L(s)$ then (u, v) in $\text{eval}(s)$. To this end, let p be a path from u to v in G with $\text{lab}(p) \in L(s)$. By definition of s we have that p matches s_1 or s_2. Since s_1 and s_2 are computed correctly by induction hypothesis, we have that (u, v) is in $\text{eval}(s_1)$ or in $\text{eval}(s_2)$. Since $\text{eval}(s) = \text{eval}(s_1) \cup \text{eval}(s_2)$ it follows that $(u, v) \in \text{eval}(s)$ and $\text{eval}(s)$ fulfills (I).

If $s = s_1 \cdot s_2$ then we compute $\text{eval}(s) := \text{eval}(s_1) \bowtie \text{eval}(s_2)$, where

$$\text{eval}(s_1) \bowtie \text{eval}(s_2) := \{(u, v) \mid \exists z \in V : (u, z) \in \text{eval}(s_1) \wedge (z, v) \in \text{eval}(s_2)\}$$

is the (natural) join of the relation $\text{eval}(s_1)$ and $\text{eval}(s_2)$. Let (u, v) be an arbitrary tuple in $V \times V$. First, we show that if (u, v) in $\text{eval}(s)$ then there exists a path p from u to v in G such that $\text{lab}(p) \in L(s)$. By the definition of the join operator we know that for every tuple $(u, v) \in \text{eval}(s)$ there exist tuples (u, z) and (z, v) such that $(u, z) \in \text{eval}(s_1)$ and $(z, v) \in \text{eval}(s_2)$. Since $\text{eval}(s_1)$ and $\text{eval}(s_2)$ are computed correctly by induction hypothesis there exists a path p_1 from u to z in G with $\text{lab}(p_1) \in L(s_1)$ and a path p_2 from z to v with $\text{lab}(p_2) \in L(s_2)$. Thus the concatenation of these two paths $p = p_1 \cdot p_2$ is a path from u to v in G with $\text{lab}(p) \in L(s_1 \cdot s_2)$. Second, we show that if there exists a path p from u to v in G with $\text{lab}(p) \in L(s_1 \cdot s_2)$ then (u, v) in $\text{eval}(s)$. Therefore, let p be such a path. Since $\text{lab}(p) \in L(s_1 \cdot s_2)$ we know that there exist paths p_1 and p_2 such that $p = p_1 \cdot p_2$ and p_1 is a path from u to some node z with $\text{lab}(p_1) \in L(s_1)$ and p_2 is a path from z to v with $\text{lab}(p_2) \in L(s_2)$. Since $\text{eval}(s_1)$ and $\text{eval}(s_2)$ are computed correctly by induction hypothesis, we know that $(u, z) \in \text{eval}(s_1)$ and $(z, v) \in \text{eval}(s_2)$. It holds that the tuple (u, v) is also in $\text{eval}(s)$ by definition of the join operator. Moreover, it follows that $\text{eval}(s)$ fulfills (I). Notice that if $\varepsilon \in L(s_1)$ (or $\varepsilon \in L(s_2)$) then the relation of s_1 (or s_2, respectively) is reflexive.

If $s = s_1?$ then we compute $\text{eval}(s) := \text{eval}(s_1) \cup \text{eval}(\varepsilon)$. In this case invariant (I) follows immediately from the correctness of the union operation.

If $s = s_1^*$ then we compute $\text{eval}(s)$ as the reflexive and transitive closure of $\text{eval}(s_1)$. Let (u, v) be an arbitrary tuple in $V \times V$. By construction we have that $(u, v) \in \text{eval}(s)$ if and only if (u, v) is in the reflexive and transitive closure of $\text{eval}(s_1)$. This means that either $u = v$, $(u, v) \in \text{eval}(s_1)$, or there exist nodes $v_1, \ldots, v_k$ with $k \geq 1$ such that $(u, v_1), (v_1, v_2), \ldots, (v_k, v) \in \text{eval}(s_1)$. In the first case we have that (u, u) corresponds to a path of length 0. Therefore and since $\varepsilon \in L(s_1^*)$, there is a path from u to u that matches s. In the second case there is a path p from u to v

that matches s_1 (and therefore also s) by the induction hypothesis. Finally, in the third case we know that, by induction hypothesis, there are paths

- p_0 from u to v_1 that matches s_1,
- p_i from v_i to v_{i+1} that match s_1 for every $i = 1, \ldots, k-1$, and
- p_k from v_k to v that match s_1.

It follows that the path $p = p_0 p_1 \cdots p_k$ matches $s = s_1^*$. Now, let p be a path from u to v that matches $s = s_1^*$. By definition of the Kleene star we either have that $u = v$ or there exist paths $p_0, p_1, \ldots, p_k$ with $k \geq 0$ and $p = p_0 p_1 \cdots p_k$ such that p_i matches s_1 for each $i = 0, \ldots, k$. By the definition of the reflexive and transitive closure and since $\text{eval}(s_1)$ is computed correctly by induction hypothesis, we then have that $(u, v) \in \text{eval}(s)$. Therefore, $\text{eval}(s)$ fulfills (I).

If $s = s_1^+$ then we compute the transitive closure of $\text{eval}(s_1)$. The proof of correctness is analogous to the proof for the case $s = s_1^*$.

The last two cases concern the counting operators. If $s = s_1^{k,\ell}$ and $\ell \neq \infty$ then $\text{eval}(s) := \text{eval}(s_1)^k \bowtie \text{eval}(s_1?)^{\ell-k}$. Moreover, we know that $s = s_1^k \cdot (s_1?)^{\ell-k}$. For $\text{eval}(s_1)^k$ invariant (I) holds immediately by correctness of the concatenation case and by induction hypothesis. By correctness of the ?-operator it follows that $\text{eval}(s)$ fulfills (I).

If $s = s_1^{k,\infty}$ then we compute $\text{eval}(s)$ as the relation for the expression $s_1^k \cdot s_1^*$. Since $\text{eval}(s_1)$ is calculated correctly by induction hypothesis, (I) follows directly from the case s_1^k, the concatenation case, and the Kleene star case. This concludes our proof of correctness.

It remains to prove that the algorithm can be implemented to run in polynomial time. Since the syntax tree of the input expression s has linear size and the algorithm processes the syntax tree bottom-up we only need to show that each case can be implemented in time $O(|V|^3 \cdot \log k)$ for cases $s = s_1^k$ and $s = s_1^{k,\infty}$, in time $O(|V|^3 \cdot \log \ell)$ for the case $s = s_1^{k,\ell}$, or in time $O(|V|^3)$ otherwise. Moreover, notice that each relation $\text{eval}(s)$ has size $O(|V|^2)$. The proof of the assumption for the cases $s \in \Delta$, $s = \varepsilon$, $s = \bullet$, $s = !(a_1 + \cdots + a_n)$, $s = s_1 \cdot s_2$, $s = s_1^*$, $s = s_1^+$, and $s = s_1?$ are either straightforward or immediate from Lemma 2.3.1. The cases $s = s_1^k$ and $s = s_1^{k,\infty}$ can be computed in time $O(|V|^3 \cdot \log k)$ by applying Lemmas 2.3.1 and 2.3.3. Similarly, we obtain that the case $s = s_1^{k,\ell}$ can be implemented in time $O(|V|^3 \cdot \log \ell)$. This concludes the proof. □

We conclude this section with a more precise analysis of the complexity of the algorithm from Theorem 5.3.2. Therefore, consider the algorithm where the join operation from Lemma 2.3.1 is replaced by the join operation of Lemma 2.3.2. To this end, let

- $R_{\max}$ be the maximal size of any relation $\text{eval}(s)$ for any subexpression s in the algorithm of Theorem 5.3.2 (including the intermediate results for fast squaring), and

– $j_{\max}$ be the maximum of $R_{\max}$ and every number

$$j = |\{(u, z, v) \mid (u, z) \in \text{eval}(s_1) \text{ and } (z, v) \in \text{eval}(s_2)\}|$$

where $\text{eval}(s_1)$ and $\text{eval}(s_2)$ range over all joins that the algorithm performs during a computation (including the intermediate results for fast squaring).

5.3.3 Corollary. *Let G be a graph and r be an RE(#, !, •).* Evaluation *for r and G under regular path semantics is in time $O(|r| \cdot j_{max} \log R_{max})$.*

Since our evaluation algorithm is still rather naïve we feel that further improvements towards better data complexity are very likely to be possible. However, our main goal in [LM12, LM13] was to show that SPARQL property path expressions are tractable under regular path semantics. Later, we will see that this is not the case under simple path semantics.

5.3.2 Evaluation for Regular Expressions with Negation

In the last section we have seen that Evaluation for SPARQL regular expressions is tractable under regular path semantics. Our algorithm for this problem handles counting operators as well as a restricted negation operator (!). In this section we revisit the tractability of negation operators in regular expressions and show that once unlimited negation is allowed in regular expressions the Evaluation problem over graphs becomes intractable.

Nevertheless, we first provide a positive result by proving that Evaluation for regular expressions with full-fledged negation (¬) over words (instead of graphs) is in P. Notice that this problem is better known as Membership in the literature (see, e.g., Section 2.4). To obtain the result we use the algorithm from Theorem 5.3.2. We will also use this result again to examine the complexity of Counting in Section 5.4.

5.3.4 Theorem. *Let r be an RE(#, !, ¬, •) and w be a word.* Membership *for r and w is in time $O(|r| \cdot |w|^3)$.*

Proof. To simplify the technical presentation in this proof we abstract a word as an acyclic, connected, edge-labeled graph in which every node has at most one incoming or outgoing edge. As such, we can re-use the algorithm from Theorem 5.3.2. We already showed in the proof of Theorem 5.3.2 that Evaluation (and therefore also Membership) for RE(#, !, •) is in P. Next, we show how the algorithm of Theorem 5.3.2 can be extended to RE(#, !, ¬, •) expressions in the case that the input is a word (instead of a graph). Therefore, we extend our algorithm by the following extra case:

– If $s = \neg s_1$ then $\text{eval}(s) := \{(u, v) \mid (u, v) \notin \text{eval}(s_1)\}$.

Next, we prove that the algorithm is correct. To this end, let r be an RE(#, !, ¬, •). We prove the same invariant (I) as in the proof of Theorem 5.3.2, that is:

For each subexpression s of r, we have

$$(u, v) \in \text{eval}(s) \Leftrightarrow \exists \text{ path } p \text{ in } G \text{ from } u \text{ to } v \text{ such that } \text{lab}(p) \in L(s). \qquad \text{(I)}$$

Since we are considering only words we get that, by (I), a tuple (u, v) represents two positions in the input word such that $(u, v) \in \text{eval}(s)$ if and only if the subword from u to v is in $L(s)$.

The induction base cases and the cases for all operators, except $\neg$, can be proved analogously to the cases in Theorem 5.3.2. Thus, it remains to consider the case $\text{eval}(\neg s_1) = \{(u, v) \mid (u, v) \notin \text{eval}(s_1)\}$. By the definition of the negation operator and since we are dealing with a word, the (unique) path from u to v matches $\neg s_1$ if and only if it does not match s_1. By induction hypothesis $\text{eval}(s_1)$ is calculated correctly, i.e., a tuple (u, v) is in $\text{eval}(s_1)$ if and only if there exists a path from u to v such that $\text{lab}(p) \in L(s_1)$. By definition we have that $(u, v) \in \text{eval}(\neg s_1)$ if and only if $(u, v) \notin \text{eval}(s_1)$. Thus, $\text{eval}(\neg s_1)$ fulfills (I). It is straightforward to implement the new case in time $O(|r| \cdot |w|^3)$. □

Notice that, for arbitrary graphs and expressions in $\text{RE}(\#, !, \neg, \bullet)$, the algorithm from Theorem 5.3.4 would fail because we cannot assume that there is only one unique path from node u to node v in a graph. Intuitively, it can be the case that $(u, v) \in \text{eval}(s)$ and $(u, v) \in \text{eval}(\neg s)$ in the same graph (in this case there exist two distinct paths that are labeled respectively).

In fact, if we want to evaluate over graphs instead of words, allowing unrestricted negation in expressions does not allow for an efficient algorithm for Evaluation on graphs and makes the complexity of Evaluation non-elementary. The reason why is that Evaluation is at least as hard as testing non-emptiness for the considered class of regular expressions.

5.3.5 Lemma. *Let C be a class of regular expressions over a finite alphabet Σ. Then there exists a LOGSPACE reduction from the non-emptiness problem for C-expressions to* Evaluation *for C-expressions under regular path semantics.*

Proof. The proof is immediate from the observation that non-emptiness of an expression r over an alphabet Σ is the same decision problem as Evaluation for r and the graph $G = (V, E)$ with $V = \{x\}$ and $E = \{(x, a, x) \mid a \in \Sigma\}$. □

Since it is well-known that the non-emptiness problem for star-free generalized regular expressions is non-elementary [Sto74] it immediately follows that Evaluation is non-elementary for the class $\text{RE}(\neg)$ by applying Lemma 5.3.5. However, every $\text{RE}(\#, !, \neg, \bullet)$ can be converted to an $\text{RE}(\#, !, \bullet)$ with a non-elementary blow-up such that Evaluation still remains decidable for regular expression with negation.

5.3.6 Theorem.

- Evaluation *for $\text{RE}(\neg)$ under regular path semantics is non-elementary.*
- Evaluation *for $\text{RE}(\#, !, \neg, \bullet)$ under regular path semantics is decidable.*

5.3.3 Evaluation under Simple Walk Semantics

In Section 5.3.1, we have seen that SPARQL regular expressions can be efficiently evaluated under regular path semantics via a rather simple algorithm. In this section we investigate how the complexity of Evaluation changes when simple walk

semantics is applied. In this case it follows from classical results that EVALUATION under simple walk semantics is already NP-complete for very restricted expressions. However, EVALUATION remains in NP even when counting operators are allowed.

5.3.7 Theorem.

(1) EVALUATION *for expressions* $(aa)^*$ *and* $(aa)^+$ *under simple walk semantics is NP-complete.*

(2) EVALUATION *for RE(*$\#, !, \bullet$*) under simple walk semantics is NP-complete.*

(3) EVALUATION *for RE under simple walk semantics is NP-complete.*

Proof. (1) Lapaugh and Papadimitriou showed that, given a directed graph and two nodes x and y, it is NP-hard to decide whether there exists a simple path of even length from x to y (see, e.g., Theorem 2.4.5). The NP upper bound is obtained by guessing a simple path from x to y and checking if it has even length. Since the path has to be simple it is of linear size in the graph.

Moreover, the theorem shows that EVALUATION under simple walk semantics is also NP-hard when considering data complexity.

(2) The NP lower bound is immediate by (1). The NP upper bound follows from an adaptation of the algorithm of Theorem 5.3.2 where in the cases $s = s_1^*$ and $s = s_1^+$ simple walks are guessed between nodes to test whether they belong to $\text{eval}(s)$. We illustrate this algorithm in the following. Let $G = (V, E, x, y)$ be the input graph and let r be the input regular expression. Our NP algorithm traverses the syntax tree of the expression r bottom-up and computes, for every node with associated subexpression s, a binary relation $\text{eval}_{\text{sw}}(s) \subseteq V \times V$ such that

$$(u, v) \in \text{eval}_{\text{sw}}(s) \Leftrightarrow \exists \text{ path } p \text{ in } G \text{ from } u \text{ to } v \text{ such that } p \text{ matches } s \text{ under simple walk semantics.}$$

The type of join that is computed, while going up in the syntax tree, depends on the associated subexpression of the node. All possible cases for the computation are depicted in Table 5.4. In the table we denote by eval the function for evaluating expressions under regular path semantics (as it is defined in Table 5.3). Finally, the algorithm returns *true* if and only if $\text{eval}_{\text{sw}}(r)$ contains the pair (x, y).

type of expression	computation of eval_{sw}
$\text{eval}_{\text{sw}}(\emptyset)$	$\emptyset$
$\text{eval}_{\text{sw}}(\varepsilon)$	$\{(u,u) \mid u \in V\}$
$\text{eval}_{\text{sw}}(\bullet)$	$\text{eval}(\bullet)$
$\text{eval}_{\text{sw}}(a)$, for $a \in \Delta$	$\text{eval}(a)$
$\text{eval}_{\text{sw}}(!(a_1 + \cdots + a_n))$	$\text{eval}(!(a_1 + \cdots + a_n))$
$\text{eval}_{\text{sw}}(s_1 + s_2)$	$\text{eval}_{\text{sw}}(s_1) \cup \text{eval}_{\text{sw}}(s_2)$
$\text{eval}_{\text{sw}}(s_1 \cdot s_2)$	$\text{eval}_{\text{sw}}(s_1) \bowtie \text{eval}_{\text{sw}}(s_2)$
$\text{eval}_{\text{sw}}(s?)$	$\text{eval}_{\text{sw}}(s) \cup \text{eval}_{\text{sw}}(\varepsilon)$
$\text{eval}_{\text{sw}}(s^+)$	$\{(u,v) \mid \exists$ simple walk p from u to v such that $\text{lab}(p) \in L(s^+)\}$
$\text{eval}_{\text{sw}}(s^*)$	$\text{eval}_{\text{sw}}(s^+) \cup \text{eval}_{\text{sw}}(\varepsilon)$
$\text{eval}_{\text{sw}}(s^{k,\infty})$	$\text{eval}_{\text{sw}}(s)^k \bowtie \text{eval}_{\text{sw}}(s^*)$
$\text{eval}_{\text{sw}}(s^{k,\ell})$, for $\ell \neq \infty$	$\text{eval}_{\text{sw}}(s)^k \bowtie \text{eval}_{\text{sw}}(s?)^{\ell-k}$

Table 5.4: Inductive definition of the relation eval_{sw}.

It remains to show that $\text{eval}_{\text{sw}}(s)$ can be computed in polynomial time using nondeterminism. Therefore, we basically implement the polynomial-time algorithm of Theorem 5.3.2 except for the case $\text{eval}_{\text{sw}}(s^+)$. In this case nondeterminism is needed. To decide whether a pair (u,v) is in $\text{eval}_{\text{sw}}(s^+)$ the algorithm guesses a simple walk p from u to v, which trivially has polynomial length, and tests whether $\text{lab}(p) \in L(s^+)$.

Correctness of the algorithm is proved analogously to the proof of Theorem 5.3.2. To this end, we have to adapt invariant (I) to nondeterminism and simple walk semantics. Formally, the new invariant is:

> There is a run of the algorithm such that, for every subexpression s of r,
> $(u,v) \in \text{eval}_{\text{sw}}(s) \Leftrightarrow \exists$ path p in G from u to v such that
> p matches s under simple walk semantics.

The proof of the induction is straightforward.

(3) Directly holds by (1) and (2). □

The results in Theorem 5.3.7 restrain the possibilities for finding polynomial-time fragments for Evaluation under simple walk semantics rather severely. In order to find such fragments and in order to trace a tractability frontier for the problem, we examine syntactically very restricted regular expressions in the following. That, is we investigate the Evaluation problem for CHAREs (see Definition 3.2.1) under simple walk semantics. In this way we can show that it is possible to use the *- and $^+$-operators in an expression and obtain a polynomial-time evaluation complexity at the same time. However, below these operators it is only allowed to use a disjunction of single symbols.

5.3.8 Theorem. EVALUATION *for CHARE*$((+a)^*, (+a)^+, (+w), (+w)?)$ *under simple walk semantics is in P.*

Proof. The theorem directly holds by the observation that, for every expression $r \in$ CHARE$((+a)^*, (+a)^+, (+w), (+w)?)$ and graph G, we have that whenever there is a path in G that matches r under simple walk semantics there also exists a path in G that matches r under regular path semantics. In particular, this means that $\text{eval}_{\text{sw}}(r) = \text{eval}(r)$ and that r can be evaluated in polynomial time by Theorem 5.3.2. (For the formal definitions of $\text{eval}(r)$ and $\text{eval}_{\text{sw}}(r)$ see Tables 5.3 and 5.4). □

Regarding the result of Theorem 5.3.8 we would like to point out the following two remarks. First, we stress that for the proof of Theorem 5.3.8 it makes a minor difference if we would only allow simple paths instead of simple walks. (For details, on the simple path versus simple walk discussion revisit our formalization of **ZeroOrMorePath** and **OneOrMorePath** in Section 5.2.) However, the result holds for both variants. More precisely, EVALUATION for CHARE$((+a)^*, (+a)^+, (+w), (+w)?)$ remains in P, even if we would not allow simple cycles to match expressions of the form $(+a)^*$ or $(+a)^+$. To see that this is true let id be the identity relation and let $\text{eval}'(r)$ be the set of pairs of nodes (u, v) such that there exists a path from u to v in the graph that matches an expression r under these semantics (i.e., simple walk semantics that do not allow simple cycles). Then, observe that

- for every subexpression s of the form $(a_1 + \cdots + a_k)^*$ or $(a_1 + \cdots + a_k)^+$ we have $\text{eval}'(s) = \text{eval}(s) - id$,
- for every subexpression s of the form $(w_1 + \cdots + w_k)$ or $(w_1 + \cdots + w_k)?$ we have $\text{eval}'(s) = \text{eval}(s)$, and
- $\text{eval}'(s_1 \cdot s_2) = \text{eval}'(s_1) \bowtie \text{eval}'(s_2)$ for all subexpressions s_1 and s_2.

Second, we stress that there exists an odd relationship between Theorem 5.3.8 and Theorem 1 in [MW95]. According to the first theorem it can be decided whether there exists a path that matches the expression a^*ba^* under simple walk semantics in polynomial time. The latter theorem states that deciding whether there exists a simple path that matches the expression a^*ba^* is NP-complete. The difference, however, is that under simple walk semantics we do not require the *entire* path to be simple.

Regarding the search for more polynomial-time cases we note that the range of possible fragments between the expressions in CHARE$((+a)^*, (+a)^+, (+w), (+w)?)$ and the expressions in Theorem 5.3.7 is quite limited. For example, a limitation of Theorem 5.3.8 is that CHAREs do not allow arbitrary nesting of disjunctions. Since simple walk semantics and regular path semantics coincide for RE-expressions that do not use the Kleene star or the $^+$-operator EVALUATION for those expressions under simple walk semantics is tractable as well.

5.3.9 Observation. EVALUATION for star-free regular expressions under simple walk semantics is in P.

To conclude we remark that Bagan et al. recently studied a variation of the Evaluation problem under simple walk semantics in which the whole regular expression should be matched by a simple path in [BBG13]. Under the assumption that P $\neq$ NP, they can precisely characterize which kinds of regular expressions can be efficiently evaluated on graphs (when considering data complexity).

5.4 The Complexity of the Counting Problem for Property Paths

Our motivation to study the Counting problem originates from the SPARQL 1.1 working draft [HS12] which requires that, for example, for SPARQL queries of the form `SELECT ?x, ?y WHERE {?x r ?y}` where r is a property path the result is a multiset that has n copies of a pair $(x, y) \in V \times V$ where n is the number of paths between x and y that match r. Thus, the SPARQL 1.1 specification demands that that all correct answers to certain queries have to be counted. We informally refer to this requirement as the path counting requirement.

5.4.1 Note. Property paths that are evaluated under the SPARQL 1.1 specification are required to fulfill the *path counting requirement*, that is, the number of paths from x to y that match a property path expression r needs to be counted.

Notice that, according to the path counting requirement, the answer to the previous stated `SELECT` query needs to contain the answer to the Counting problem in unary notation and that the number of paths that match an expression is always finite under simple walk semantics. Under regular path semantics this is not always the case. To complete the picture for the comparison between regular path semantics and simple walk semantics, we discuss the complexity of Finiteness in Section 5.5.

In the following we first investigate Counting under regular path semantics and, afterwards, under simple walk semantics.

5.4.1 Counting under Regular Path Semantics

To get some intuition for the challenges of Counting for regular path semantics we illustrate the problem on an example instance in the following. In the example we also illustrate that, although dynamic programming has been a good approach to attack Evaluation, this is no longer true for Counting (at least not in an obvious manner).

5.4.2 Example. Consider the graph $(V, E, 0, 6)$ in Figure 5.4. We are interested in how many paths from node 0 to node 6 match a regular expression r. An obvious approach for solving this problem is through divide-and-conquer (as we did for Evaluation in Section 5.3.1). To this end, we could try to count partial results for subexpressions and subgraphs and combine these partial results to obtain the overall result. Next, we briefly demonstrate on two instances why such an approach will not work in a naïve way.

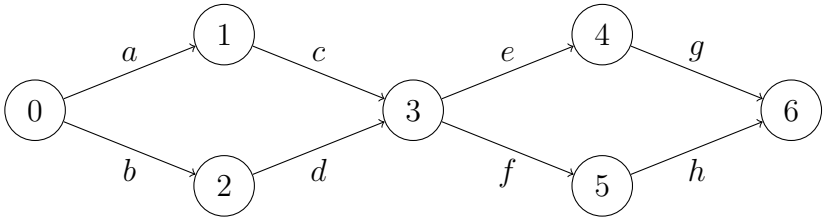


Figure 5.4: The edge-labeled graph $(V, E, 0, 6)$ from Example 5.4.2.

Consider the expression $r = (a \bullet \bullet \bullet) + (\bullet\, c \bullet \bullet)$. If we would recursively count the number of paths that match $s_1 = a \bullet \bullet \bullet$ and $s_2 = \bullet\, c \bullet \bullet$ and add these numbers together then the result does not equal the number of paths that match r. More precisely, there are two paths from 0 to 6 that match s_1 and two paths that match s_2. However, these are the same two paths such that there are only two paths that match r. (We counted the paths labeled $aceg$ and $acfh$ twice.)

An idea that would solve this problem for the above expression is to color all the edges that are on some path that matches one of the subexpressions and count the number of colored paths from 0 to 6 afterwards. For the above expression, this approach would color the edges labeled a, c, e, g, f, and h and would therefore lead to the correct result.

However, this approach does not work in general. To see that this is true, consider the expression $r = (ac(eg + fh)) + ((ac + bd)fh)$. With this expression, we would color all the edges in the graph. Thus, the number of colored paths from node 0 to node 6 would be four but the number of paths that match r is only three. Here the problem is that we erroneously add the path $bdeg$ to the result.

A key for obtaining a polynomial-time algorithm therefore seems to lie in some way to decompose the paths of the graph such that matching paths are not counted twice and such that partial paths that are not allowed to occur together are not erroneously combined. In Theorem 5.4.7, we show that it is unlikely that such an algorithm exists for the class of arbitrary SPARQL regular expressions by proving that Counting under regular path semantics is #P-hard (even for very restricted CHARE fragments).

Before we prove these lower bounds, we show that Counting for unambiguous regular under regular path semantics can be solved in polynomial time using an automata-based approach. Regarding this result notice that, although every regular expression can be converted to an unambiguous one (see, e.g., Section 3.3), the conversion results in an exponentially larger expression in the worst case. Since deterministic regular expression are a proper subclass of unambiguous expressions it follows that Counting can also be solved for deterministic regular expressions in polynomial time. To prove this polynomial-time result for Counting we use a unique product graph of the automaton for the input expression and the input graph. The definition of this product graph is very similar to the standard product for finite automata (see, e.g., [HMU13]). In the following definition we also use s-t graphs that have a set of target nodes (see Definition 2.2.3).

5.4.3 Definition. Let $G = (V, E, x, y)$ be an s-t graph and $N = (Q, \Sigma, \Delta, \delta, q_0, F)$ be an NFA^w. Then, the *product graph* $G_{x,y} \times N$ of G and N is the s-t graph $(V_{G,N}, E_{G,N}, x_{G,N}, Y_{G,N})$ where

- the set of nodes $V_{G,N}$ is $V \times Q$,
- the source node $x_{G,N}$ is (x, q_0),
- the set of target nodes $Y_{G,N}$ is $\{(y, q_f) \mid q_f \in F\}$, and
- for each $a \in \Delta$ there is an edge $((v_1, q_1), a, (v_2, q_2)) \in E_{G,N}$ if and only if there is an edge (v_1, a, v_2) in G and either
  - $a \in \Sigma$ and $q_2 \in \delta(q_1, a)$, or
  - $a \notin \Sigma$ and $q_2 \in \delta(q_1, \circ)$.

If N is a UFA^w then there exists a strong relation between paths from x to y in G and paths from $x_{G,N}$ to $Y_{G,N}$ in $G_{x,y} \times N$. We formalize this relation by a mapping φ_{PATHS}. Let $p = x[a_1]v_1[a_2]v_2 \cdots v_{n-1}[a_n]y$ be a path in G. Then,

$$\varphi_{\text{PATHS}}(p) = (x, q_0)[a_1](v_1, q_1)[a_2](v_2, q_2) \cdots (v_{n-1}, q_{n-1})[a_n](y, q_n)$$

where $q_0 q_1 \dots q_n$ is the unique accepting run of N on the word $a_1 \dots a_n$. There is only one accepting run for $a_1 \dots a_n$ since N is unambiguous. Thus, φ_{PATHS} is well-defined.

5.4.4 Lemma. *If N is a UFA^w then φ_{PATHS} is a bijection between paths from x to y in G that match N and paths from $x_{G,N}$ to some node in $Y_{G,N}$ in $G_{x,y} \times N$. Furthermore, φ_{PATHS} preserves the length of paths.*

Proof. Observe that the mapping φ_{PATHS} preserves the length and label of a path by definition. We show that φ_{PATHS} is a bijection from

- the set $P(G)$ of paths from x to y in G that match N, to
- the set $P(G \times N)$ of paths from $x_{G,N}$ to some node in $Y_{G,N}$ in $G_{x,y} \times N$.

To this end, we need to show that φ_{PATHS} is surjective and injective.

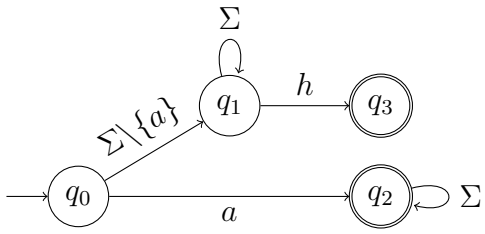
We show first that φ_{PATHS} is surjective. Let $N = (Q, \Sigma, \Delta, \delta, q_0, F)$ be a UFA^w and let $p = (x, q_0)[a_1](v_1, q_1)[a_2](v_2, q_2) \cdots (v_{n-1}, q_{n-1})[a_n](y, q_n)$ be a path in $P(G_{x,y} \times N)$ with $(y, q_n) \in Y_{G,N}$. We prove that for the path $p^G = x[a_1]v_1[a_2]v_2 \cdots v_{n-1}[a_n]y$ it holds that $p = \varphi_{\text{PATHS}}(p^G)$. By the definition of the graph $G_{x,y} \times N$ there is an edge $((v_i, q_i), a, (v_j, q_j))$ if and only if there is a transition in δ that takes q_i to q_j by reading a. As such, the existence of p in $G_{x,y} \times N$ implies that there is an accepting run $q_0 \cdots q_n$ of N on the word $a_1 \cdots a_n$. By the unambiguity of N the run $q_0 \cdots q_n$ is unique. Therefore, we have that, by definition of φ_{PATHS},

$$\begin{aligned}\varphi_{\text{PATHS}}(p^G) &= \varphi_{\text{PATHS}}(x[a_1]v_1[a_2]v_2 \cdots v_{n-1}[a_n]y) \\ &= (x, q_0)[a_1](v_1, q_1)[a_2](v_2, q_2) \cdots (v_{n-1}, q_{n-1})[a_n](y, q_n) = p.\end{aligned}$$

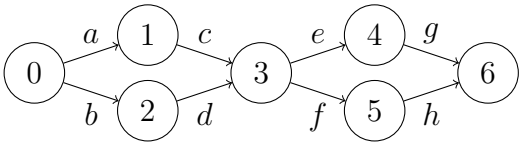
We now show that φ_{PATHS} is injective. Let $p = x[a_1]v_1 \cdots v_{n-1}[a_n]y$ and $p' = x[a'_1]v'_1 \cdots v'_{n-1}[a'_n]y$ be two paths in $P(G)$ with $\varphi_{\text{PATHS}}(p) = \varphi_{\text{PATHS}}(p')$. We prove that $p = p'$. Let $\varphi_{\text{PATHS}}(p) = (x, q_0)[a_1](v_1, q_1)[a_2](v_2, q_2) \cdots (v_{n-1}, q_{n-1})[a_n](y, q_n)$. Since φ_{PATHS} preserves the labels on edges and the nodes in V it follows that $p = x[a_1]v_1[a_2]v_2 \cdots v_{n-1}[a_n]y = p'$. This concludes the proof. □

Next, we illustrate a short example of how the product graph from Definition 5.4.3 can be used to efficiently solve COUNTING for UFA^{w}s.

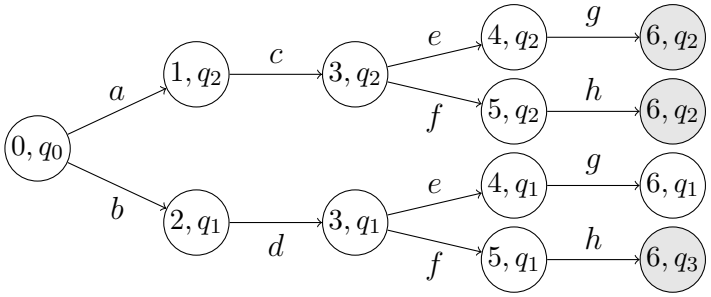
5.4.5 Example. Let A be the UFA^w from Figure 5.5(a) and $G = (V, E, 0, 6)$ be the s-t graph from Figure 5.5(b). Notice that A describes the language $L(a\Sigma^* + \Sigma^+h)$ where Σ is an abbreviation for the expression $(a + b + c + d + e + f + g + h)$. The product graph $G_{0,6} \times A$ of A and G is depicted in Figure 5.5(c). On closer inspection of the product graph we can observe that the number of paths from node 0 to 6 in G that match A is precisely the number of paths from the source node to a target node in $G_{0,6} \times A$.



(a) A UFA^w A.



(b) An s-t graph $G = (V, E, 0, 6)$.



(c) Fragment of the graph $G_{0,6} \times A$. (Nodes in $Y_{G,A}$ are gray.)

Figure 5.5: An illustration of the algorithm for COUNTING and UFA^{w}s.

In the following we prove some general upper bounds for the COUNTING problem under regular path semantics. More precisely, we show that COUNTING can be solved in polynomial time for each class of regular expressions that can be converted to UFA^{w}s in polynomial time. Afterwards, it directly follows that COUNTING for unambiguous (and deterministic) regular expressions can be solved in polynomial time. In addition, we observe that COUNTING for the class of all regular expressions is in #P.

5.4.6 Theorem.

(1) COUNTING *for UFA^{w}s under regular path semantics is in P, even if the number* max *is given in binary.*

(2) COUNTING *for DFA^{w}s under regular path semantics is in P, even if the number* max *is given in binary.*

(3) COUNTING *for unambiguous and deterministic regular expressions under regular path semantics is in P, even if the number* max *is given in binary.*

(4) COUNTING *for RE(#, !, ¬, •) under regular path semantics is in #P.*

Proof. (1) To prove the assumption we present a polynomial-time algorithm that reduces COUNTING for UFA^{w}s to the problem of counting the number of paths in a graph using the product graph from Definition 5.4.3. Notice that, due to Theorem 2.2.4, the latter problem can be solved in polynomial time even if the number *max* is given in binary. Now, let $G = (V, E, x, y)$ be a graph and $A = (Q, \Sigma, \Delta, \delta, q_0, F)$ be a UFA^w. Then, the algorithm works as follows:

- Construct $G_{x,y} \times A$, i.e., the product graph of (V, E, x, y) and A.
- Return $\sum_{q_f \in F} \text{PATHS}((x, q_0), (y, q_f))$ in $G_{x,y} \times A$.

Here, $\text{PATHS}((x, q_0), (y, q_f))$ denotes the number of paths from node (x, q_0) to (y, q_f) in $G_{x,y} \times A$ that have at most length *max*. By Lemma 5.4.4 this algorithm is correct. In fact, the lemma shows that the number of paths from x to y in G that have at most length *max* and match A equals the number of paths of length at most *max* from (x, q_0) to some node in $\{y\} \times F$ of $G_{x,y} \times A$.

(2) Since every DFA^w is a UFA^w, (2) directly holds by the proof of (1).

(3) The statement directly holds by translating the expressions to Glushkov automata (see Proposition 5.1.7). By the definition of unambiguous regular expressions their Glushkov automata are UFA^{w}s.

(4) Let $G = (V, E, x, y)$ be the input graph, r be an RE(#, !, ¬, •) expression, and $max \in \mathbb{N}$ be a number given in unary notation. The nondeterministic Turing machine for the $\#P$-procedure guesses a path p of length at most *max* in G and tests whether $\text{lab}(p) \in L(r)$ afterwards. Since MEMBERSHIP for RE(#, !, ¬, •) can be solved in polynomial time by Theorem 5.3.4 and the number *max* is given in unary, this can be done in polynomial time. The only nondeterminism in the algorithm originates from guessing the path. Thus, the number of accepting computations of the Turing machine corresponds to the number of paths of length at most *max* in G that match the expression. □

We now turn our attention to lower bounds for the COUNTING problem and show that the problem becomes intractable as soon as we allow only a little bit more nondeterminism in the input expressions. To this end, we consider the complexity

of COUNTING for chain regular expressions (see Section 3.2). In more detail, we show that COUNTING is #P-complete for all classes of CHAREs that allow a single label (i.e., "a") as a factor and cannot be trivially converted to DFA^{w}s or UFA^{w}s in polynomial time.

Our lower bound proofs are obtained by adapting a proof in [Mar06, MNS09] which shows that language inclusion for various classes of CHAREs is coNP-hard. Moreover, the technique from [Mar06, MNS09] is itself a robust generalization of a proof by Miklau and Suciu [MS04] which shows that inclusion for particular XPath expressions is coNP-hard.

5.4.7 Theorem. COUNTING *under regular path semantics is #P-complete for*

(1) $\mathit{CHARE}(a, a^*)$,

(2) $\mathit{CHARE}(a, a?)$,

(3) $\mathit{CHARE}(a, w^+)$,

(4) $\mathit{CHARE}(a, (+a^+))$,

(5) $\mathit{CHARE}(a, (+a)^+)$,

(6) $\mathit{CHARE}((+a), a^+)$, *and*

(7) $\mathit{RE}(\#, !, \neg, \bullet)$.

Moreover, #P-hardness already holds if the graph G is acyclic.

Proof. We prove the cases (1) to (6) first. The upper bounds for the cases (1) to (6) are immediate from Theorem 5.4.6. We prove the lower bounds for these cases by reductions from #DNF. As previously mentioned the used techniques are heavily inspired by proofs in [Mar06, MNS09, MS04]. To this end, we first describe a meta-reduction for the cases (1) to (3) and then instantiate it with slightly different subgraphs and expressions to deal with the different cases. For the cases (4) to (6) we perform a similar approach.

Meta-reduction for the cases (1) to (3): Let $\Phi = C_1 \vee \cdots \vee C_k$ be a propositional formula in 3DNF using variables $\{x_1, \ldots, x_n\}$. We encode truth assignments for Φ by paths in the graph. In particular, we construct a graph $G = (V, E, x, y)$, an expression r, and a number *max* such that each path from x to y of length at most *max* in G that matches r corresponds to a unique satisfying truth assignment for Φ and vice versa. Formally, we will prove that the following holds:

> The number of paths from x to y of length at most *max* in G that match r is equal to the number of truth assignments that satisfy Φ. (*)

The structure of the graph G is depicted in Figure 5.6.

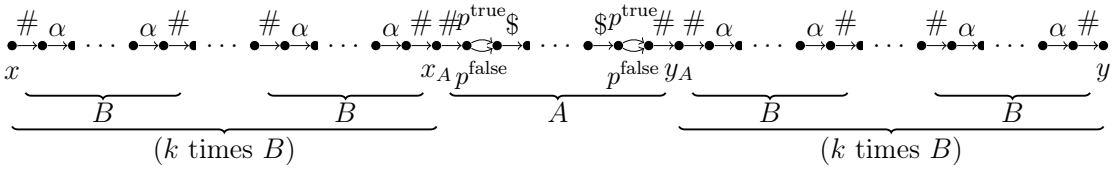


Figure 5.6: The graph G from the proof of Theorem 5.4.7 for the cases (1)–(3).

Basically, the graph has the structure $B^k A B^k$ where

- B is a path labeled $\#\alpha\$\alpha\$\cdots\$\alpha\#$ (containing n copies of α), and
- A is a subgraph with a distinguished source node x_A, a target node y_A, and containing n copies of the gadget labeled $p^{\text{true}}/p^{\text{false}}$. Notice that all paths from x to y will enter A through the node x_A and leave A through y_A.

The subgraphs α, p^{true}, and p^{false} are supposed to be paths themselves which we instantiate differently in each of the cases (1) to (3). Moreover, notice that we instantiate α, p^{true}, and p^{false} with acyclic graphs such that the instantiated graphs remain acyclic in each case.

By the structure of the graph each path from x_A to y_A in A can be associated to exactly one truth assignment for the variables $\{x_1, \ldots, x_n\}$. That is, for each truth assignment ψ, there is a path p_ψ in A from x_A to y_A and, for each path p in A from x_A to y_A, there is a corresponding truth assignment ψ_p. More precisely, consider the structure of A as depicted in Figure 5.6 with n occurrences of p^{true} and p^{false}. Here, p^{true} and p^{false} are paths in A whose labels depend on the case (or the CHARE fragment) we are considering. The paths p^{true} and p^{false} do not use any of the special labels, i.e., \$ or #. A path through A from x_A to y_A therefore has n choices of going through p^{true} or p^{false}. If the i-th choice goes through p^{true} then this corresponds to a truth assignment that sets x_i to *true*. The same holds for p^{false} and an assignment that sets x_i to *false*. This concludes the description of the graph.

We now illustrate the structure of the expression r that is used in the meta-reduction for the cases (1) to (3). To this end, we construct an expression

$$r = NF(C_1)\cdots F(C_k)N$$

such that

- each path labeled $\text{lab}(B)^i$ for $i = 1, \ldots, k$ matches N,
- each subexpression $F(C_i)$ can be associated with clause C_i where $i \in \{1, \ldots, k\}$,
- each path B matches every subexpression $F(C_i)$, and
- for each clause C_i and path p in A the path p matches $F(C_i)$ if and only if the truth assignment ψ_p that is associated with p satisfies C_i.

In the following we use the subexpressions r^{true}, r^{false}, and r^{all} to define the expressions $F(C_i)$. Intuitively, these expressions correspond to a variable occurring

positively, negatively, or not at all in a clause C_i. Later, we instantiate these expressions separately for the cases (1) to (3). Formally we define, for each clause C, the expression

$$F(C) = \#e_1\$\cdots\$e_n\#$$

where, for each $j = 1, \ldots, n$,

$$e_j := \begin{cases} r^{\text{false}}, & \text{if } x_j \text{ occurs negated in } C, \\ r^{\text{true}}, & \text{if } x_j \text{ occurs positive in } C, \\ r^{\text{all}}, & \text{otherwise.} \end{cases}$$

The subexpression N will be defined differently for each case.

Next, we want to prove some general properties about the meta-reduction that we need later to prove correctness for the different cases. Therefore, we make the following assumptions on paths and expressions. In the proofs of the cases (1) to (3) we then prove that the assumptions are met:

$$\text{lab}(\alpha) \in L(r^{\text{false}}) \cap L(r^{\text{true}}) \tag{P1}$$
$$\text{lab}(p^{\text{true}}) \in L(r^{\text{true}}) - L(r^{\text{false}}) \tag{P2}$$
$$\text{lab}(p^{\text{false}}) \in L(r^{\text{false}}) - L(r^{\text{true}}) \tag{P3}$$
$$\{\text{lab}(p^{\text{true}}), \text{lab}(p^{\text{false}}), \text{lab}(\alpha)\} \subseteq L(r^{\text{all}}) \tag{P4}$$

Notice that (P2) and (P3) imply that $\text{lab}(p^{\text{true}}) \neq \text{lab}(p^{\text{false}})$. Due to the structure of G and since p^{true} and p^{false} do not use the symbols \$ or #, we therefore have that paths p_1 and p_2 from x to y are different if and only $\text{lab}(p_1) \neq \text{lab}(p_2)$. Moreover, we know that the only part where paths from x to y can differ is in the subgraph A (because B is a path). Thus, we can uniquely identify a path p from x to y in G with the label of the subpath p' of p that goes from x_A to y_A. In addition, we can associate a truth assignment to each such path and vice versa:

- For a path with label $w = \#w_1\$\cdots\$w_n\#$ we define V_w as the truth assignment where, for each $j = 1, \ldots, n$,

$$V_w(x_j) := \begin{cases} \mathit{true}, & \text{if } w_j = \text{lab}(p^{\text{true}}), \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

- For a truth assignment V we define the word $w_V = \#w_1\$\cdots\$w_n\#$ such that, for each $j = 1, \ldots, n$,

$$w_j = \begin{cases} \text{lab}(p^{\text{true}}), & \text{if } V(x_j) = \mathit{true}, \\ \text{lab}(p^{\text{false}}), & \text{otherwise.} \end{cases}$$

  Notice that w_V is always the label of a path in A.

Next, we prove that the above encoding between truth assignments and paths is always unique due to the particular structure of the graph G. Since we basically

	(1) CHARE(a, a^*)	(2) CHARE$(a, a?)$	(3) CHARE(a, w^+)
lab(α)	a	aa	$aaaa$
lab(p^{true})	ab	aaa	aaa
lab(p^{false})	ba	a	aa
N	$(\#^*a^*\$^* \cdots \$^*a^*\#^*)^k$	$(\#?a?a?\$? \cdots \$?a?a?\#?)^k$	$(\#aaaa\$ \cdots \$aaaa\#)^+$
lab(r^{true})	a^*b^*	$aaa?$	$a^+(aa)^+$
lab(r^{false})	b^*a^*	$aa?$	$(aa)^+$
lab(r^{all})	$a^*b^*a^*$	$aa?a?$	a^+

Table 5.5: Paths and subexpressions for cases (1)–(3) from Theorem 5.4.7.

transferred the structure of the respective expression from the proof of [MNS09] into a graph, the formal proof below is analogous to Claim 3.3 in [MNS09]. For the sake of completeness we review the proof adapted for graphs in the following.

Assume that C is a clause from Φ. First, we show that if $V_w \models C$ then $w \in L(F(C))$. Therefore, let $w = \#w_1\$ \cdots \$w_n\#$ be the label of a path from x_A to y_A in A such that $V_w \models C$ and let $F(C) = \#e_1\$ \cdots \$e_n\#$ be as previously defined. Then we distinguish three cases for $j \leq n$:

- If x_j occurs positively in C then $e_j = r^{\text{true}}$. Since $V_w \models C$ we know that $V_w(x_j) = \mathit{true}$ and, by definition of V_w, it follows that $w_j = \text{lab}(p^{\text{true}})$. Thus, we get that $w_j \in L(r^{\text{true}}) = L(e_j)$ by condition (P2).

- If x_j occurs negatively in C then $e_j = r^{\text{false}}$. Since $V_w \models C$ we know that $V_w(x_j) = \mathit{false}$ and, by definition of V_w, it follows that $w_j = \text{lab}(p^{\text{false}})$. Thus, we get that $w_j \in L(r^{\text{false}}) = L(e_j)$ by condition (P3).

- If x_j does not occur in C then $e_j = r^{\text{all}}$. Since $w_j \in \{\text{lab}(p^{\text{true}}), \text{lab}(p^{\text{false}})\}$ we know that $w_j \in L(e_j)$ by condition (P4).

It follows that $w_j \in L(e_j)$ for each $j = 1, \ldots, n$ and, therefore, $w \in L(F(C))$.

Second, we show that if $w_V \in L(F(C))$ then $V \models C$. Towards a contradiction, let V be a truth assignment that does not make clause C *true*. We show that $w_V \notin L(F(C))$ and distinguish two cases in the following:

- There exists an x_j that occurs positively in C but $V(x_j)$ is *false*. By definition we know that the component e_j in $F(C)$ is r^{true} and that the component w_j in w_V is lab(p^{false}). By condition (P3) it follows that $w_j \notin L(r^{\text{true}})$. Therefore, we have that $w_V \notin L(F(C))$.

- There exists an x_j that occurs negatively in C but $V(x_j)$ is *true*. By definition we know that the component e_j in $F(C)$ is r^{false} and that the component w_j in w_V is lab(p^{true}). By condition (P2) it follows that $w_j \notin L(r^{\text{false}})$. Therefore, we have that $w_V \notin L(F(C))$.

This proves that the above encoding between truth assignments and paths is unique.

Instantiation of the meta-reduction for the cases (1) to (3): We instantiate the meta-reduction for the cases (1) to (3) by the paths and subexpressions that are specified in Table 5.5. It is straightforward to verify that each instantiation fulfills the conditions (P1) to (P4). Moreover, notice that all expressions are defined over the fixed alphabet $\{a, b, \$, \#\}$. Before we finally prove (*) for the cases (1) to (3), we need to prove the following additional properties about the relation between paths in G and the expression r:

(a) $\mathrm{lab}(B)^i \in L(N)$ for every $i = 1, \ldots, k$,

(b) $\mathrm{lab}(B) \in L(F(C_i))$ for every $i = 1, \ldots, k$, and

(c) for a label w of a path from x_A to y_A in A it holds that if there exists a path p from x to y in G with $\mathrm{lab}(p) = \mathrm{lab}(B)^k \cdot w \cdot \mathrm{lab}(B)^k \in L(r)$ then $w \in F(C_i)$ for some $i = 1, \ldots, k$.

Again, these properties basically come from Martens et al. and are transferred from the original proof (see Claim 3.4. in [MNS09]) to our graph setting. For the sake of completeness we review the proof in the following with a notation that is consistent to our setting.

Properties (a) and (b) hold for each case (1) to (3) immediately by their definitions and the definitions of conditions (P1) to (P4). Therefore, it remains to prove that condition (c) holds.

Let p be a path as desired in (c), i.e., a path from x to y in G such that $u^k w u^k \in L(r)$ with $u = \mathrm{lab}(B)$ is the label of p. We need to show that the subword w is in the language of some expression $F(C_i)$. To this end, we use that every word u, w, and every word in some $L(F(C_i))$ is of the form $\#z\#$ where $z \in \{a, b, \$\}^+$. In addition, every word in the language $L(N)$ is of the form $\#z_1\#\#z_2\#\cdots\#z_\ell\#$ where $z_1, \ldots, z_\ell \in \{a, \$\}^+$ or a subsequence thereof. By the precise definitions of the subwords $z, z_1, \ldots, z_\ell$ for each case we get that $u^k w u^k \in L(r)$. Moreover, we know that either $w \in F(C_i)$ for some i or w is entirely contained in the language of a sub-expression of N because no word $z, z_1, \ldots, z_\ell$ contains "#".

In the following we distinguish between cases (1)–(2) and case (3) to prove that w is always in the language of some $F(C_i)$. Remember that, by assumption, $\mathrm{lab}(p) = u^k w u^k$.

– *Cases (1–2):* We know that words in $L(N)$ contain at most k subwords z_j, i.e., $\ell \leq k$. Therefore, the word $u^k w$ (or a longer prefix of $\mathrm{lab}(p)$) cannot be in the language $L(N)$ for the left occurrence of N in r because $u^k w$ contains at least $k + 1$ subwords of the form z_j. For the same reason the word $w u^k$ (or a longer suffix of $\mathrm{lab}(p)$) cannot be in the language of $L(N)$ for the right occurrence of N in r. Therefore, the subword w has to be in the language of some $F(C_i)$.

– *Case (3):* By definition we have that $\ell \geq 1$. However, notice that every word in $L(F(C_1) \cdots F(C_k)N)$ contains at least $k+1$ subwords of the form z_j. Towards a contradiction, assume that the word $u^k w$ (or a longer prefix of $\mathrm{lab}(p)$) is in the language $L(N)$ for the left occurrence of N in r. Then, it follows that u^k (or the remaining suffix of $\mathrm{lab}(p)$) is too short to be in the language $L(F(C_1) \cdots F(C_k)N)$. We can argue analogously that the word wu^k (or a longer suffix of $\mathrm{lab}(p)$) cannot be in the language $L(N)$ for the right occurrence of N in r. Therefore, the subword w has to be in the language of some $F(C_i)$.

This concludes the proof of property (c) for the cases (1) to (3). We are now ready to prove (*) for these cases.

Therefore, we show that every path of length at most *max* in G from x to y that matches r corresponds to exactly one truth assignment that satisfies Φ and vice versa. Formally, we define a bijection φ between paths from x to y of length at most *max* in G and truth assignments for Φ. Let p be an arbitrary path from x to y in G. Since *max* is at most the number of nodes in G and every path in G can visit every node at most once (G is acyclic), every such path p has length at most *max*. Let $\mathrm{lab}(p) = \mathrm{lab}(B)^k \cdot w \cdot \mathrm{lab}(B)^k$. We define $\varphi(p) := V_w$ where V_w is the truth assignment as previously defined. Notice that there are exactly 2^n paths from x to y and the same amount of truth assignments for Φ. Moreover, by the definition of V_w and the structure of G it is immediate that φ is a bijection. In addition, φ^{-1} maps each truth assignment V onto the path labeled w_V where w_V is as previously defined. It follows from properties (a) to (c) that φ is a bijection between the paths from x to y that have at most length *max* and match r, and the truth assignments that satisfy Φ. This concludes the proof for the cases (1) to (3).

Modifying the meta-reduction for the cases (4) to (6): Next, we prove the lower bounds for the cases (4) to (6). Therefore, we have to slightly modify the given meta-reduction. The main difference of these cases and the cases (1) to (3) is that we will not use a fixed size alphabet. Instead, we use, for $j = 1, \ldots, n$,

– symbols b_j and c_j,

– paths p_j^{true} and p_j^{false} (instead of the paths p^{true} and p^{false}), and

– expressions r_j^{true}, r_j^{false}, and r_j^{all} (instead of expressions r^{true}, r^{false}, and r^{all}).

Moreover, we require that, for every j, these expressions fulfill properties (P1) to (P4) where $p^{\mathrm{true}}, p^{\mathrm{false}}, r^{\mathrm{true}}, r^{\mathrm{false}}$, and r^{all} are replaced by p_j^{true}, p_j^{false}, r_j^{true}, r_j^{false}, and r_j^{all}, respectively. To this end, we need to redefine the structure of the graph G and the expression r. The structure of the graph G in the meta-reduction for the cases (4) to (6) is depicted in Figure 5.7.

subgraph	(4) CHARE$(a, (+a^+))$	(5) CHARE$(a, (+a)^+)$	(6) CHARE$((+a), a^+)$
lab(α)	a	a	a
lab(p_j^{true})	b_j	b_j	b_j
lab(p_j^{false})	c_j	c_j	c_j
N	a^+	a^+	a^+
lab(r_j^{true})	$(a^+ + b_j^+)$	$(a + b_j)^+$	$(a + b_j)$
lab(r_j^{false})	$(a^+ + c_j^+)$	$(a + c_j)^+$	$(a + c_j)$
lab(r_j^{all})	$(a^+ + b_j^+ + c_j^+)$	$(a + b_j + c_j)^+$	$(a + b_j + c_j)$

Table 5.6: Paths and subexpressions for cases (4)–(6) from Theorem 5.4.7.

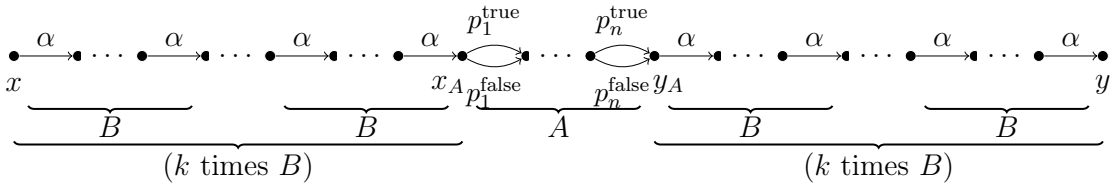


Figure 5.7: The graph G from the proof of Theorem 5.4.7 for the cases (4)–(6).

Basically, the graph has the structure B^kAB^k where B is a path consisting of n concatenations of the subpath α. The subgraph A is analogous to the cases (1) to (3) except that now we use distinct subpaths p_j^{true} and p_j^{false} for each $j = 1, \ldots, n$.

The expression r is defined as $NF(C_1) \cdots F(C_k)N$ where for a clause C of Φ the subexpression $F(C)$ is defined as follows:

$$e_1 \cdots e_n$$

where, for each $j = 1, \ldots, n$,

$$e_j := \begin{cases} r_j^{\text{false}}, & \text{if } x_j \text{ occurs negated in } C, \\ r_j^{\text{true}}, & \text{if } x_j \text{ occurs positive in } C, \\ r_j^{\text{all}}, & \text{otherwise.} \end{cases}$$

Instantiation of the meta-reduction for the cases (4) to (6): We instantiate the meta-reduction for the cases (4) to (6) by the paths and expressions that are specified in Table 5.6. It is straightforward to verify that the conditions (P1) to (P4) are fulfilled for each of the cases.

We now prove that the reduction is correct for the cases (4) to (6). Again, we can associate the paths from x_A to y_A in G with truth assignments for Φ:

– For a path with label $w = w_1 \cdots w_n$ we define V_w as the truth assignment where, for every $j = 1, \ldots, n$ and $w_j \in \{p_j^{\text{true}}, p_j^{\text{false}}\}$,

$$V_w(x_j) := \begin{cases} \mathit{true}, & \text{if } w_j \in L(r_j^{\text{true}}), \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

– For a truth assignment V we define the word $w_V = w_1 \cdots w_n$ such that, for each $j = 1, \ldots, n$,

$$w_j = \begin{cases} p_j^{\text{true}}, & \text{if } V(x_j) = \mathit{true}, \\ p_j^{\text{false}}, & \text{otherwise.} \end{cases}$$

Notice that w_V is always the label of a path in A.

Analogously to cases (1) to (3) it can be proved that the encoding of different paths in the graph and different truth assignments for the formula Φ is unique. Properties (a) and (b) hold for the cases (4) to (6) directly by their definition.

Analogously to [MNS09] we argue that property (c) holds for every instantiated case (4) to (6) in the following. Let p be a path as desired in (c) from x to y in G such that $u^k w u^k \in L(r)$ with $u = \text{lab}(B)$ is the label of p. We need to show that the subword w is in the language of some expression $F(C_i)$. For every $j = 1, \ldots, n$ let Σ_j denote the set $\{b_j, c_j\}$. Observe that the word w is of the form $z_1 \cdots z_n$ where $z_j \in \Sigma_j^+$ for every $j = 1, \ldots, n$. Moreover, no words in $L(N)$ contain symbols from Σ_j for any $j = 1, \ldots, n$. Hence, there does not exist a subword of w that is in the language $L(N)$. Consequently, there exists a subword $v_1 w v_2$ of $u^k w u^k$ which is in the language $L(F(C_1) \cdots F(C_k))$. Notice that every word in every $F(C_i)$ for $i = 1, \ldots, k$ is of the form $z'_1 \ldots z'_n$, where each z'_j is a word in $(\Sigma_j \cup \{a\})^+$. By the structure of w and since all alphabets Σ_j are distinct, it follows that w is in the language of some expression $F(C_i)$. This proves that (c) holds for cases (4) to (6).

Finally, the proof of $(*)$ is analogous to the proof for the cases (1) to (3). In addition, the constructed graphs G are always acyclic such that every case is #P-hard even if G is acyclic. This concludes the proof of the lower bounds for the cases (1) to (6).

It remains to prove (7), i.e., Counting under regular path semantics is #P-complete for RE($\#, !, \neg, \bullet$). The lower bound directly holds from any of the proofs for the cases (1) to (6). The upper bound comes from Theorem 5.4.6. □

To conclude we remark that Arenas et al. studied a similar problem as above in [ACP12]. They also examined the tractability of SPARQL 1.1 queries, though they focus was on a different part of the SPARQL specification to formalize their interpretation of the W3C's semantics. In particular, they showed that Counting under regular path semantics is complete for the complexity class *spanL*. This class was introduced by Alvarez and Jenner [AJ93] and, informally, contains all functions f such that $f(x)$ is the size of the set of different output values of accepting paths of a LOGSPACE Turing machine. Opposed to that, functions in #P compute the size of the multiset containing all different output values of accepting paths of a polynomial-time Turing machine. Moreover, it is known that if #P = spanL then it follows that NLOGSPACE = P = NP [AJ93]. Therefore, it seems quite hazardous (at first glance) that Counting under regular path semantics is complete for the class #P as well as for the class spanL. However, both results are correct since we use polynomial Cook reductions to obtain our results and it is very unlikely that spanL is closed under such kind of reductions.

5.4.2 Counting under Simple Walk Semantics

In this section we investigate how the complexity of COUNTING changes under simple walk semantics. We will see that the situation is even more severe than in Section 5.3.3 for the EVALUATION problem. Although our results also include a polynomial-time result for a very small class of CHAREs, we show that COUNTING turns #P-complete as soon as the Kleene star (*) or plus ($^+$) operator is used. To complete the picture we show that COUNTING remains in #P for the full fragment of SPARQL regular expressions, i.e., expressions in RE(#, !, •).

5.4.8 Theorem. COUNTING *under simple walk semantics is*

(1) in P for the class CHARE(a, (+a)),

(2) is #P-complete for expressions a^* and a^+,

(3) is #P-complete for the class CHARE(a, a?), and

(4) is #P-complete for the class RE(#, !, •).

Proof. (1) The result holds since for this fragment simple walk semantics coincides with regular path semantics and every expression in CHARE(a, (+a)) can be translated to a DFA^w in polynomial time.

(2) We show that the problem is #P-hard by a reduction from the #P-complete problem #SIMPLEPATHS (see, e.g., Section 2.4). Notice that #SIMPLEPATHS is also #P-hard when the source and target node for the input are different. We restrict ourselves to such instances in the following to prove that the hardness result also holds for simple walks.

Given an s-t graph $G = (V, E, x, y)$ with $x \neq y$ and a number *max* in unary, we construct an edge-labeled s-t graph $G' = (V, E', x, y)$ by labeling each edge with a. The number of simple walks from x to y in G of length at most *max* is equal to the number of paths from x to y in G' of length at most *max* that match the regular expression a^*. The reduction for the expression a^+ is similar. Notice that this immediately implies that COUNTING under simple walk semantics is #P-hard for the classes CHARE(a, a^*), CHARE(a, w^+), CHARE(a, (+a^+)), CHARE(a, (+a)$^+$), and CHARE(a^+, (+a)).

(3) The lower bound is immediate from the observation that the reduction for regular path semantics from Theorem 5.4.7 applies here as well.

(4) Finally, we show how to solve COUNTING for RE(#, !, •) in #P. To this end, we construct an NP Turing machine that guesses a path from x to y of length at most *max* in the graph and then tests whether it matches the expression under simple walk semantics. For the path the Turing machine needs to guess the nodes as well as the labels. Since the number *max* is given in unary it can guess the entire path in polynomial time. Moreover, notice that the entire path does not have to be simple walk because only subexpressions of the form s^*

and s^+ have to be matched by simple walks. Afterwards, the Turing machine runs the dynamic programming procedure for words (i.e., the one given in the proof of Theorem 5.3.4) on the path, but during the computation it removes all pairs in all relations that do not correspond to simple walk semantics. In particular, it removes all pairs (x, y) in a relation eval(s^*) and eval(s^+) such that the subpath from x to y is not a simple walk. Otherwise, the procedure remains unchanged. Since the only nondeterminism in the algorithm comes from guessing the path, the number of accepting computations of the Turing machine corresponds to the number of paths of length at most *max* that match the expression. □

5.5 The Complexity of the Finiteness Problem for Property Paths

For a regular expression r and a graph $G = (V, E, x, y)$ there cannot exist infinitely many paths p in G such that p matches r under simple walk semantics. Under regular path semantics this can be the case. In order to be able to make a fair comparison between the complexity of counting paths for regular path semantics and for simple walk semantics, we study the FINITENESS problem. However, we only consider regular path semantics in the following.

We start with a simple observation. Using the product graph from Definition 5.4.3 we can test in polynomial time whether there is a path p from x to y that is labeled by a word uvw such that a subpath of p labeled with $v \in \Sigma^*$ is a cycle and $uv^kw \in L(r)$ for every $k \in \mathbb{N}$. If there is such a cycle then we return that there are infinitely many paths. This solves FINITENESS for RE in time $O(|r| \cdot |G|)$.

We now show how to solve FINITENESS for RE$(\#, !, \bullet)$ in polynomial time by adapting the algorithm for EVALUATION in Section 5.3.1.

5.5.1 Theorem. FINITENESS *for RE(*$\#, !, \bullet$*) is in P.*

Proof. Let $G = (V, E, x, y)$ be a graph, x and y be nodes in G, and r be an RE$(\#, !, \bullet)$. To prove the assumption we construct an algorithm that adapts the polynomial-time algorithm from Section 5.3.1 such that it additionally annotates the length of the longest paths associated to a pair of nodes in every computed relation. We use a pumping argument to argue that the number of paths from x to y that match r is infinite if and only if there is a *very long* path p from x to y such that lab$(p) \in L(r)$.

We first formally define when a path p is considered to be "very long", i.e., we prove an upper bound of the length for paths from x to y that match r. More precisely, we show that it suffices to consider paths of at most exponential length in $|r|$. To this end, we translate the RE$(\#, !, \bullet)$ r to an RE$(!, \bullet)$ r' by unfolding the counting operators. (That is, we replace subexpressions of the form $s^{k,k}$ with a concatenation of k expressions s.) The size of the resulting expression r' is at most exponential in the size of r. Let N be an NFA^w for $L(r)$ which is obtained

from r' via the Glushkov construction. We know that N has polynomial size in r', i.e., exponential size in r.

Next, we examine the product graph $G_{x,y} \times N = (V_{G,N}, E_{G,N}, x_{G,N}, Y_{G,N})$ (see Definition 5.4.3). In this graph the following holds:

> There are infinitely many paths from x to y in G that match r if and only if there are infinitely many paths from $x_{G,N}$ to some node in $Y_{G,N}$ in $G_{x,y} \times N$.

The latter holds if and only if there is a cycle in $G_{x,y} \times N$ or, more precisely, there exists a path from $x_{G,N}$ to some node in $Y_{G,N}$ of length at least $|G| \cdot |N| + 1$ in $G_{x,y} \times N$. Since the size of N is exponential in $|r|$ it follows that we only need to consider paths of at most exponential length in $|r|$.

In the following we adapt the algorithm that solves Evaluation for RE$(\#, !, \bullet)$ (see Section 5.3.1) such that it remembers the lengths of already computed paths while trying to find more paths that are as long as possible. To this end, we define $M := |G| \cdot |N| + 1$ as upper bound on the length of relevant paths. Once a path becomes longer than M the algorithm simply remembers that the path is long enough. The algorithm works as follows. Let r be an RE$(\#, !, \bullet)$ and let $G = (V, E, x, y)$ be a graph. It computes, for all subexpressions s of r, the ternary relation

$$\text{eval}_c(s) \subseteq V \times V \times \{0, \ldots, M\}$$

such that if $(u, v, i) \in \text{eval}_c(s)$ then there exists a path from u to v that matches s and has at least length i. More precisely, the following holds for eval_c:

- for each $(u, v) \in V \times V$ there is at most one tuple of the form $(u, v, i) \in \text{eval}_c(s)$,
- if there is a path from u to v in G that matches s then there exists a tuple $(u, v, i) \in \text{eval}_c(s)$,
- for each $(u, v, i) \in \text{eval}_c(s)$ with $i \in \{0, \ldots, M-1\}$ there is a path from u to v of length i in G that matches s, and
- there is a path from u to v in G of length at least M that matches s if and only if $(u, v, M) \in \text{eval}_c(s)$.

The relation $\text{eval}_c(s)$ can be computed inductively on the structure of r by one bottom-up pass of the syntax tree of r and joining already computed relations for the subexpressions on the way. All possible cases for the computation are depicted in Table 5.7. Basically, the definition of eval_c from Table 5.3 is an extension of the relation eval (see Table 5.3) to a ternary relation. During the computation of eval_c the algorithm checks the length of the longest paths for every subexpression in every step. Since we already showed that it is sufficient to keep track of paths upto length M we change the definition of the original relation eval accordingly. That means the algorithm stores the exact length of a path as long as it is smaller than M. Once length M is reached it only stores the information "long enough" (i.e., entries annotated with M). Next, we present the differences between the relation eval and eval_c in more detail.

type of expression	computation of eval_c
$\text{eval}_c(\emptyset)$	$\emptyset$
$\text{eval}_c(\varepsilon)$	$\{(u,u,0) \mid u \in V\}$
$\text{eval}_c(\bullet)$	$\{(u,v,1) \mid \exists a \in \Delta \text{ with } (u,a,v) \in E\}$
$\text{eval}_c(a)$, for $a \in \Delta$	$\{(u,v,1) \mid (u,a,v) \in E\}$
$\text{eval}_c(!(a_1 + \cdots + a_n))$	$\{(u,v,1) \mid \exists a \in \Delta \backslash \{a_1,\ldots,a_n\} \text{ with } (u,a,v) \in E\}$
$\text{eval}_c(s_1 + s_2)$	$\text{eval}_c(s_1) \cup_{\min} \text{eval}_c(s_2)$
$\text{eval}_c(s_1 \cdot s_2)$	$\{(u,v,i) \mid i = \max\{j \mid \exists z : (u,z,k) \in \text{eval}_c(s_1), (z,v,\ell) \in \text{eval}_c(s_2), \text{ and } j = \min(k+\ell, M)\}\}$
$\text{eval}_c(s?)$	$\text{eval}_c(s + \varepsilon)$
$\text{eval}_c(s^*)$	$\text{eval}_c((s+\varepsilon)^M)$
$\text{eval}_c(s^+)$	$\text{eval}_c(s \cdot s^*)$
$\text{eval}_c(s^k)$	- $k = 1$: $\text{eval}_c(s^k) = \text{eval}_c(s)$ - even k: $\text{eval}_c(s^k) := \text{eval}_c(s^{k/2} \cdot s^{k/2})$ - odd k: $\text{eval}_c(s^k) := \text{eval}_c(s \cdot s^{k/2} \cdot s^{k/2})$
$\text{eval}_c(s^{k,\infty})$	$\text{eval}_c(s^k \cdot s^*)$
$\text{eval}_c(s^{k,\ell})$, for $\ell \neq \infty$	$\text{eval}_c(s^k \cdot (s?)^{\ell-k})$

Table 5.7: Inductive definition of the relation eval_c.

For the base cases we additionally store the length of the considered path, i.e., paths that match ε are of length 0 and paths that match $\bullet$-symbols, Δ-symbols, and restricted negation expressions are of length 1.

We define $\text{eval}_c(s_1+s_2) = \text{eval}_c(s_1) \cup_{\min} \text{eval}_c(s_2)$ where, for two ternary relations $R_1, R_2 \subseteq V \times V \times \{0,\ldots,M\}$, the set $R_1 \cup_{\min} R_2$ contains all tuples (u,v,i) such that either

- $(u,v,i) \in \text{eval}_c(s_1)$ and $\nexists j : (u,v,j) \in \text{eval}_c(s_2)$,
- $(u,v,i) \in \text{eval}_c(s_2)$ and $\nexists j : (u,v,j) \in \text{eval}_c(s_1)$, or
- $(u,v,k) \in \text{eval}_c(s_1), (u,v,\ell) \in \text{eval}_c(s_2)$, and $i = \min(\max(k,\ell), M)$.

In this way we guarantee that the relation $\text{eval}_c(s_1 + s_2)$ stores, for each pair of nodes (u,v), the length of the longest paths from u to v in the graph that matches the expression $s_1 + s_2$.

For the concatenation case $\text{eval}_c(s_1 \cdot s_2)$ the relation can be computed by searching for the best (intermediate) node z such that there exist a path p_1 from u to z that matches s_1, a path p_2 from z to v that matches s_2, and no other such paths p_1' and p_2' with $|p_1'p_2'| > |p_1p_2|$.

Notice that for the Kleene-star case $\text{eval}_c(s^*)$ it is not necessary to compute the transitive closure of the relation. It is sufficient to test whether a path of length M can be found in the relation $\text{eval}_c((s+\varepsilon)^M)$, which can be done in polynomial time by adapting the fast squaring method as specified in Table 5.7.

Given a graph G, nodes x and y, and an RE$(\#, !, \bullet)$ r, the algorithm returns *true* for FINITENESS if and only if $\text{eval}_c(r)$ contains the triple (x, y, M). Analogously to the computation of eval the relation eval_c can be computed in polynomial time. The proof of correctness is a straightforward induction and analogous to the proof of Theorem 5.3.2. □

Next, we show that the complexity of the FINITENESS problem becomes non-elementary once unrestricted negation is allowed in regular expressions. The result is obtained by proving that FINITENESS is at least as hard as testing emptiness for the considered class of regular expressions.

5.5.2 Lemma. *Let C be a class of regular expressions r over a finite alphabet Σ such that testing whether $\varepsilon \in L(r)$ is in P. Then there exists a polynomial reduction from the emptiness problem for C-expressions to* FINITENESS *for C-expressions.*

Proof. Let r be a C-expression over Σ. For the reduction we construct a graph $G = (V, E, x, y)$ and a C-expression s such that $L(r) = \emptyset$ if and only if FINITENESS is *true* for s and G. We distinguish two cases:

- If $\varepsilon \in L(r)$ then we know that $L(r) \neq \emptyset$. In this case we return the expression a^* and the graph $G = (V, E, x, x)$ with $V = \{x\}$ and $E = \{(x, a, x)\}$.

- If $\varepsilon \notin L(r)$ then we return the expression r^* and the graph $G = (V, E, x, x)$ with $V = \{x\}$ and $E = \{(x, a, x) \mid a \in \Sigma\}$.

We now prove that the reduction is correct. If $\varepsilon \in L(r)$ then it holds that $L(r) \neq \emptyset$ and FINITENESS for a^* and G is *false*. If $\varepsilon \notin L(r)$ then it holds that either $L(r) = \emptyset$ or $L(r)$ contains at least one word w with $|w| > 0$. In the case that $L(r) = \emptyset$ we know that $L(r^*) = \{\varepsilon\}$. Thus, there exists only one path with length 0 in G that matches r^*, i.e., FINITENESS returns *true*. In the case that $L(r) \neq \emptyset$ it follows that $w^i \in L(r^*)$ for all $i \geq 1$. Thus, for every $i \geq 1$, there is a path p_i with $\text{lab}(p_i) = w^i$ in G. Since all paths p_i and p_j for $i \neq j$ are different there exist infinitely many paths p in G that match r^*, i.e., FINITENESS returns *false*. □

For an RE$(\neg)$ r we can test whether $\varepsilon \in L(r)$ in linear time by traversing the syntax tree of r. Since Stockmeyer showed that the language emptiness problem for the class RE$(\neg)$ of generalized regular expressions is non-elementary [Sto74] the following statement holds by applying Lemma 5.5.2.

5.5.3 Theorem. FINITENESS *for RE$(\neg)$ is decidable but non-elementary.*

type of expression	computation
$[\![\varepsilon]\!]_G$	$\{(u,u) \mid u \in V\}$
$[\![\bullet]\!]_G$	$\{(u,v) \mid (u,a,v) \in E \text{ for some } a \in \Delta\}$
$[\![a]\!]_G$	$\{(u,v) \mid (u,a,v) \in E\}$
$[\![a^-]\!]_G$	$\{(u,v) \mid (v,a,u) \in E\}$
$[\![!(a_1 + \cdots + a_n)]\!]_G$	$\{(u,v) \mid \exists a \in \Delta \backslash \{a_1, \ldots, a_n\} \text{ with } (u,a,v) \in E\}$
$[\![r + s]\!]_G$	$[\![r]\!]_G \cup [\![s]\!]_G$
$[\![r \cdot s]\!]_G$	$[\![r]\!]_G \bowtie [\![s]\!]_G$
$[\![r^*]\!]_G$	reflexive and transitive closure of $[\![r]\!]_G$
$[\![r^{k,\ell}]\!]_G$	$([\![r]\!]_G)^k \bowtie ([\![r + \varepsilon]\!]_G)^{\ell - k}$
$[\![\langle r \rangle]\!]_G$	$\{(u,u) \mid \exists z : (u,z) \in [\![r]\!]_G\}$

Table 5.8: Regular paths semantics of NREs with respect to a graph $G = (V, E)$.

5.6 SPARQL and Nested Regular Expressions

In this section we show that the complexity upper bound for Evaluation from Theorem 5.3.2 also translates to more complex queries. Therefore, we consider *nested regular expressions (NREs)* which are regular expressions that are enhanced by a mechanism to branch out in the graph. For such expression it is already known that they can be evaluated in linear-time using the product construction for automata [AI00, PAG10]. In fact, since nested regular expressions are a fragment of propositional dynamic logic (PDL), linear-time evaluation of such expressions already holds by linear-time evaluation of PDL [CS93, AI00]. Nested regular expressions (without counting operators) have also been studied in the context of SPARQL queries by Pérez et al. [PAG10]. They showed that nested regular expressions can be evaluated over RDF graphs in linear-time in the size of the expression and the graph. Since the complexity of evaluating them is essentially not worse than for ordinary regular expressions, NREs are an interesting extension for SPARQL queries.

Next, we show that nested regular expressions under regular path semantics remain tractable even when they are built over SPARQL property path expressions (i.e., over RE($\#, !, \bullet$) expressions). To this end, we equip NREs with counting operators, the wildcard symbol ($\bullet$), and the restriction negation operator (!) and show that Evaluation for these expressions can be solved in polynomial time.

5.6.1 Definition. Let Δ be an infinite set of symbols. The set of *SPARQL nested regular expressions* or *NRE($\#, !, \bullet$)* over Δ is defined as follows:

$$r, s := \varepsilon \mid \bullet \mid a \mid a^- \mid !(a_1 + \cdots + a_n) \mid (r + s) \mid (r \cdot s) \mid (r)^* \mid (r)^{k,\ell} \mid \langle r \rangle$$

where $a, a_1, \ldots, a_n \in \Delta$, $k \in \mathbb{N}$, and $\ell \in \{k, k+1, \ldots, \infty\}$.

The class NRE($\#, !, \bullet$) strictly generalizes SPARQL regular expressions by extending them with the *nesting operator* $\langle \cdot \rangle$. The semantics of NRE($\#, !, \bullet$) expressions under regular path semantics is defined in Table 5.8. Notice that we added an additional operator a^- for $a \in \Delta$. This expression allows to navigate through

a directed a-edge in the reverse direction and is also inspired by a corresponding operator in the SPARQL working draft [HS12]. We illustrate an example instance of EVALUATION for nested regular expressions in the following.

5.6.2 Example. Consider the expression $r = a\langle(b^{2,2})^*c\rangle d$ and the graph G from Figure 5.8. For r and a pair (x, y) of nodes in G we know that EVALUATION is *true* for (x, y) if and only if there is a path from x to y over some node z labeled with ad and there is a path from z to some node in G that is labeled with a word in $L((b^{2,2})^*c)$. Notice that none of these nodes has to be distinct to another under regular path semantics.

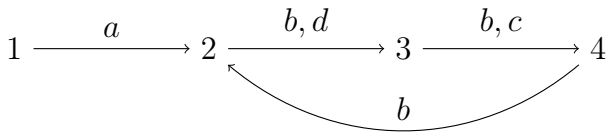


Figure 5.8: An edge-labeled graph G

For the graph from Figure 5.8 EVALUATION is *true* for $(1, 3)$ since there is a path from node 1 to node 3 labeled with ad and a path from node 2 to node 4 labeled with $bbbbc \in L((b^{2,2})^*c)$. (The path from 2 to 4 is not a simple walk.)

All our previous lower bounds from Chapter 5 transfer to NRE$(\#, !, \bullet)$ expressions immediately. Next, we show that EVALUATION for NRE$(\#, !, \bullet)$ under regular path semantics can be solved in polynomial time by adapting the algorithm from Section 5.3.1. We only study the EVALUATION problem for NRE$(\#, !, \bullet)$ under regular path semantics because EVALUATION under simple walk semantics and COUNTING under regular path semantics is already intractable for very restricted classes of regular expressions (without nesting).

5.6.3 Theorem. *Let G be a graph and r be an NRE$(\#, !, \bullet)$.* EVALUATION *for r and G under regular path semantics is in time $O(|r| \cdot |V|^3)$.*

Proof. To prove the assumption we construct an algorithm that, given $G = (V, E, x, y)$ and r, decides whether $(x, y) \in [\![r]\!]_G$ in polynomial time. Basically, this algorithm is an extension of the algorithm in the proof of Theorem 5.3.2. Therefore, we extend the previous algorithm that computes the relation $\text{eval}(s)$ by two additional cases:

- for every $a \in \Delta$, $\text{eval}(a^-) := \{(v, u) \mid (u, a, v) \in E\}$, and
- $\text{eval}(\langle s\rangle) := \{(u, u) \mid \exists z : (u, z) \in \text{eval}(s)\}$.

Both cases can be computed in linear time such that the entire algorithm still runs in polynomial time. We argue correctness similar to the proof of Theorem 5.3.2. More precisely, we prove that the following invariants (I1) and (I2) hold:

For each subexpression $s \neq \langle s_1\rangle$ of r we have

$$(u, v) \in \text{eval}(s) \Leftrightarrow \exists \text{ path } p \text{ in } G \text{ from } u \text{ to } v \text{ such that } \text{lab}(p) \in L(s). \qquad \text{(I1)}$$

For each subexpression $s = \langle s_1 \rangle$ of r we have

$$(u, v) \in \text{eval}(s) \Leftrightarrow (u = v) \wedge (\exists \text{ node } z, \text{ path } p \text{ from } u \text{ to } z \text{ in } G \text{ such that } \text{lab}(p) \in L(s_1)). \quad \text{(I2)}$$

The proof of invariant (I1) is analogous to the proof of the invariant (I) in Theorem 5.3.2 and the proof of (I2) is immediate from the above definition of $\text{eval}(\langle s \rangle)$. Finally, EVALUATION for $G = (V, E, x, y)$ and r is *true* if and only if $(x, y) \in \text{eval}(r)$. □

To conclude our study of SPARQL queries we remark that we believe the results in this chapter to be relevant in a larger context, too. In particular, foundational research on formal languages for querying graph databases can benefit from the proven lower and upper bounds. For example, in [LMV13], the authors examine the behavior and applicability of XPath-like query languages for graphs which are closely related to nested regular expressions. In addition, they showed that efficient evaluation can even be extended to variants of nested regular expressions that use a more powerful negation.

5.7 Recent Developments in the SPARQL 1.1 Specification

At the time we investigated the computational complexity of SPARQL queries, the W3C specification for SPARQL 1.1 was still under development. We therefore started studying regular path semantics which seems to be, based on our observations, a recommendable alternative from a complexity point of view.

Opposed to that, the NP-complete and #P-complete data complexities make the semantics recommended by the W3C in [HS12] highly problematic from a computational complexity perspective. More precisely, two orthogonal requirements render the evaluation of simple queries of the form

```
SELECT ?x, ?y WHERE {?x r ?y}
```

computationally difficult: the *simple walk requirement* and the *path counting requirement* (see Note 5.2.2 and 5.4.1, respectively). In 2012 our work as well as the work of Arenas et al. [ACP12] proposed that the W3C should use a semantics for property paths without the simple walk and path counting requirement. These proposals were taken into consideration by the W3C and influenced the following changes to the official W3C recommendation for SPARQL 1.1 in 2013 [HS13]:

(1) The ambiguity between the definitions of **ZeroOrMorePath** and **OneOrMorePath** in Sections 18.4 and 18.5 in [HS12] has been removed and the simple walk requirement has been dropped.

(2) The path counting requirement has been dropped for subexpressions of the form r^* and r^+.

(3) Subexpressions of the form $r^{k,\ell}$ with $k, \ell \in \mathbb{N}$ are no longer part of the property path syntax.

We feel that change (1) is very welcome. Change (2) will undeniably make the evaluation of SPARQL property paths more efficient. However, we feel that, even after this change, the semantics of property paths may be rather counter-intuitive since the change does not completely remove multiset semantics for property paths. Instead, change (2) means that, for a given regular expression r, its subexpressions of the form s^* and s^+ are evaluated under set semantics and the others under multiset semantics. As such, it is difficult to understand the number of occurrences of tuples in the output of a query. For example, the combination of set semantics and multiset semantics has as a result that a^+ is not equivalent to aa^*, since the former always returns tuples with multiplicity one (set semantics) and the latter can return tuples with higher multiplicity due to the partial multiset semantics.

Finally, we feel that change (3) is a pity. We showed that property paths with this feature can be evaluated efficiently in principle (see Theorem 5.3.2 and Corollary 5.3.3). This holds, even though these expressions can be exponentially more succinct than standard regular expressions. Since expressions with counting operators seem to be convenient for the user in practical applications (see, for example, the regexlib regular expression library [Reg]) we feel that it would be a win-win situation to include regular expressions with counting operators in the definition of SPARQL property paths.

6

Enumerating Answers under Updates

Efficient query evaluation is one of the most central problems in databases. In this chapter we investigate the query evaluation problem in the case of queries that return a set of answers instead of a boolean one. This set can be extremely large in general and it may be infeasible to compute the set in its entirety. Therefore, algorithms evolved that do not return all answers at the same time but enumerate the set of answers sequentially to the user. The problems corresponding to such algorithms are known as *query enumeration problems* which attracted some attention in the last decade (see, e.g., [Bag06, Cou09, DS11, KS13a, KS13b, Seg13]). Another database problem that recently got more important again is the *view maintenance problem* for databases. That is, efficiently computing the answer to a fixed query over a database that is subject to updates. In this chapter we construct algorithms that are sensitive to both problems and examine their computational complexity. More precisely, we want to investigate efficient enumeration algorithms for queries that are defined over data that is likely to change.

We call this problem the *incremental enumeration* problem. To solve the problem we construct an algorithm that works as follows. First, the algorithm takes a query M (i.e., an automaton on words or trees) and a data structure d (i.e., a word or a tree) as input and computes an auxiliary data structure. We refer to this computation as *preprocessing*. Second, the algorithm subsequently enumerates all answers in the result $M(d)$ of the query M on d. Here, we are interested in the maximal delay that is needed between two answers. We refer to this delay as the *enumeration delay*. When an update on d occurs (i.e., d is changed to a data structure d') then the algorithm stops in an instant, processes the update by possibly changing the auxiliary data structure, and starts enumerating the answers in $M(d')$ again. For the *update* we are interested in the maximal time that is needed to process the update such that the first answer in $M(d')$ can be computed afterwards.

As far as we know, we are the first to examine incremental enumeration (for k-ary) queries formally. Moreover, it is unclear to us how one can adapt existing enumeration techniques that are related to our setting (see, e.g., [Bag06, Cou09]) to provide algorithms that are sensitive to updates and still efficient. This also holds for constant enumeration delay algorithms using deterministic factorization forests (see, e.g., [KS13b, Seg13]). Colcombet's deterministic factorization forests [Col07], which are itself based on a result of Simon [Sim90], can be used to enumerate answers to certain queries on words or trees with only a constant delay via a good divide-and-conquer technique. However, we believe that the built structures have to be entirely recomputed each time an update occurs. Bojanczyk and Figueira [BF11] consider evolutions $t_1, \ldots, t_m$ of trees (which they call *document evolutions*) and evaluate two-dimensional logics over such sequences. Such logics can express properties of single trees and how such properties evolve over time. (For example, "eventually, every a-node will have a b-child".) They read the input as t_1 followed by a sequence of $m-1$ local updates and give an $O(m \cdot \log n)$ algorithm to decide if a formula is satisfied over the entire evolution (assuming $m > n$). Therefore, in the temporal dimension, the setting in [BF11] is more general than ours: we cannot compare different versions of the tree. Since they are only concerned with satisfaction of a property, they do not consider small delay algorithms for enumerating answers. In dynamic complexity theory, previous research mainly focuses on lower bounds for the incremental evaluation problem (see, e.g., [MSVT94]) or on parallel dynamic computing (see, e.g., [PI97, DS93]).

In the following we examine the incremental enumeration problem for queries that are defined by k-ary nondeterministic finite selecting (tree) automata [NPTT05] and evaluated over words and trees, respectively. It is well-known that these automata can express MSO queries with free node variables [NPTT05]. We formally define them in Section 6.1 and state the problems under consideration (including the considered types of updates) in Section 6.2.

In Section 6.3, we construct an algorithm that solves the incremental enumeration problem for a word of size n and a given query, uses $O(n)$ time for preprocessing, and enumerates all answers to the query with $O(\log n)$ time between two answers. In the case that an update to the word occurs, the algorithm stops in an instant and starts enumerating the answers to the query over the updated data after $O(\log n)$ time. Afterwards, we show how to extend the word algorithm for trees. Therefore, we heavily rely on a technique of Balmin et al. [BPV04] that shows how to decompose a tree automaton and a tree into a set of NFAs and heavy paths such that one can maintain membership of the tree and the tree automaton by evaluating the NFAs over the induced words of the heavy paths. For a tree of size n the original algorithm of Balmin et al. needs $O(n)$ time for the preprocessing and computes the new answer after an update occurred in $O(\log^2 n)$ time. Using this result we construct in Section 6.4 an algorithm that enumerates, for a tree of size n and a query, all answers to the query with $O(\log^2 n)$ enumeration delay and processes updates in $O(\log^2 n)$ time. We note that the complexity results in Sections 6.3 and 6.4 are presented in more detail, i.e., in terms of the size of the data, the arity k of the query, and the number $|Q|$ of states of the selecting automaton.

In Section 6.5 we briefly explain how our results can be extended to work for more expressive semantics, i.e., *multiset semantics*. A recent application of our result from information extraction is given in Section 6.6. Last but not least, we discuss how our results relate to MSO logic as well as XPath query evaluation in Section 6.7. In addition, we provide some interesting open questions for further work on incremental enumeration.

6.1 Tuple Selecting Automata

We use (node- and tuple-) selecting finite automata (see [FGK03, Nev99]) to model queries. In this section we provide formal definitions of these automata which will be used in the remainder of the chapter. Before we define these automata, we need to introduce the following notation. For readability, we often associate to a word w a *set of nodes* $\text{Nodes}(w) = \{v_1, \ldots, v_n\}$ such that each node v_i bears the label $\text{lab}(v_i) = a_i$. Due to the structure of the word, the set of nodes is linearly ordered. Therefore, we often assume that $\text{Nodes}(w) = \{1, \ldots, n\}$ to simplify notation. However, our results do not require that $\text{Nodes}(w) = \{1, \ldots, n\}$. For example, if we want to insert a new node at the beginning of a word w then we do not require that all old nodes should obtain a new label. For nodes v_i, v_j with $1 \leq i \leq j \leq n$ we denote by $w[v_i..v_j]$ the subword $a_i \cdots a_j$ and by $w[v_i]$ the symbol a_i. For a finite alphabet Σ and a word $w \in \Sigma^*$ or tuple $t = (a_1, \ldots, a_k) \in \Sigma^k$ we regularly need the *set of symbols* (or *labels*) occurring in it. We refer to this set as $\text{set}(w)$ or $\text{set}(t)$, respectively. It is defined as $\text{set}(w) := \{a \in \Sigma \mid \exists v \in \text{Nodes}(w), \text{lab}(v) = a\}$ and $\text{set}(t) = \{a_1, \ldots, a_k\}$.

6.1.1 Definition. A *(bottom-up) nondeterministic tree automaton* (or *NTA*) N is a tuple (Q, Σ, δ, F) where Q is a finite set of states, Σ is a finite alphabet, $F \subseteq Q$ is the set of accepting states, and δ is a set of transition rules that are either of the form $(q_1, q_2, a) \to q$ or $a \to q$ for states $q_1, q_2, q \in Q$ and a label $a \in \Sigma$.

A *run* of N on a labeled binary tree t is an assignment $\lambda : \text{Nodes}(t) \to Q$ such that the following holds for every $v \in \text{Nodes}(t)$:

- if v is a leaf then $\text{lab}(v) \to \lambda(v) \in \delta$;
- if v has children v_1 and v_2 then $(\lambda(v_1), \lambda(v_2), \text{lab}(v)) \to \lambda(v) \in \delta$.

A run is *accepting* if $\lambda(r) \in F$ for the root r of t. A run λ *visits* v *in* q if $\lambda(v) = q$. A tree t is *accepted* if there exists an accepting run on t. The set of all accepted trees is denoted by $L(N)$ and is called a *tree language*. Next, we define *node selecting* automata. We give the definition for word automata first. For the remainder of the chapter, let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and $w = a_1 \cdots a_n$ be a word.

6.1.2 Definition. For $k \in \mathbb{N}$, a *k-ary nondeterministic finite selecting automaton* (or *k-NFSA*) M is a pair (N, S) where N is an NFA over Σ with states Q and $S \subseteq Q^k$ is a set of *selecting tuples*.

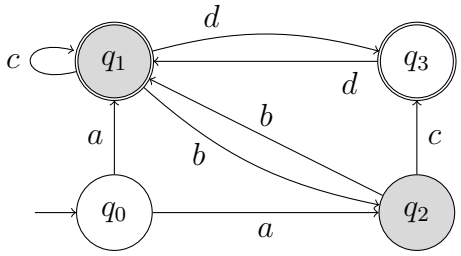


Figure 6.1: 2-NFSA M with $S = \{(q_1, q_2), (q_2, q_1)\}$.

The *size* of a k-NFSA $M = (N, S)$ is defined as $|N| + |S|$. When M reads a word w of length n, it computes a set $M(w)$ of tuples in $\text{Nodes}(w)^k$. More precisely, we define the result $M(w)$ of M on w as follows:

$$M(w) = \{(v_1, \ldots, v_k) \mid \text{ there is an accepting run } r \text{ of } N \text{ on } w \text{ and a tuple } (p_1, \ldots, p_k) \in S \text{ such that } r \text{ visits } v_\ell \text{ in } p_\ell \text{ for every } \ell \in \{1, \ldots, k\}\}.$$

Recall that, for a run $r = q_0 \cdots q_n$ of N such that $q_i \in \delta(q_{i-1}, a_i)$, for every $i \in \{1, \ldots, n\}$, the run r *visits node* i *in state* q_i, denoted $r(i) = q_i$, for each i. Moreover, it holds that if $w \notin L(N)$ then $M(w) = \emptyset$. The corresponding definitions for a *k-ary nondeterministic finite selecting tree automaton (k-NFSTA)* are the same as for k-NFSAs, except that N is an NTA instead of an NFA. We illustrate 2-NFSA M in Figure 6.1. For the word $w = abcd$, we have that

$$M(w) = \{(1, 2), (2, 1), (1, 3), (3, 1), (2, 4), (4, 2)\}.$$

6.2 Problems of Interest

Let M be a node selecting automaton (i.e., a k-NFSA or k-NFSTA), d be the input for M (i.e., a word or a binary tree), and $M(d)$ be the result of M on d. We are interested in efficiently maintaining $M(d)$ under updates on d. More precisely, if an update on d occurs then it yields a new data structure d' and a new query result $M(d')$. We want to be able to efficiently compute $M(d')$ after the update. In this case the costs for computing $M(d')$ should be less than computing $M(d')$ from scratch. We consider the following *updates* on trees (cfr. [BPV04]):

(i) replace the current label of a specified node by another label,

(ii) insert a new leaf node after a specified node,

(iii) insert a new leaf node as first child of a specified node, and

(iv) delete a specified leaf node.

For words we consider the updates (i), (ii), and (iv) where the word "leaf" is omitted. If M is only an NTA or NFA (i.e., a 0-ary NFSA or NFSTA) then this problem is known as *incremental evaluation* and was studied by, e.g., Balmin et al. [BPV04].

Problem (IncrementalEvaluation).
Given: An NTA or NFA M.
Compute whether $M(d) \neq \emptyset$.
If d is updated to d' then efficiently compute whether $M(d') \neq \emptyset$.

Here, we examine the *incremental enumeration* problem that extends the setting of Balmin et al. from 0-ary queries to k-ary queries.

Problem (IncrementalEnumeration).
Given: A k-ary NFSA or NFSTA M.
Compute an auxiliary data structure A and enumerate $M(d)$.
If d is updated to d' then update A and enumerate $M(d')$.

We measure the complexity of our algorithms in terms of the following parameters: *(i)* size of the auxiliary data structure, *(ii)* time needed to compute the auxiliary data structure, *(iii)* time needed to update the auxiliary data structure, and *(iv)* time delay we can guarantee between answers of $M(d')$. We present our complexity results in terms of the size $|d|$ of the input data, the arity k of the query, the number $|S|$ of selecting tuples, and the number $|Q|$ of states of the automaton.

Before we start investigating the IncrementalEnumeration problem, we remark the following. Through Sections 6.3 to 6.4 we only illustrate how to handle updates of the kind (i), i.e., *relabeling updates*. Basically, this is sufficient because it is argued in Balmin et al. [BPV04] that one can use self-balancing auxiliary tree structures to generalize the techniques for updates (i) to updates of the kind (ii)–(iv). Moreover, we present all results for binary trees. The results can be immediately generalized to unranked trees in which nodes can have arbitrarily many children [FGK03] which is done by encoding unranked trees as binary trees [BPV04]. Unranked trees are particularly relevant in the context of XML where they serve as an abstract model for XML documents.

6.3 Incremental Enumeration for Words

Now, we examine IncrementalEnumeration for a k-NSFA and a word. We present a well-known algorithm for solving incremental evaluation for NFAs first. For simplicity of notation, we assume that $\text{Nodes}(w) = \{1, \ldots, n\}$ and that n is a power of 2 (i.e., $n = 2^m$ for some $m \in \mathbb{N}$). However, the illustrated algorithm can be adapted to words whose lengths are not a power of two.

6.3.1 An Algorithm for Incremental Evaluation

The following algorithm, first described by Patnaik and Immerman [PI97], solves IncrementalEvaluation for an NFA $N = (Q, \Sigma, \delta, q_0, F)$ and a word $w = a_1 \cdots a_n \in \Sigma^*$. During preprocessing the algorithm builds the following auxiliary data structure.

6.3.1 Definition. For a word w with $\text{Nodes}(w) = \{1, \ldots, n\}$ the *auxiliary tree* N_w^{aux} is defined as follows:

- the root of N_w^{aux} is v_{1n},
- each node v_{xy} with $y - x > 0$ in N_w^{aux} has a left child v_{xz} and a right child $v_{(z+1)y}$ where $z = x - 1 + \lfloor \frac{y-x+1}{2} \rfloor$, and
- the nodes v_{xx} are leaves for all $1 \leq x \leq n$.

We identify the nodes $x \in \{1, \ldots, n\}$ of w with the leaves v_{xx} in N_w^{aux}, i.e., the nodes of w are leaves in N_w^{aux}. Moreover, every node v_{xy} in N_w^{aux} is associated to the subword $w[x..y]$. We know compute the information about how N's state can change when reading $w[x..y]$ and store this information at the associated node v_{xy}.

6.3.2 Definition. For every node v_{xy} in N_w^{aux} the *extended transition relation* $\text{T}(v_{xy})$ is defined as follows:

- if $x = y$ then $\text{T}(v_{xx}) := \{(q_1, q_2) \mid q_2 \in \delta(q_1, a_x)\}$,
- otherwise, the node v_{xy} has a left child v_{xz}, a right child $v_{(z+1)y}$, and we define $\text{T}(v_{xy}) := \{(q_1, q_2) \mid \exists q \in Q \text{ such that } (q_1, q) \in \text{T}(v_{xz}) \text{ and } (q, q_2) \in \text{T}(v_{(z+1)y})\}$.

By the definition of the relation T it holds that $(q_1, q_2) \in \text{T}(v_{xy})$ if and only if $q_2 \in \delta^*(q_1, w[x..y])$. More precisely, $(q_1, q_2) \in \text{T}(v_{xy})$ if and only if reading $w[x..y]$ can bring N from q_1 to q_2. We can compute $\text{T}(v_{xy})$ from $\text{T}(v_{xz})$ and $\text{T}(v_{(z+1)y})$ in time $O(|Q|^3)$ using the natural join (see, e.g., Definition 2.3.1). Since N_w^{aux} has $2n-1$ nodes and $O(\log n)$ depth, N_w^{aux} and T can be computed in time $O(|Q|^3 \cdot n)$. Finally, it holds that $w \in L(N)$ if and only if $(q_0, q_F) \in \text{T}(v_{1n})$ for some $q_F \in F$.

Updates can be maintained as follows using the auxiliary tree N_w^{aux} and the extended transition relation T. Assume that we change label a_x to b in the word w, i.e., the new word is $w = a_1 \cdots a_{x-1} b a_{x+1} \cdots a_n$. The relations T that are affected by the update are those lying on the path from the leaf v_{xx} to the root v_{1n}. Since N_w^{aux} has $O(\log n)$ depth, these are $O(\log n)$ many. Each of these relations can be updated in time $O(|Q|^3)$ in a bottom-up pass through N_w^{aux} using the natural join, yielding a total time of $O(|Q|^3 \cdot \log n)$ for one update.

6.3.3 Theorem ([BPV04, PI97]). IncrementalEvaluation *for an NFA N and a word w can be solved with preprocessing time $O(|Q|^3 \cdot n)$, an auxiliary data structure of size $O(|Q|^2 \cdot n)$, and within time $O(|Q|^3 \cdot \log n)$ after each new update.*

6.3.2 Preprocessing an Auxiliary Data Structure for Words

In the following we construct an algorithm that solves IncrementalEnumeration for a k-NSFA M and a word w. We start by explaining the preprocessing and fix the following notation for the remainder of the section. By $M = ((Q, \Sigma, \delta, q_0, F), S)$ we denote a k-NSFA and by $w = a_1 \cdots a_n \in \Sigma^*$ the input word. By Q_S we denote the set of all states that appear in some selecting tuple, i.e., $Q_S = \cup_{s \in S} \text{set}(s)$. (Remember that we defined $\text{set}(s)$ as the set of symbols occurring in s.) Next, we define the

auxiliary data structure that is stored in the enumeration algorithm during updates. The auxiliary data structure can be constructed in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot n)$ and, whenever w receives an update, it can be updated in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log n)$ and recommence a logarithmic-delay enumeration of the new answer set.

In the beginning, the algorithm builds the auxiliary tree $\mathrm{N}_w^{\mathrm{aux}}$ from Section 6.3.1. For every node in $\mathrm{N}_w^{\mathrm{aux}}$ it stores an extended transition relation that is slightly different from the relation T from Definition 6.3.2. In addition to the pair of states in a tuple of T, the relation contains the set of selecting states that can be reached by a run on the subword associated to the node. We denote this relation by T^+.

6.3.4 Definition. For each $v_{xy} \in \mathrm{N}_w^{\mathrm{aux}}$ we define $\mathrm{T}^+(v_{xy})$ to be the set of tuples $(q_1, q_2, I) \in (Q^2 \times 2^{Q_S})$ for which there exist a selecting tuple $s \in S$ and partial run $r = q_1 \cdots q_2$ on $w[x..y]$ such that $I = \mathrm{set}(r) \cap \mathrm{set}(s)$.

Since the set I in a tuple $(q_1, q_2, I) \in \mathrm{T}^+(v_{xy})$ is a subset of a $\mathrm{set}(s)$ for some selecting tuple $s \in S$, there exist at most 2^k candidate sets I for a fixed selecting tuple s and states q_1 and q_2. Therefore, the relation $\mathrm{T}^+(v_{xy})$ has size $O(|Q^2| \cdot |S| \cdot 2^k)$ for every node $v_{xy} \in \mathrm{N}_w^{\mathrm{aux}}$. To efficiently compute the relation T^+ we use the following join operation.

6.3.5 Definition. Let R_1 and R_2 be relations over $(Q^2 \times 2^{Q_S})$. Then,

$$R_1 \bowtie^+ R_2 := \{(q_1, q_2, I) \mid \exists p \in Q, \exists I_1, I_2 \subseteq Q_S, \exists s \in S \\ \text{such that } (q_1, p, I_1) \in R_1, (p, q_2, I_2) \in R_2, \text{ and } I = (I_1 \cup I_2) \cap \mathrm{set}(s)\}.$$

6.3.6 Lemma. *The relation* T^+ *for* N_w^{aux} *can be computed in time* $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot n)$.

Proof. We first show how the relation T^+ is computed. Afterwards, we prove that it is correct. Let $\mathrm{N}_w^{\mathrm{aux}}$ be the auxiliary tree from Definition 6.3.1. Then, the relation T^+ for $\mathrm{N}_w^{\mathrm{aux}}$ can be computed bottom-up in $\mathrm{N}_w^{\mathrm{aux}}$ as follows:

- If $x = y$ then $\mathrm{T}^+(v_{xx}) = \{(q_1, q_2, I) \mid q_2 \in \delta(q_1, a_x) \text{ and } I = \{q_2\} \cap Q_S\}$. (The condition on I states that $I = \{q_2\}$ if q_2 appears in some selecting tuple s; and $I = \emptyset$ otherwise.)

- Otherwise, let v_1 and v_2 be the left and right child of v_{xy} in $\mathrm{N}_w^{\mathrm{aux}}$, then $\mathrm{T}^+(v_{xy}) = \mathrm{T}^+(v_1) \bowtie^+ \mathrm{T}^+(v_2)$.

We now prove that this computation is correct in terms of Definition 6.3.4. For a node v of $\mathrm{N}_w^{\mathrm{aux}}$ the depth $\mathrm{d}(v)$ of v is the length of the path from the root of $\mathrm{N}_w^{\mathrm{aux}}$ to v. The proof is by induction on decreasing values of the depth $\mathrm{d}(v_{xy})$ of nodes v_{xy} in the auxiliary tree $\mathrm{N}_w^{\mathrm{aux}}$.

For the base case, a leaf node of $\mathrm{N}_w^{\mathrm{aux}}$, we have that

$$\mathrm{T}^+(v_{xx}) = \{(q_1, q_2, I) \mid q_2 \in \delta(q_1, a_x) \text{ and } I = \{q_2\} \cap Q_S\}.$$

Thus, the partial run $r = q_1 q_2$ on $w[x]$ proves the assumption.

For the induction case, a node v_{xy} with $x \neq y$, we have to show that the assumption holds for $\mathrm{T}^+(v_{xy}) = \mathrm{T}^+(v_{xz}) \bowtie^+ \mathrm{T}^+(v_{(z+1)y})$ where $z = x - 1 + \lfloor \frac{y-x+1}{2} \rfloor$ and the relations $\mathrm{T}^+(v_{xz})$ and $\mathrm{T}^+(v_{(z+1)y})$ are already computed correctly. We show "$\Rightarrow$" and "$\Leftarrow$" separately.

"$\Rightarrow$": Let (q_1, q_2, I) be a tuple in $\mathrm{T}^+(v_{xy})$. Then, we know that there exist tuples $(q_1, p, I_1) \in \mathrm{T}^+(v_{xz})$ and $(p, q_2, I_2) \in \mathrm{T}^+(v_{(z+1)y})$ such that $(I_1 \cup I_2) \cap \mathrm{set}(s) = I$ for some $s \in S$. By induction hypothesis we know that there exist

- a partial run $r_1 = q_1 \cdots p$ on $w[x..z]$ such that $I_1 = \mathrm{set}(r_1) \cap \mathrm{set}(s)$, and
- a partial run $r_2 = p \cdots q_2$ on $w[z+1..y]$ such that $I_2 = \mathrm{set}(r_2) \cap \mathrm{set}(s)$.

Since $(I_1 \cup I_2) \cap \mathrm{set}(s) = I$ we can join r_1 and r_2 to a partial run $r = q_1 \cdots q_2$ on $w[x..y]$ such that $I = \mathrm{set}(r) \cap \mathrm{set}(s)$.

"$\Leftarrow$": Let s be a selecting tuple and $r = q_1 \cdots q_2$ be a partial run on $w[x..y]$ such that $I = \mathrm{set}(r) \cap \mathrm{set}(s)$. Then, we can decompose r in two partial runs r_1 and r_2 such that

- $r_1 = q_1 \cdots p$ on $w[x..z]$ such that $I_1 = \mathrm{set}(r_1) \cap \mathrm{set}(s)$,
- $r_2 = p \cdots q_2$ on $w[z+1..y]$ such that $I_2 = \mathrm{set}(r_2) \cap \mathrm{set}(s)$, and
- $(I_1 \cup I_2) \cap \mathrm{set}(s) = I$.

By applying the induction hypothesis on the nodes v_{xz} and $v_{(z+1)y}$ there exist tuples $(q_1, p, I_1) \in \mathrm{T}^+(v_{xz})$ and $(p, q_2, I_2) \in \mathrm{T}^+(v_{(z+1)y})$. Therefore there exists a tuple $(q_1, q_2, I) \in \mathrm{T}^+(v_{xy})$ by definition of $\bowtie^+$.

The relation T^+ can be computed by one bottom-up pass through $\mathrm{N}_w^{\mathrm{aux}}$. For leaf nodes, T^+ can be calculated in time and space $O(|Q|^2)$. For all other nodes the relation T^+ is of size $O(|Q|^2 \cdot |S| \cdot 2^k)$. Since the tree $\mathrm{N}_w^{\mathrm{aux}}$ has $O(n)$ nodes by construction, we need to compute n $\bowtie^+$-joins where each join can be computed in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k)$. Thus, the computation needs $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot n)$ in total. □

The relation T^+ can be maintained under updates analogously to the relation T in Section 6.3.1 except that some extra time is needed for the $\bowtie^+$-operation.

6.3.7 Lemma. *For a k-NSFA M and a word w of length n the tree N_w^{aux} and T^+ have size $O(|Q|^2 \cdot |S| \cdot 2^k \cdot n)$, can be computed in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot n)$, and updated in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log n)$.*

Proof. The tree $\mathrm{N}_w^{\mathrm{aux}}$ has $O(n)$ nodes by construction. For every node $v \in \mathrm{N}_w^{\mathrm{aux}}$ the transition relation $\mathrm{T}^+(v)$ is of size $O(|Q|^2 \cdot |S| \cdot 2^k)$. Therefore, the auxiliary data structure has size $O(|Q|^2 \cdot |S| \cdot 2^k \cdot n)$ in total. By Lemma 6.3.6 it can be built in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot n)$.

Now, assume that an update occurs at node v. The algorithm updates every relation on the path from v to the root of $\mathrm{N}_w^{\mathrm{aux}}$ (i.e., $\log n$ many relations). This can be done by $\log n$ $\bowtie^+$-operations yielding a total time of $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log n)$ for one update. Analogously to the proof of Lemma 6.3.6 it follows that the relation T^+ is correctly computed. □

This concludes the description of the preprocessing and the construction of the auxiliary data structure $\mathrm{N}_w^{\mathrm{aux}}$.

Algorithm 2 Enumeration of $M(w)$

```
1: Enum(M, w) {
2: Input: k-NSFA M = ((Q, Σ, δ, F), S), word w
3: Output: Enumeration of all answers in M(w)
4: A = Complete({∅})
5: while A ≠ ∅ do
6:     output(A)
7:     A = Next(A)
8: }
9: Next(A) {
10: Input: set A of annotated answers
11: Output: set of smallest annotated answers larger than A
12: while Nextnode(A) = ∅ do
13:     A ← Back(A)
14:     if A = ∅ then return ∅
15: return Complete(Nextnode(A))
16: }
```

6.3.3 Enumerating Query Answers for Words

We now discuss how to enumerate query answers. Therefore, we assume that $\mathrm{N}_w^{\mathrm{aux}}$ and T^+ are already computed. A high-level description of the enumeration algorithm is outlined in Algorithm 2. This procedure is similar to enumerating words in a dictionary in lexicographic order but the details are rather different. The strategy of the algorithm is as follows. The procedure **Enum** takes a k-NSFA M and a word w as input and invokes the procedure "Complete" to compute the first set of answers $\mathcal{A}$. (There can be several smallest answers.) Afterwards, the algorithm starts the enumeration by repeatedly calling **Next** (which allows to go from one set of answers to the next) until all answers are depleted.

In the following we describe the operations that are used in Algorithm 2 in detail. First, we define some preliminary notions. We start with a definition of the *output ordering* $\preceq$ in which the algorithm outputs the answers to the query.

6.3.8 Definition. For a tuple $t = (i_1, \ldots, i_k) \in \mathbb{N}^k$, the word $\mathrm{sort}(t)$ is defined as $i_{\sigma(1)} \cdots i_{\sigma(k)}$ where σ is a permutation on $\{1, \ldots, k\}$ such that $i_{\sigma(j)} \leq i_{\sigma(j+1)}$ for every $j \in \{1, \ldots, k-1\}$. The total order $\preceq$ between integer tuples t is defined as the lexicographical order on $\mathrm{sort}(t)$ (taking the empty word to be the lexicographically smallest word).

In other words, $\mathrm{sort}(t)$ denotes the word obtained by sorting $i_1, \ldots, i_k$ in increasing order and concatenating the result. For example, it holds that $\mathrm{sort}((4,1,7)) =$ 1 4 7, $\mathrm{sort}((3,2,9)) = 2\ 3\ 9$, and $\mathrm{sort}((9,2,3)) = 2\ 3\ 9$ such that $(4,1,7) \preceq (3,2,9)$, $(9,2,3) \preceq (3,2,9)$, and $(9,2,3) = (3,2,9)$ with respect to $\preceq$.

Notice that we also use the order $\preceq$ to compare tuples of different size. For example, it holds that $(4,1,7) \preceq (3,2)$ (according to the lexicographic order on

words). By $\prec$ we denote the strict variant of $\preceq$ and we analogously define $\preceq$ (and $\prec$) over multisets instead of tuples.

During a computation the algorithm computes so-called *annotated answers*, which are multisets of pairs in $\mathrm{Nodes}(w) \times Q$. Annotated answers contain, in addition to nodes of w, the states that were responsible for selecting the nodes. If (i, q) occurs j times in the annotated answer then there is a selecting tuple $s \in S$ (with at least j occurrences of q) and node i is associated to j occurrences of q in s for this annotated answer. (Intuitively, this means that the algorithm will eventually produce an answer that has j occurrences of node i.) Annotated answers are formally defined as follows.

6.3.9 Definition. An *annotated answer* of $M = (N, S)$ on w is a multiset A^{full} over $\mathrm{Nodes}(w) \times Q$ of the form

$$\{\!\{(i_1, q_1), \ldots, (i_k, q_k)\}\!\}$$

such that there is an accepting run r of N on w and a $(q_1, \ldots, q_k) \in S$ such that r visits i_ℓ in q_ℓ for every $\ell \in \{1, \ldots, k\}$. An *incomplete (annotated) answer* is a (not necessarily strict) subset A of some annotated answer A^{full}.

Notice that $|A^{\mathrm{full}}| = k$ and that every incomplete answer can be completed into an answer. We sometimes also say that A^{full} is an *annotated answer w.r.t. r* if we want to emphasize the connection between A^{full} and r. For a multiset $A = \{\!\{(i_1, q_1), \ldots, (i_k, q_k)\}\!\}$ over $\mathrm{Nodes}(w) \times Q$ we denote by $\mathrm{Nodes}(A) = \{\!\{i_1, \ldots, i_k\}\!\}$ the multiset of nodes in A.

Moreover, we extend the order $\preceq$ to multisets over $\mathrm{Nodes}(w) \times Q$ to compare (incomplete) annotated answers with respect to the output order. For two such multisets A and B it holds that $A \preceq B$ if $\mathrm{Nodes}(A) \preceq \mathrm{Nodes}(B)$. We extend $\prec$ analogously. For a set $\mathcal{A}$ of (incomplete) annotated answers, the set of minima is defined as

$$\min(\mathcal{A}) = \{A \in \mathcal{A} \mid \forall B \in \mathcal{A} : A \preceq B\}.$$

Next, we define the semantics of the procedures in Algorithm 2.

6.3.10 Definition. Let $\mathcal{A}$ be a set of multisets over $\mathrm{Nodes}(w) \times Q$, then

$$\mathrm{Complete}(\mathcal{A}) := \min\{A^{\mathrm{full}} \mid A^{\mathrm{full}} \text{ is an annotated answer such that } \exists A \in \mathcal{A} : A \subseteq A^{\mathrm{full}} \text{ and } A \preceq A^{\mathrm{full}}\}.$$

Intuitively, $\mathrm{Complete}(\mathcal{A})$ contains the smallest (with respect to $\preceq$) annotated answers of M obtained from extending elements $A \in \mathcal{A}$ on nodes of w that are all larger or equal to the maximal node that is already used in A. (In this way, $\mathrm{Complete}(\{\emptyset\})$ is the set of smallest annotated answers.)

The procedure $\mathbf{Next}(\mathcal{A})$ computes, for a set of annotated answers, the set of immediate successors with respect to the output order. For the definition of $\mathbf{Next}(\mathcal{A})$ in Algorithm 2 we use the following subroutines. Let $A = \{\!\{(i_1, q_1), \ldots, (i_j, q_j)\}\!\}$ be a multiset over $\mathrm{Nodes}(w) \times Q$ where, for each i_ℓ, there is at most one q_ℓ such that $(i_\ell, q_\ell) \in A$. Let i_j be one of the maximal (rightmost) nodes in $\mathrm{Nodes}(A)$. Then, we define $A_{\mathrm{del}} = \{\!\{(i_1, q_1), \ldots, (i_{j-1}, q_{j-1})\}\!\}$.

6.3.11 Definition. Let $\mathcal{A}$ be a set of multisets over $\text{Nodes}(w) \times Q$, then

$$\text{Back}(\mathcal{A}) := \min\{A \mid A \text{ is an incomplete answer such that } \exists A' \in \mathcal{A} : \text{Nodes}(A) = \text{Nodes}(A'_{\text{del}})\}.$$

Intuitively, $\text{Back}(\mathcal{A})$ performs a kind of backtracking step. It returns all smallest incomplete answers that, compared with an element $A = \{\!\{(i_1, q_1), \ldots, (i_j, q_j)\}\!\} \in \mathcal{A}$, annotate exactly the nodes $i_1, \ldots, i_{j-1}$ (if we assume i_j to be maximal).

6.3.12 Definition. Let $\mathcal{A}$ be a set of multisets over $\text{Nodes}(w) \times Q$, then

$$\text{Nextnode}(\mathcal{A}) := \min\{A \mid A \text{ is an incomplete answer such that } \exists A' \in \mathcal{A} : |A| = |A'| \text{ and } A' \prec A \text{ and } A_{\text{del}} \subseteq A'\}.$$

For a set of incomplete answers $\mathcal{A}$ the procedure $\text{Nextnode}(\mathcal{A})$ returns the set of all incomplete answers that are of the same size as the incomplete answers in $\mathcal{A}$ and for which only the maximal node of an answer in $\mathcal{A}$ has been incremented.

6.3.13 Lemma. *Let $\mathcal{A}$ be a set of annotated answers. Then **Next**$(\mathcal{A})$ in Algorithm 2 returns*

$$\min\{A^{full} \mid A^{full} \text{ is an annotated answer such that } \exists A \in \mathcal{A} : A \prec A^{full}\}.$$

Proof. By the definition of Complete (see Definition 6.3.10) we know that $\mathbf{Next}(\mathcal{A})$ always returns a set of minima for a set of annotated answers. Therefore, it remains to show that for every annotated answer A^{full} in $\mathbf{Next}(\mathcal{A})$ there exists an $A \in \mathcal{A}$ such that $A \prec A^{\text{full}}$.

In Algorithm 2 the procedure $\mathbf{Next}(\mathcal{A})$ returns the set $\text{Complete}(\mathcal{B})$ where $\mathcal{B} = \text{Nextnode}(\text{Back}^b(\mathcal{A}))$ and b is the minimal number such that $\text{Nextnode}(\text{Back}^b(\mathcal{A}))$ is not empty. (The superscript ^{b} denotes that we apply the procedure b times on its own output.) Let $A^{\text{full}} = \{\!\{(i_1, q_1), \ldots, (i_k, q_k)\}\!\} \in \mathbf{Next}(\mathcal{A})$. Then, we know that there exists an annotated answer $A = \{\!\{(\ell_1, p_1), \ldots, (\ell_k, p_k)\}\!\} \in \mathcal{A}$ such that

$$\{\!\{(\ell_1, p_1), \ldots, (\ell_{k-b}, p_{k-b})\}\!\} = \{\!\{(i_1, q_1), \ldots, (i_{k-b}, q_{k-b})\}\!\} \in \text{Back}^b(\mathcal{A}).$$

By the definition of Nextnode (see Definition 6.3.12) we get that $\ell_{k-b+1} < i_{k-b+1}$. It follows that, for every $j > k - b + 1$, node i_j is larger or equal to the node i_{k-b+1} by the definition of Complete. Therefore, $A \prec A^{\text{full}}$ which concludes the proof. □

The procedure $\mathit{output}(\mathcal{A})$ gets a set of annotated answers $\mathcal{A}$ as input and writes the following set to the output in arbitrary order:

$$\{(i_1, \ldots, i_k) \mid \{\!\{(i_1, q_1), \ldots, (i_k, q_k)\}\!\} \in \mathcal{A} \text{ and } (q_1, \ldots, q_k) \in S\}.$$

Notice that this set can contain multiple tuples which are all equal with respect to $\preceq$. (For example, one tuple could be $(1, 2, 2, 3, 4)$ and another could be $(2, 4, 3, 2, 1)$.) The output procedure can be designed such that the delay between these tuples in the output is constant.

The proof of the next lemma, which shows that the the procedure **Enum(**M, w**)** works correctly, relies on the following observation about the sets $\mathcal{A}$ of annotated answers that are computed in Algorithm 2.

6.3.14 Observation. All $\mathcal{A}$ in Algorithm 2 are such that, for all $A, B \in \mathcal{A}$, we have $\text{Nodes}(A) = \text{Nodes}(B)$.

The observation is true because all operations in Algorithm 2 return a set of minima of incomplete annotated answers.

6.3.15 Lemma. ***Enum(***M, w***)*** *correctly enumerates all answers in* $M(w)$*.*

Proof. By the definition of annotated answers we know that there exists an annotated answer for every answer in $M(w)$. Therefore, it remains to show that the procedure **Enum(**M, w**)** computes all annotated answers for M and w and does not output an answer twice.

Let $\mathcal{A}_1, \ldots, \mathcal{A}_m$ be the sequence of sets of annotated answers that are given to $\mathit{output}(\mathcal{A})$ during a run of the algorithm **Enum(**M, w**)**. We know that $\mathcal{A}_1 = \text{Complete}(\{\emptyset\})$. By the definition of Complete, $\mathcal{A}_1$ contains the set of smallest annotated answers to $M(w)$.

Next, we show that the set $\mathcal{A}_i$ with $i > 1$ is the set of all minimal annotated answers that are strictly larger than the answers in $\mathcal{A}_{i-1}$. To this end, observe that every set $\mathcal{A}_i$ with $i \in \{1, \ldots, m\}$ is a set $\text{Complete}(\mathcal{B})$ for some set of annotated answers $\mathcal{B}$. By Observation 6.3.14 it holds that $\text{Nodes}(A) = \text{Nodes}(B)$ for all $A, B \in \mathcal{A}_i$. By Lemma 6.3.31 we know that the answers in $\mathcal{A}_i$ are minimal and larger (with respect to $\preceq$) which means there is no annotated answer $C \in M(w)$ such that $C \notin \mathcal{A}_i$ for all $i \in \{1, \ldots, m\}$. Since the procedure $\mathit{output}(\mathcal{A})$ deletes duplicate answers in the set $\mathcal{A}_i$ the algorithm never outputs an answer twice. □

In the following we explain how $\text{Complete}(\mathcal{A})$, $\text{Back}(\mathcal{A})$, and $\text{Nextnode}(\mathcal{A})$ are implemented in the algorithm.

6.3.4 Computing the First Answer to a Query

To compute $\text{Complete}(\{\emptyset\})$, the set of smallest answer(s) in $M(w)$ with respect to the output order, the algorithm first needs to find the leftmost piece of information in N_w^{aux} that is relevant to some answer. This piece of information is stored in a set of so-called *growing (annotated) answers*, which contain the full information of some answers in $M(w)$ up to a node j. These growing answers evolve into the first set of answers for $M(w)$ during a computation of the algorithm. Therefore, the algorithm navigates further to the right to search subsequently for the leftmost nodes in w that can be used to add more and more information to the growing answers until at least one growing answer is *complete*. In the end, a growing answer contains the full information of some answer in $M(w)$. Next, we formally define *growing (annotated) answers*. For a multiset A over $\text{Nodes}(w) \times Q$ we denote by $A((i, p))$ the number of occurrences of the tuple (i, p) in A.

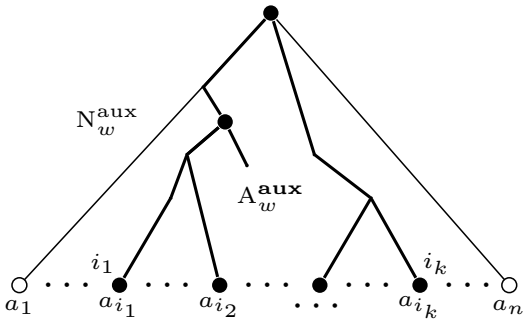


Figure 6.2: $\mathrm{A}_w^{\mathrm{aux}}$ is the part of $\mathrm{N}_w^{\mathrm{aux}}$ that is used to compute the first answers.

6.3.16 Definition. Let $q \in Q$ be a state in the NFA N, A be a multiset over $\mathrm{Nodes}(w) \times Q$, and $j \in \{1, \ldots, n\}$. Then, (q, A) is a *growing annotated answer up to node j* if there is an accepting run r of N on w such that

– r visits j in q, and

– there is an annotated answer A^{full} w.r.t. r such that, for every $p \in Q$ and $i \in \{1, \ldots, n\}$,

- if $i < j$ then $A^{\mathrm{full}}((i,p)) = A((i,p))$,
- if $i = j$ then $A((i,p)) \leq A^{\mathrm{full}}((i,p))$, and
- if $i > j$ then $A((i,p)) = 0$.

The second bullet point in the above definition states that A has the same information as A^{full} concerning the nodes up to j and possibly partial information about j itself. For brevity, we often refer to (q, A) as *growing answer*.

Growing answers can be computed as follows. Assume that $(i_1, \ldots, i_k)$ (see Figure 6.2) is a smallest answer in $M(w)$ with respect to the output order. (Notice that we abbreviate $v_{i_j i_j}$ with i_j and that some of the i_j can be equal.) Let $\mathrm{A}_w^{\mathrm{aux}}$ be the tree induced by all ancestors of nodes i_j in the auxiliary tree $\mathrm{N}_w^{\mathrm{aux}}$ (from Definition 6.3.1). Hence, the tree $\mathrm{A}_w^{\mathrm{aux}}$ has at most k leaves, its root is the root of $\mathrm{N}_w^{\mathrm{aux}}$, and each of its leaves corresponds to a node i_j. To obtain $(i_1, \ldots, i_k)$ we perform a depth-first left-to-right traversal of $\mathrm{A}_w^{\mathrm{aux}}$. Since the depth of $\mathrm{N}_w^{\mathrm{aux}}$ is logarithmic in n such a traversal costs at most $O(k \log n)$ steps. In particular, one can travel from one leaf in $\mathrm{A}_w^{\mathrm{aux}}$ to the next within $O(\log n)$ steps. Our goal is to show that we can compute this traversal when the right kind of information is stored along the paths of $\mathrm{A}_w^{\mathrm{aux}}$.

We first explain how to compute and traverse the leftmost path of $\mathrm{A}_w^{\mathrm{aux}}$. Therefore, we start at the root of $\mathrm{N}_w^{\mathrm{aux}}$ and need to decide which child to choose. To this end, we compute *relevant tuples*, which store for every node in $\mathrm{N}_w^{\mathrm{aux}}$ the set of (partial) runs that can be used for a run that produces some answer in $M(w)$.

6.3.17 Definition. For each $v \in \mathrm{N}_w^{\mathrm{aux}}$ the set of *relevant tuples of* v, denoted $R(v)$, is inductively defined as follows:

- $R(v_{1n}) = \{(q_0, q_F, \mathrm{set}(s)) \in \mathrm{T}^+(v_{1n}) \mid q_F \in F, s \in S\}$,
- otherwise, if $(q_1, q_2, I) \in R(v)$ and v_1 and v_2 are left and right child of v then we want to "split" I between v_1 and v_2. More precisely, let

$$R_{v_1,v_2} = \{(q_1, q_2, J_1, q_2, q_3, J_2) \mid \exists (q_1, q_3, I) \in R(v), (q_1, q_2, I_1) \in \mathrm{T}^+(v_1), (q_2, q_3, I_2) \in \mathrm{T}^+(v_2) \text{ such that } J_1 \cup J_2 = I, J_1 \subseteq I_1, \text{ and } J_2 \subseteq I_2\}.$$

  Then, $R(v_1) = \{(q_1, q_2, J_1) \mid (q_1, q_2, J_1, q_2, q_3, J_2) \in R_{v_1,v_2}\}$ and $R(v_2) = \{(q_2, q_3, J_2) \mid (q_1, q_2, J_1, q_2, q_3, J_2) \in R_{v_1,v_2}\}$.

For the root node v_{1n} of $\mathrm{N}_w^{\mathrm{aux}}$ a tuple is relevant when it indicates that there is a run from the initial state q_0 to an accepting state q_F on which every state in a selecting tuple s can be visited. Thus, if there exists such a relevant tuple for v_{1n} then $M(w)$ contains at least one answer by the definition of T^+ (see Definition 6.3.4). Moreover, if $(q_1, q_2, I) \in R(v)$ then we split I among the children v_1 and v_2 of v in all possible ways. This is to ensure that, if we find a partial result for v_1 then we are *certain* that we can find sufficient information below v_2 to annotate *every state in* I. Therefore, the sets R contain exactly the information from T^+ that is relevant for producing answers, i.e., tuples in R can be associated to accepting runs that produce at least one answer. This is captured in the following lemma.

For an annotated answer $A^{\mathrm{full}} = \{\!\{(i_1, q_1), \ldots, (i_k, q_k)\}\!\}$ and nodes x, y in w the *projection of* A^{full} *onto* $[x, y]$, denoted $A^{\mathrm{full}}_{[x,y]}$, is defined as the multiset $\{\!\{(i, q_i) \mid x \leq i \leq y\}\!\}$. For a run r we denote by $r(x)$ the state q at the xth position in r.

6.3.18 Lemma. *For every node* $v_{xy} \in \mathrm{N}_w^{aux}$ *the relation* $R(v_{xy})$ *is the set of all tuples* (q_1, q_2, I) *such that there is an annotated answer* A^{full} *w.r.t. some run* r *with* $r(x) \in \delta(q_1, w[x])$, $r(y) = q_2$, *and* $I = Nodes(A^{full}_{[x,y]})$.

Proof. The proof is by induction on increasing values of the depth $\mathrm{d}(v_{xy})$ in the auxiliary tree $\mathrm{N}_w^{\mathrm{aux}}$. (The depth $\mathrm{d}(v_{xy})$ of a node in $\mathrm{N}_w^{\mathrm{aux}}$ is defined as the length of the path from the root to v_{xy}.)

For the base case, the root $v_{1n} \in \mathrm{N}_w^{\mathrm{aux}}$, we have that

$$R(v_{1n}) = \{(q_0, q_F, \mathrm{set}(s)) \in \mathrm{T}^+(v_{1n}) \mid q_F \in F, s \in S\}.$$

The assumption holds by the definition of relation T^+.

For the induction case we have to show that the assumption holds for a left-child-node and for a right-child-node v_{xy}. Since both cases are symmetric we only show the case where v_{xy} is a left child in the following. Let v_{xy} be a left child of a node v_{xz} and let $v_{(y+1)z}$ be the right child of v_{xz}. Recall that, by Definition 6.3.17,

$$R(v_{xy}) = \cup_{(q_1,q_3,I) \in R(v_{xz})} \{(q_1, q_2, J_1) \mid \exists (q_1, q_2, I_1) \in \mathrm{T}^+(v_{xy}) \text{ such that } J_1 \subseteq I_1, \exists (q_2, q_3, I_2) \in \mathrm{T}^+(v_{(y+1)z}) \text{ such that } J_2 \subseteq I_2, \text{ and } J_1 \cup J_2 = I\}.$$

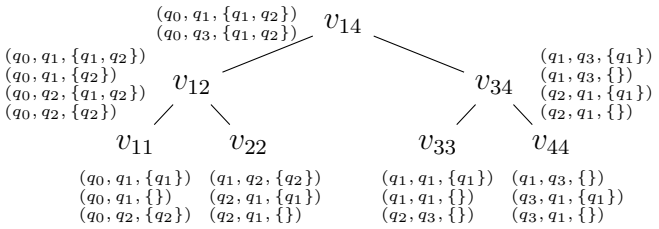


Figure 6.3: The relation R for the 2-NSFA M from Figure 6.1 and $w = abcd$.

We show the assumption by proving "$\Rightarrow$" and "$\Leftarrow$" separately.

"$\Rightarrow$": Let (q_1, q_2, J_1) be a tuple in $R(v_{xy})$. Then there are, according to Definition 6.3.17, tuples $(q_1, q_3, I) \in R(v_{xz})$, $(q_1, q_2, I_1) \in \mathrm{T}^+(v_{xy})$ and $(q_2, q_3, I_2) \in \mathrm{T}^+(v_{(y+1)z})$ such that $J_1 \subseteq I_1$ and there exists a set $J_2 \subseteq I_2$ where $J_1 \cup J_2 = I$. By the definition of T^+ we know that there exist

- a partial run $r_1 = q_1 \cdots q_2$ on $w[x..y]$ such that $J_1 \subseteq \mathrm{set}(r_1)$, and
- a partial run $r_2 = q_2 \cdots q_3$ on $w[y+1..z]$ such that $J_2 \subseteq \mathrm{set}(r_2)$.

By applying the induction hypothesis on the parent node v_{xz} we know that for the tuple $(q_1, q_3, I) \in R(v_{xz})$ there is an annotated answer A w.r.t. some run r such that $r(x) \in \delta(q_1, w[x])$, $r(z) = q_3$, and $I = \mathrm{Nodes}(A_{[x+1,z]})$. Thus, r is a run $r_\ell r_m r_r$ such that

- r_ℓ is a partial run $q_0 \cdots q_1$ on $w[1..x-1]$, and
- r_r is a partial run $q_3 \cdots q_F$ on $w[z+1..n]$.

Observe that $\mathrm{set}(r_\ell) \cup \mathrm{set}(r_r) \cup I = \mathrm{set}(s)$ for some $s \in S$. In total, the run $r_\ell r_1 r_2 r_r$ is an accepting run that produces an annotated answer A^{full}, which proves the assumption for v_{xy}.

"$\Leftarrow$": Towards a contradiction, assume that $(q_1, q_2, J_1) \notin R(v_{xy})$ and there is an annotated answer A^{full} w.r.t. some run r with $r(x) \in \delta(q_1, w[x])$, $r(y) = q_2$, and $J_1 = \mathrm{Nodes}(A^{\mathrm{full}}_{[x+1,y]})$. By applying the induction hypothesis on the parent node v_{xz} we know that there exists a tuple $(q_1, q_3, I) \in R(v_{xz})$ that is compatible with A^{full} and r. By the definition of T^+ it follows that there are tuples $(q_1, q_2, I_1) \in \mathrm{T}^+(v_{xy})$ with $J_1 \subseteq I_1$ and $(q_2, q_3, I_2) \in \mathrm{T}^+(v_{(y+1)z})$ such that there is a set $J_2 \subseteq I_2$ and $J_1 \cup J_2 = I$. This contradicts the assumption that $(q_1, q_2, J_1) \notin R(v_{xy})$. □

For the 2-NSFA in Figure 6.1 the relation R is shown in Figure 6.3. The relation R can be computed by straightforwardly implementing its definition in a top-down way.

6.3.19 Lemma. *Given* N_w^{aux} *and* T^+*, we can compute* $R(v_{1n})$ *in time* $O(|Q|^2 \cdot |S| \cdot 2^k)$ *and, for every other* $v \in \mathrm{N}_w^{aux}$ *with parent* v_p*, compute* $R(v)$ *in time* $O(|Q|^3 \cdot |S|^2 \cdot 2^k)$ *if* $R(v_p)$ *is known.*

Proof. For the root node v_{1n} we have that

$$R(v_{1n}) = \{(q_0, q_F, \mathrm{set}(s)) \in \mathrm{T}^+(v_{1n}) \mid q_F \in F, s \in S\}.$$

Thus, we only need to traverse the relation $\mathrm{T}^+(v_{1n})$ which is of size $O(|Q|^2 \cdot |S| \cdot 2^k)$. For all other nodes we have to distinguish whether they are a left or a right child of its parent v_p. Let v_1 be the left and v_2 be the right child of v_p. To calculate $R(v_1)$ and $R(v_2)$ we traverse the relation $R(v_p)$. We check for every tuple $(q_1, q_3, I) \in R(v_p)$ whether there exist tuples $(q_1, q_2, I_1) \in \mathrm{T}^+(v_1)$ and $(q_2, q_3, I_2) \in \mathrm{T}^+(v_2)$ such that there are subsets $J_1 \subseteq I_1$ and $J_2 \subseteq I_2$ with $J_1 \cup J_2 = I$. If J_1 and J_2 exist then we add a tuple (q_1, q_2, J_1) to $R(v_1)$ for every such set J_1 and a tuple (q_2, q_3, J_2) to $R(v_2)$ for every such set J_2. Since the relations R and T^+ contain at most $O(|Q|^2 \cdot |S| \cdot 2^k)$ different tuples for every node, it needs time $O(|Q|^3 \cdot |S|^2 \cdot 2^k)$ to calculate $R(v_1)$ and $R(v_2)$. □

We state Lemma 6.3.19 as it is because our algorithm for IncrementalEnumeration does not compute $R(v)$ for every node v of $\mathrm{N}_w^{\mathrm{aux}}$ but only among the paths of $\mathrm{A}_w^{\mathrm{aux}}$.

The First Part of the First Answer In order to find the leftmost path of $\mathrm{A}_w^{\mathrm{aux}}$ we start at the root of $\mathrm{N}_w^{\mathrm{aux}}$ and iteratively perform the following. Whenever we are in a node v, we compute the sets of relevant tuples of its two children. We proceed to the leftmost child for which the set of relevant tuples contains a tuple (q_1, q_2, I) with $I \neq \emptyset$ and stop when we reach a leaf. We claim that this leaf is the leftmost node i_1 in $\mathrm{N}_w^{\mathrm{aux}}$ that can be used in some smallest answer of $M(w)$ (see Figure 6.2). Notice that we only know that i_1 is used in such a smallest answer but not necessarily as the leftmost element of a tuple in $M(w)$. (For example, $M(w)$ can contain tuples of the form $(i_2, i_1, \ldots)$ with $i_2 > i_1$.)

6.3.20 Lemma. *Let* u *be the leftmost leaf node of* N_w^{aux} *such that* $R(u)$ *has a tuple* (q_1, q_2, I) *with* $I \neq \emptyset$*. Then* u *is the node* i_1 *in* w*.*

Proof. Let u be as desired in the lemma. Towards a contradiction, assume that $i_1 \neq u$, i.e., there exists a leaf node $v < u$ such that there is an annotated answer A^{full} with $v \in \mathrm{Nodes}(A^{\mathrm{full}})$. By Lemma 6.3.18 there exists a tuple $(p, q, J) \in R(v)$ with $J \neq \emptyset$. This contradicts the assumption. □

This allows us to define our first set G of growing answers, that is

$$G(i_1) := \{(q_2, \{\!\{(i_1, q_2)\}\!\}) \mid (q_1, q_2, \{q_2\}) \in R(i_1)\}.$$

By Lemmas 6.3.18 and 6.3.20 every element in $G(i_1)$ is a growing answer up to node i_1. We can compute i_1 and the set $G(i_1)$ by traversing the path from the root of $\mathrm{N}_w^{\mathrm{aux}}$ to i_1.

6.3.21 Lemma. *The node i_1 and the set $G(i_1)$ can be computed in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log n)$.*

Proof. The node i_1 can be computed by one top-down pass of the path from the root of $\mathrm{N}_w^{\mathrm{aux}}$ to i_1. Therefore, we start at the root and, whenever we are in a node v, we compute the sets of relevant tuples of its two children v_1 and v_2 (if these exist, otherwise we are done). We proceed to the leftmost child for which the set of relevant tuples contains a tuple (p, q, I) with $I \neq \emptyset$. We are done when we reach a leaf. At every node on this path we have to calculate two sets of relevant tuples. By Lemma 6.3.19 this can be done in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k)$ for each node. Since a path from the root to a leaf in $\mathrm{N}_w^{\mathrm{aux}}$ is of length $\log n$ this needs time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log n)$ in total. □

For the running example in Figure 6.1 we have that

$$G(1) = \{(q_1, \{\!\{(1, q_1)\}\!\}), (q_2, \{\!\{(1, q_2)\}\!\})\}.$$

Growing Until the First Answer is Complete We assume that from now on we know some $j \in \{1, \ldots, k\}$ such that the set $G(i_j)$ is defined and not empty. In the following we will explain the semantics of a set $G(i_j)$ for $j > 1$ and how to compute the set $G(i_{j+1})$ for the node i_{j+1} when $G(i_j)$ is given. To this end, we first have to find the node i_{j+1} itself (recall that, $i_{j+1} = i_j$ is possible) and, second, all the information that is needed to compute the correct set of growing answers. We therefore navigate from i_j to the right and keep track of the relevant tuples that are *compatible* with our growing answer(s). Intuitively, a relevant tuple is compatible to a growing answer (q, A) if the tuple can be associated to a partial run that is used when extending the incomplete answer A to a complete answer A^{full}. Next, we formally define the notion of compatibility.

The *projection* of a multiset $A = \{\!\{(i_1, q_1), \ldots, (i_k, q_k)\}\!\}$ of tuples over $\mathbb{N} \times Q$ onto Q, denoted $\pi_Q(A)$, is defined as $\{\!\{q_1, \ldots, q_k\}\!\}$.

6.3.22 Definition (Compatibility). Let v_{xy} be a node in $\mathrm{N}_w^{\mathrm{aux}}$. For an annotated answer A^{full} w.r.t. a run r, a tuple

- $(q_1, q_2, I) \in R(v_{xy})$ *is compatible with* A^{full} *and* r if $r(x) \in \delta(q_1, w[x])$, $r(y) = q_2$, and $I = \mathrm{set}(\pi_Q(A^{\mathrm{full}}_{[x,y]}))$. (Here, $I = \mathrm{set}(\pi_Q(A^{\mathrm{full}}_{[x,y]}))$ ensures that I is the set of selecting states in A^{full} that are used between nodes x and y in w.)

Furthermore, for (q, A) a growing answer up to node i,

- (q, A) is *compatible with* A^{full} *and* r if $r(i) = q$, $A_{[1,i-1]} = A^{\mathrm{full}}_{[1,i-1]}$ and $A_{[i,i]} \subseteq A^{\mathrm{full}}_{[i,i]}$.

Finally, $(q_1, q_2, I) \in R(v_{xy})$ *is compatible with* (q, A) if there exists an annotated answer A^{full} w.r.t. some run r such that both (q_1, q_2, I) and (q, A) are compatible with A^{full} and r.

Now, we can define the node i_{j+1} in w in terms of compatibility.

6.3.23 Proposition. *The node $i_{j+1} \geq i_j$ is the smallest node in w for which there is a tuple $(q_1, q_2, I) \in R(i_{j+1})$ with $I \neq \emptyset$ that is compatible with some $(q, A) \in G(i_j)$.*

Once we have i_{j+1}, we define

$$G(i_{j+1}) = \{(q_2, A \cup \{\!\{(i_{j+1}, q_2)\}\!\}) \mid \text{there exists a tuple } (q_1, q_2, \{q_2\}) \in R(i_{j+1}) \text{ that is compatible with some } (q, A) \in G(i_j)\}.$$

In this way, our algorithm will successively compute sets $G(i_j)$ for increasing values of j. The next lemma states that the set $G(i_k)$ contains the desired answer(s).

6.3.24 Lemma. *Let $\mathcal{A}^{first}$ be the set of smallest annotated answers. Then, it holds that $G(i_k) = \{(q, A) \mid A \in \mathcal{A}^{first}$ and (q, A) is compatible with $A\}$.*

Proof. We prove that the lemma is true by showing Claim 6.3.25. The statement of the lemma holds directly by applying the claim for $j = k$.

6.3.25 Claim. *Let $\mathcal{A}^{first}$ be the set of smallest annotated answers. Let $\mathcal{A}^{cand}$ be the set of annotated answers A^c for which there exists an $A^f \in A^{first}$ with*

$$Nodes(A^c_{[1,i_j-1]}) = Nodes(A^f_{[1,i_j-1]}) \text{ and } Nodes(A^c_{[i_j,i_j]}) \subseteq Nodes(A^f_{[i_j,i_j]}).$$

Then, it holds that $G(i_j) = \{(q, A) \mid |A| = j$ and $\exists A^c \in \mathcal{A}^{cand}$ such that (q, A) is compatible with $A^c\}$.

Proof of Claim 6.3.25: The proof is by induction on increasing values j. For the induction base case, $j = 1$, we know that

$$G(i_1) := \{(q, \{\!\{(i_1, q)\}\!\}) \mid (p, q, \{q\}) \in R(i_1)\}.$$

Obviously, we have that $|\{\!\{(i_1, q)\}\!\}| = 1$. By Definition of R and Lemma 6.3.18, there is an annotated answer A^{full} w.r.t. some run r such that $A^{\text{full}}_{[i_1,i_1]} = \{\!\{(i_1, q), \ldots, (i_1, q)\}\!\}$. By Lemma 6.3.20 we know that i_1 is the smallest position that can be assigned in any annotated answer. Moreover, we have that $A^{\text{full}}_{[1,i_1]} = A^{\text{full}}_{[i_1,i_1]}$ which means that $(q, \{\!\{(i_1, q)\}\!\})$ is compatible with A^{full}. It remains to show that $A^{\text{full}} \in \mathcal{A}^{\text{cand}}$. Therefore, let A^f be an annotated answer in $\mathcal{A}^{\text{first}}$. By correctness of i_1 (see Lemma 6.3.20) it holds that

$$\text{Nodes}(A^{\text{full}}_{[1,i_j-1]}) = \emptyset = \text{Nodes}(A^f_{[1,i_j-1]}) \text{ and } \text{Nodes}(A^{\text{full}}_{[i_j,i_j]}) \subseteq \text{Nodes}(A^f_{[i_j,i_j]}).$$

Thus, $A^{\text{full}} \in \mathcal{A}^{\text{cand}}$ which proves the assumption for i_1.

Now, assume that the statement holds for j. Then, we have that

$$G(i_{j+1}) = \{(q, A \cup \{\!\{(i_{j+1}, q)\}\!\}) \mid \text{there exists a tuple } (p, q, \{q\}) \in R(i_{j+1}) \text{ that is compatible with some } (q', A) \in G(i_j)\}.$$

Since the statement holds for j we have that $|A| = j$ and, hence, it follows that $|A \cup \{\!\{(i_{j+1}, q)\}\!\}| = j + 1$. By the definition of compatibility (see Definition 6.3.22)

there is an annotated answer A^{full} w.r.t some run r such that $(p, q, \{q\})$ and (q', A) are compatible with A^{full} and r. Thus, $(q, A \cup \{\!\{(i_{j+1}, q)\}\!\})$ is compatible with A^{full} and r. It remains to show that $A^{\text{full}} \in \mathcal{A}^{\text{cand}}$. By applying the induction hypothesis on $(q', A) \in G(i_j)$ we know that there exists annotated answer $A^f \in \mathcal{A}^{\text{first}}$ with

$$\text{Nodes}(A^{\text{full}}_{[1,i_j-1]}) = \text{Nodes}(A^{f}_{[1,i_j-1]}) \text{ and } \text{Nodes}(A^{\text{full}}_{[i_j,i_j]}) \subseteq \text{Nodes}(A^{f}_{[i_j,i_j]}).$$

By correctness of i_{j+1} (see Lemma 6.3.31) it follows that A^{full} and A^f have the same node i_{j+1}, i.e., it also holds that

$$\text{Nodes}(A^{\text{full}}_{[i_j,i_j]}) = \text{Nodes}(A^{f}_{[i_j,i_j]}) \text{ and } \text{Nodes}(A^{\text{full}}_{[i_{j+1},i_{j+1}]}) \subseteq \text{Nodes}(A^{f}_{[i_{j+1},i_{j+1}]}).$$

In total, we have $A^{\text{full}} \in \mathcal{A}^{\text{cand}}$ which proves the assumption. □

For the running example from Figure 6.1 we have $k = 2$ and

$$G(2) = \{(q_1, \{\!\{(1, q_2), (2, q_1)\}\!\}), (q_2, \{\!\{(1, q_1), (2, q_2)\}\!\})\}.$$

It remains to show how i_{j+1} and $G(i_{j+1})$ can be efficiently computed. Previously, we have seen that we can compute $G(i_1)$ in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log n)$. Next, we prove that we can compute i_{j+1} in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log n)$ when $G(i_j)$ is given. Afterwards, we examine the computation of the set $G(i_{j+1})$.

To this end, we extend the notion of relevant tuples to j-relevant tuples. Intuitively, j-relevant tuples contain all tuples from R that remain relevant for constructing the smallest possible answer, given the knowledge we have at node i_j.

6.3.26 Definition. For $v_{xy} \in \text{N}_w^{\text{aux}}$ and $j \in \{0, \ldots, k\}$, the set of *j-relevant tuples*, denoted $R_j(v_{xy})$, is defined as follows:

- $R_0(v_{xy}) := R(v_{xy})$, and
- for each $j \geq 1$
  - if $y < i_j$ then $R_j(v_{xy}) := R_{j-1}(v_{xy})$,
  - otherwise, $R_j(v_{xy}) := \{(q_1, q_2, I) \in R(v_{xy}) \mid (q_1, q_2, I)$ is compatible with a growing answer $(q, A) \in G(i_j)\}$.

For example, if $i_1 = 1$ then every tuple from Figure 6.3 is in the relation R_1 except the tuple $(q_0, q_1, \{\}) \in R(v_{11})$. Since tuples in R_j are compatible with a growing answer in $G(i_j)$ by definition, we can now reformulate Proposition 6.3.23 in terms of j-relevant tuples such that

> i_{j+1} is the smallest node in w for which there is a tuple $(q_1, q_2, I) \in R_j(i_{j+1})$ with $I \neq \emptyset$.

Notice that if $R_j(i_j)$ itself contains such a tuple then $i_{j+1} = i_j$. Otherwise, we compute i_{j+1} by traversing the tree N_w^{aux} using the following lemma.

6.3.27 Lemma. *For a node $v_{xy} \in \mathrm{N}_w^{aux}$ the relation $R_j(v_{xy})$ can be computed in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k)$ in each of the following cases:*

(1) v_{xy} is a leaf, $v_{xy} = i_j$, and $G(i_j)$ and $R_{j-1}(i_j)$ are known,

(2) v_{xy} has parent v, $x > i_j$, and $R_j(v)$ is known, and

(3) v_{xy} has child v, $y \geq i_j$, and $R_j(v)$ and $R_{j-1}(v_{xy})$ are known.

Proof. We prove cases (1) to (3) separately. By Definition 6.3.26 we have to show that $R_j(v_{xy})$ for each case (1) to (3) is the set of all tuples (q_1, q_2, I) such that

- $(q_1, q_2, I) \in R(v_{xy})$, and
- (q_1, q_2, I) is compatible with a growing answer $(q, A) \in G(i_j)$.

(1) We define $R_j(i_j) = \{(p, q, \{q\}) \in R_{j-1}(i_j) \mid \exists (q, A) \in G(i_j)\}$, which can be computed in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k)$. Next, we show that $R_j(i_j)$ is computed correctly. Let $(p, q, \{q\})$ be as specified above. The tuple $(p, q, \{q\})$ is compatible with $(q, A) \in G(i_j)$ by definition and, therefore, also relevant. Now, assume that there is a tuple $(p, q, \{q\}) \in R(i_j)$ that is compatible with a growing answer $(q, A) \in G(i_j)$ but not in $R_{j-1}(i_j)$. Since every tuple $(p, q, \{q\}) \in R(i_j)$ that is compatible with $(q, A) \in G(i_j)$ is also compatible with a growing answer $(q', A') \in G(i_{j-1})$, it follows that $(p, q, \{q\}) \in R_{j-1}(i_j)$.

(2) We distinguish two subcases (a) and (b) depending on whether v_{xy} is a left or a right child.

(a) If v_{xy} is a left child and v is the parent of v_{xy} then let v' be the right child of v. We define

$$R_j(v_{xy}) = \{(q_1, p, J) \mid \exists (q_1, p, I_1) \in \mathrm{T}^+(v_{xy}), \exists (p, q_2, I_2) \in T^+(v'), \\ \exists (q_1, q_2, I) \in R_j(v), \text{ such that } J \subseteq I_1 \text{ and } J \cup I_2 = I\}.$$

(b) If v_{xy} is a right child and v is the parent of v_{xy} then let v' be the left child of v. We define

$$R_j(v_{xy}) = \{(p, q_2, J) \mid \exists (p, q_2, I_2) \in \mathrm{T}^+(v_{xy}), \exists (q_1, p, I_1) \in T^+(v'), \\ \exists (q_1, q_2, I) \in R_j(v) \text{ such that } J \subseteq I_2 \text{ and } J \cup I_1 = I\}.$$

We prove that the above definition for $R_j(v_{xy})$ is correct for the case (2)(a). The proof of case (2)(b) is analogous.
Let (q_1, p, J) be as specified above. We first prove that (q_1, p, J) belongs to $R_j(v_{xy})$. Since $(q_1, q_2, I) \in R_j(v)$ we know that (q_1, q_2, I) is compatible with a growing answer $(q, A) \in G(i_j)$. Since $x > i_j$ it holds that $\mathrm{Nodes}(A_{[x,y]}) = \emptyset$. Thus, (q_1, p, J) is compatible with (q, A) and, therefore, relevant.
Towards a contradiction, assume that $R_j(v_{xy})$ is not complete, i.e., there is a tuple $(q_1, p, J) \in R(v_{xy})$ that is compatible with some $(q, A) \in G(i_j)$ but

not captured by the right side in the above definition. However, we know that $(q_1, p, J) \in R(v_{xy})$. By Definition 6.3.17 it follows that there is a tuple $(q_1, p, I_1) \in \mathrm{T}^+(v_{xy})$ with $J \subseteq I_1$, and that there are tuples $(p, q_2, I_2) \in T^+(v')$ and $(q_1, q_2, I) \in R_j(v)$ such that $J \cup I_2 = I$. Therefore, $(q_1, p, J) \in R_j(v_{xy})$ which contradicts the assumption. We can compute $R_j(v_{xy})$ in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k)$.

(3) We distinguish two subcases (a) and (b) depending on whether v_{xy} has a left or a right child.

(a) If v_{xy} has a left child v then let v' be the right child of v_{xy}. We define

$$R_j(v_{xy}) = \{(q_1, q_2, I) \mid \exists (q_1, q_2, I) \in R_{j-1}(v_{xy}), \exists (q_1, p, I_1) \in R_j(v), \\ \exists (p, q_2, I_2) \in T^+(v') \text{ such that } I_1 \cup I_2 = I\}.$$

(b) If v_{xy} has a right child v then let v' be the left child of v_{xy}. We define

$$R_j(v_{xy}) = \{(q_1, q_2, I) \mid \exists (q_1, q_2, I) \in R_{j-1}(v_{xy}), \exists (q_1, p, I_1) \in T^+(v'), \\ \exists (p, q_2, I_2) \in R_j(v) \text{ such that } I_1 \cup I_2 = I\}.$$

We prove that the above definition for $R_j(v_{xy})$ is correct for the case (3)(a). The proof of case (3)(b) is analogous.
Let (q_1, q_2, I) be as specified above. We first prove that (q_1, q_2, I) belongs to $R_j(v_{xy})$. Since $(q_1, q_2, I) \in R_{j-1}(v_{xy})$ it holds that $(q_1, q_2, I) \in R(v_{xy})$. It remains to show that there is an annotated answer A^{full} w.r.t. some run r_A such that (q_1, q_2, I) and a growing answer $(q, A) \in G(i_j)$ are compatible with A^{full} and r_A. In the following, we construct such an annotated answer A^{full} and run r_A from the given information about (q_1, q_2, I). First, we know that (q_1, q_2, I) is compatible with a growing answer $(p, B) \in G(i_{j-1})$ because $(q_1, q_2, I) \in R_{j-1}(v_{xy})$. That is, there is an annotated answer B^{full} w.r.t. some run r_B such that (q_1, q_2, I) and (p, B) are compatible with B^{full} and r_B. In particular, it holds that $\mathrm{Nodes}(B) = \mathrm{Nodes}(A_{[1,i_{j-1}]})$. Moreover, we know that, by the definition of compatibility, $\mathrm{Nodes}(B^{\text{full}}_{[x,y]}) = I$ and $r_B = r_1 r_2 r_3$ such that r_2 is a partial run $q_1 \cdots q_2$ on $w[x..y]$ and $I \subseteq \mathrm{set}(r_2)$. However, this is not sufficient to show that B^{full} is compatible with $(q, A) \in G(i_j)$ because B^{full} does not have to agree with A on the node i_j. If this is the case then we can construct a new run r_A from r_B such that r_A produces the desired A^{full}. Therefore, we define $r_A = r_1 \cdot r_v \cdot r_{v'} \cdot r_3$ where

- $r_v = q_1 \cdots p$ is a partial run on $w[x..z]$ such that $I_1 \subseteq \mathrm{set}(r_v)$ and $r[i_j] = q$ for $(i_j, q) \in A$, and
- $r_{v'} = p' \cdots q_2$ with $p' \in \delta(p, w[z+1])$ is a partial run on $w[z+2..y]$ where $I_2 \subseteq \mathrm{set}(r_{v'})$.

Using Definition 6.3.26 and that $(q_1, p, I_1) \in R_j(v)$, we know that r_v is well-defined. Applying Lemma 6.3.6 on the tuple $(p, q_2, I_2) \in \mathrm{T}^+(v')$ it follows that $r_{v'}$ is well-defined. Since $I_1 \cup I_2 = I$ the run r_A produces the annotated answer A^{full} which proves the assumption.

Towards a contradiction, assume that $R_j(v_{xy})$ is not complete, i.e., there is a tuple $(q_1, q_2, I) \in R(v_{xy})$ that is compatible with a growing answer $(q, A) \in G(i_j)$ but not captured by the right side in the above definition. Let the annotated answer A^{full} w.r.t. some run r be such, that (q_1, q_2, I) and (q, A) are compatible with A^{full} and r. By the definition of compatibility we have that $\text{Nodes}(A^{\text{full}}_{xy}) = I$. Thus, there exist I_1 and I_2 such that, for $z = x - 1 + \lfloor \frac{y-x+1}{2} \rfloor$, it holds that $\text{Nodes}(A^{\text{full}}_{xz}) = I_1$, $\text{Nodes}(A^{\text{full}}_{(z+1)y}) = I_2$, and $I = I_1 \cup I_2$. It follows that there exist a tuple $(q_1, p, I_1) \in R_j(v_{xz})$ for the left child v_{xz} of v, and a tuple $(p, q_2, I_2) \in \mathrm{T}^+(v_{(z+1)y})$ for the right child $v_{(z+1)y}$ of v. Therefore, $(q_1, q_2, I) \in R_j(v_{xy})$ which contradicts the assumption. We can compute $R_j(v_{xy})$ in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k)$. □

In the following we argue that we need at most $\log n$ operations of the kind (1) to (3) to compute i_{j+1} from i_j. We start at node i_j where $G(i_j)$ and $R_{j-1}(i_j)$ are known. We compute $R_j(i_j)$ using (1) and test whether $i_{j+1} = i_j$. If this is not the case then we follow the path p from i_j to the root of $\mathrm{N}^{\text{aux}}_w$ and compute R_j on the way. Since we compute R_j for every node on p, we can always apply case (3). In total, this needs $\log n$ operations because p is of length $\log n$. Afterwards, we do a second bottom-up traversal of p where we compute, at each visited node, R_j for the right child of the node (applying case (2)). We stop when we find such a right child v_r which is not on p and where $R_j(v_r)$ contains a tuple (q_1, q_2, I) with $I \neq \emptyset$. Again, this can be done with at most $\log n$ operations. By the definition of R_j we know that the subtree rooted at v_r has at least one leaf node u such that $R_j(u)$ contains a tuple (q_1, q_2, I) with $I \neq \emptyset$. The leftmost of these nodes u is i_{j+1}. To arrive at i_{j+1} we go down from v_r. On the way we always compute R_j for both children (applying case (2)) and choose the leftmost child for which R_j has a tuple (q_1, q_2, I) with $I \neq \emptyset$. We are done at the moment we reach a leaf. Altogether, we navigated through $O(\log n)$ nodes in the tree.

The following characterization illustrates how $G(i_{j+1})$ can be obtained from $G(i_j)$ using j-relevant tuples:

$$G(i_{j+1}) = \{(q_2, A \cup \{\!\{(i_{j+1}, q_2)\}\!\}) \mid \exists (q_1, q_2, \{q_2\}) \in R_j(i_{j+1}) \\ \exists (q, A) \in G(i_j), \text{ and } q_1 \in \delta^*(q, w[i_j + 1..i_{j+1}])\}.$$

By maintaining reachable states in δ^* when going from i_j to i_{j+1}, we can compute i_{j+1} from $G(i_j)$ in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log n)$. Next, we show that all sets $G(i_j)$ can be efficiently stored in one shared data structure that is maintained during a computation of the algorithm.

6.3.28 Theorem. *All sets $G(i_j)$ (for $j \in \{1, \ldots, k\}$) can be stored in a data structure of size $O(|S| \cdot k!)$. This data structure, given j, can generate $G(i_j)$ within time $O(|S| \cdot k!)$.*

Proof. We first show that $O(|S| \cdot k!)$ records suffice for storing a single $G(i_j)$ and then show how all of them can be stored in a data structure of size $O(|S| \cdot k!)$.

For a given j we defined $G(i_j)$ as a set of tuples (q, A) where A is a multiset over $\text{Nodes}(w) \times Q$. By the construction of $G(i_j)$ every annotated answer A with

$(q, A) \in G(i_j)$ uses the same multiset $\{\!\{i_1, \ldots, i_j\}\!\}$ of nodes. Since k is an upper bound for j, a naïve upper bound for the size of $G(i_j)$ is $O(k + |Q| \cdot |Q|^k)$, i.e., size k for the tuple $(i_1, \ldots, i_j)$ and size $|Q| \cdot |Q|^k$ for all possible $(q, (q_1, \ldots, q_j))$ such that $(q, \{\!\{(i_1, q_1), \ldots, (i_j, q_j)\}\!\}) \in G(i_j)$. Moreover, for every $(q, A) \in G(i_j)$ and $(i_j, q_j) \in A$, it holds that $q_j = q$ by the construction of $G(i_j)$. Therefore, it is sufficient to store $(q_1, \ldots, q_j)$ instead of all possible tuples $(q, (q_1, \ldots, q_j))$.

This can be optimized even further. Observe that every growing answer in $G(i_j)$ does not contain arbitrary states but rather states in a set(s) for a selecting tuple $s \in S$. Thus, we can store, for every selecting tuple $(q_1, \ldots, q_k) \in S$, all injections $\sigma : \{1, \ldots, j\} \to \{1, \ldots, k\}$ such that

$$(q_{\sigma(j)}, \{\!\{(i_1, q_{\sigma(1)}), \ldots, (i_j, q_{\sigma(j)})\}\!\}) \in G(i_j).$$

Hence, for $j = k$, we store the tuple $(i_1, \ldots, i_k)$ once and, for every $(q_1, \ldots, q_k) \in S$, the set of permutations $\sigma : \{1, \ldots, k\} \to \{1, \ldots, k\}$ such that

$$(q_{\sigma(k)}, \{\!\{(i_1, q_{\sigma(1)}), \ldots, (i_k, q_{\sigma(k)})\}\!\}) \in G(i_k).$$

The total size of this data structure is $O(k + k! \cdot |S|)$, i.e., $O(k! \cdot |S|)$. (The representation size for $j < k$ is smaller.)

In fact, we can even store all $G(i_j)$ in a single data structure of size $O(k! \cdot |S|)$. To see this, consider a tuple $(q_1, \ldots, q_k)$ and all injections that are specified above. If $\sigma : \{1, \ldots, j\} \to \{1, \ldots, k\}$ is such, that $(q_{\sigma(j)}, \{\!\{(i_1, q_{\sigma(1)}), \ldots, (i_j, q_{\sigma(j)})\}\!\}) \in G(i_j)$ then there are two options for $(q_{\sigma(j)}, \{\!\{(i_1, q_{\sigma(1)}), \ldots, (i_j, q_{\sigma(j)})\}\!\})$. Either we can extend it to a growing answer in $G(i_{j+1})$ or not. In the former case, there is also a non-empty set of injections of the form $\sigma' : \{1, \ldots, j+1\} \to \{1, \ldots, k\}$ such that

- σ' extends σ, i.e., $\sigma'(\ell) = \sigma(\ell)$ for every $\ell \in \{1, \ldots, j\}$, and
- $(q_{\sigma'(j+1)}, \{\!\{(i_1, q_{\sigma'(1)}), \ldots, (i_{j+1}, q_{\sigma'(j+1)})\}\!\}) \in G(i_{j+1})$.

In the latter case, there exists no such non-empty set of injections extending σ.

Notice that, in the former case, σ can be immediately inferred from σ' such that σ does not have to be stored separately. For every j we therefore only store the injection $\sigma : \{1, \ldots, j\} \to \{1, \ldots, k\}$ that cannot be extended to an injection $\sigma' : \{1, \ldots, j\} \to \{1, \ldots, k+1\}$ for $G(i_{j+1})$. In particular, we store all permutations that encode answers for $j = k$.

We claim that the total number of injections that we store is $O(k! \cdot |S|)$, which can be seen as follows: for every tuple $s \in S$, each injection σ that we store cannot be extended to a permutation with the correct properties. That is, there exist permutations that extend σ, but these do not have the property that they produce the answer $(i_1, \ldots, i_k)$. Therefore, the total number of injections that we store is $O(k!)$. The size of a single such injection is not larger than the size of a tuple in S, which means that $O(|S| \cdot k!)$ records are sufficient in total. Finally, we obtain $G(i_j)$ for a given j by returning all growing answers $(q_{\sigma(j)}, \{\!\{(i_1, q_{\sigma(1)}), \ldots, (i_j, q_{\sigma(j)})\}\!\})$ for that we store an injection σ that is defined on j. □

Now, we have all the ingredients to show how long it takes to compute a set of growing answers $G(i_\ell)$ for an arbitrary $\ell \in \{1, \ldots, k\}$.

6.3.29 Lemma. *Given* N_w^{aux}, T^+, *and* $\ell \in \{1, \ldots, k\}$, *the set* $G(i_\ell)$ *can be computed in time* $O(|S| \cdot k! + |Q|^3 \cdot |S|^2 \cdot 2^k \cdot \ell \log n)$.

Proof. First, we show how much time is needed to compute the node i_ℓ. Second, we examine the time that was spend to compute the set $G(i_\ell)$ during the computation.

We know that i_1 can be computed in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log n)$ by Lemma 6.3.21. Here, we show how to compute $i_{\ell+1}$ in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log n)$ when i_ℓ is given. Therefore, we use that the following information is available. Besides the auxiliary data structure $\mathrm{N}_w^{\mathrm{aux}}$ with T^+ and all nodes in the multiset $N = \{\!\{i_1, \ldots, i_\ell\}\!\}$, we assume that we know

(a) $G(i_\ell)$ and $R(u_{i_\ell})$,

(b) the tree $\mathrm{A}_w^{\mathrm{aux}}$ induced by all ancestors of nodes in N, and

(c) for every node $v_{xy} \in \mathrm{A}_w^{\mathrm{aux}}$ the relation $R_{j-1}(v_{xy})$ or $R_j(v_{xy})$ where $i_j \in N$ is the highest position with $x \leq i_j \leq y$.

After the computation of $G(i_1)$, the tree $\mathrm{A}_w^{\mathrm{aux}}$ contains only the path from the root of $\mathrm{N}_w^{\mathrm{aux}}$ to i_1. Since $R_0(v_{xy}) = R(v_{xy})$ we compute $R_0(v_{xy})$ for every node on this path during the computation of i_1. Therefore, information (a) to (c) is available after we have computed $G(i_1)$.

Now, assume that all necessary information up to node i_ℓ has been computed. By Proposition 6.3.23 we know that

- $i_{\ell+1} \geq i_\ell$ is the leftmost node in w for that there exists a tuple $(q_1, q_2, I) \in R(i_{\ell+1})$ with $I \neq \emptyset$ that is compatible with a growing answer $(q, A) \in G(i_\ell)$.

By Definition 6.3.26 this is equivalent to

- $i_{\ell+1} \geq i_\ell$ is the leftmost node in w for that there exists a tuple $(q_1, q_2, I) \in R_\ell(i_{\ell+1})$ with $I \neq \emptyset$.

In the following, we show $i_{\ell+1}$ can be computed in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log n)$ providing that information (a) to (c) is available afterwards. Therefore, we argue that we need at most $\log n$ operations of the kind (1) to (3) from Lemma 6.3.27 to find $i_{\ell+1}$ from i_ℓ. We start at node i_ℓ where we assume that $G(i_\ell)$ and $R_{\ell-1}(i_\ell)$ are known. We compute $R_\ell(i_\ell)$ by applying (1) from Lemma 6.3.27 and test whether $i_{\ell+1} = i_\ell$. If this is not the case we follow the path p from i_ℓ to the root of $\mathrm{N}_w^{\mathrm{aux}}$ and compute R_ℓ on the way. (Notice that all necessary information (a) to (c) is still available afterwards.) Since we always compute the relation R_ℓ for every node on p we can always apply case (3) from Lemma 6.3.27. In total, this needs $\log n$ operations because p has length $\log n$. Afterwards, we do a second bottom-up traversal of p and compute, at each node, the relation R_ℓ for every right child (applying case (2) from Lemma 6.3.27). Again, all necessary information (a) to (c) is still available afterwards. We stop at the moment we find such a right child v_r that is not on p and where $R_\ell(v_r)$ contains a tuple (q_1, q_2, I) with $I \neq \emptyset$. Again, this can be done with at most $\log n$ operations. By the definition of R_ℓ we know that the subtree rooted at v_r

has at least one leaf node u such that $R_\ell(u)$ contains a tuple (q_1, q_2, I) with $I \neq \emptyset$. The leftmost of these nodes u is $i_{\ell+1}$. To arrive at $i_{\ell+1}$ we go down from v_r. On the way we always compute R_ℓ for both children (applying case (2) in Lemma 6.3.27) and choose the leftmost child for which R_ℓ contains a tuple (q_1, q_2, I) with $I \neq \emptyset$. We are done when we reach a leaf. Altogether, we navigated through $O(\log n)$ nodes in the tree $\mathrm{N}_w^{\mathrm{aux}}$ using time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \ell \log n)$ in total. Furthermore, we computed R_ℓ for every node on the path from the root of $\mathrm{N}_w^{\mathrm{aux}}$ to $i_{\ell+1}$, thereby, obtaining the necessary information for (b) and (c).

Finally, the following characterization shows how we can obtain $G(i_{\ell+1})$ from $G(i_\ell)$ using j-relevant tuples:

$$G(i_{\ell+1}) = \{(q_2, A \cup \{\!\{(i_{\ell+1}, q_2)\}\!\}) \mid \exists (q_1, q_2, \{q_2\}) \in R_\ell(u_{i_{\ell+1}}) \exists (q, A) \in G(i_\ell), \\ \text{and } q_1 \in \delta^*(q, w[i_\ell + 1..i_{\ell+1}])\}.$$

Afterwards, information (a) is available as well. When going from i_ℓ to $i_{\ell+1}$, we can maintain reachable states in δ^* in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log n)$. By Theorem 6.3.28 the computation of every set $G(i_j)$ for $j \in \{1, \ldots, \ell\}$ needs time $O(|S| \cdot k!)$. In total, we used time $O(|S| \cdot k! + |Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log n)$ to compute $G(i_\ell)$. □

Notice that the additional term $|S| \cdot k!$ comes from the size of the sets $G(i_\ell)$ which is naïvely $O(|Q|^k)$ but can be shown to be $O(|S| \cdot k!)$. Combining Lemmas 6.3.24 and 6.3.29 we obtain the following result.

6.3.30 Theorem. *Given N_w^{aux} and T^+, the first answer in $M(w)$ can be computed in time $O(|S| \cdot k! + |Q|^3 \cdot |S|^2 \cdot 2^k \cdot k \log n)$.*

Proof. Let $(v_1, \ldots, v_k)$ be the smallest answer in $M(w)$ (with respect to $\preceq$) such that there is an accepting run r of N on w, a tuple $(p_1, \ldots, p_k) \in S$, and r visits v_ℓ in p_ℓ for every $\ell \in \{1, \ldots, k\}$. By Lemma 6.3.24 the set $G(i_k)$ contains the set of smallest annotated answers. Therefore, $G(i_k)$ contains an annotated answer $A^{\mathrm{full}} = \{\!\{(v_1, p_1), \ldots, (v_k, p_k)\}\!\}$. The set $G(i_k)$ can be computed in time $O(|S| \cdot k! + |Q|^3 \cdot |S|^2 \cdot 2^k \cdot k \log n)$ by Lemma 6.3.29. □

In particular, we have that Complete($\{\emptyset\}$) returns the set $\{A \mid (q, A) \in G(i_k)\}$.

6.3.5 Computing the Next Answer to a Query

In the previous section we illustrated how to compute Complete($\{\emptyset\}$), which is the set of first answers to the query M on w. In the following, we explain how to go from one answer to the next, i.e., the details of the procedure **Next**($\mathcal{A}$) in Algorithm 2.

6.3.31 Lemma. *Complete, Back, and Nextnode can be implemented such that Algorithm 2 correctly computes Next($\mathcal{A}$). Moreover, Next($\mathcal{A}$) runs in $O(|S| \cdot k! + |Q|^3 \cdot |S|^2 \cdot 2^k \cdot k \log n)$ time.*

Proof. Recall from Section 6.3.3 that every $\mathcal{A}$ at each call of Complete, Back, or Nextnode has the property that all $A \in \mathcal{A}$ use the same multiset of nodes $\{\!\{i_1, \ldots, i_\ell\}\!\}$

(see Observation 6.3.14). We denote this multiset by Nodes($\mathcal{A}$) and assume that the following information is available at the time we call Complete, Back, Nextnode, or Next($\mathcal{A}$): the tree $\mathrm{N}_w^{\mathrm{aux}}$ with T^+ (entirely), the relations R_j, and sets G as described in the invariants (I1) and (I2) below.

(I1) Let $\mathrm{A}_w^{\mathrm{aux}}$ be the tree that is induced by all nodes in Nodes($\mathcal{A}$) and their ancestors in $\mathrm{N}_w^{\mathrm{aux}}$. For every $v_{xy} \in \mathrm{A}_w^{\mathrm{aux}}$ the relation $R_{j-1}(v_{xy})$ or $R_j(v_{xy})$, where $i_j \in$ Nodes($\mathcal{A}$) is the maximal node with $x \leq i_j \leq y$, is known.

(I2) For every $i_j \in$ Nodes($\mathcal{A}$) the set $G(i_j)$ is known. Here, $G(i_j) = \{(q, A) \mid$ Nodes$(A) \subseteq$ Nodes($\mathcal{A}$), $|A| = j$, and there is an annotated answer A^{full} such that Nodes$(A_{[1,i_j-1]})$ = Nodes$(A^{\mathrm{full}}_{[1,i_j-1]})$, Nodes$(A_{[i_j,i_j]}) \subseteq$ Nodes$(A^{\mathrm{full}}_{[i_j,i_j]})$, and (q, A) is compatible with $A^{\mathrm{full}}\}$.

In the following we show that we can implement all the aforementioned procedures correctly and such, that (I1) and (I2) hold afterwards.

Complete($\mathcal{A}$)*:* Algorithm 2 always calls Complete($\{\emptyset\}$) in the beginning. In Section 6.3.4, we proved that Complete($\{\emptyset\}$) returns the set of smallest annotated answers in $M(w)$ (if Complete($\{\emptyset\}$) = $\emptyset$ then the algorithm stops). Furthermore, we know that (I1) is satisfied by the computation done in the proof of Lemma 6.3.29 and that (I2) is satisfied by Claim 6.3.25 (which we proved for Lemma 6.3.24). We now generalize the description from Section 6.3.4 to compute Complete($\mathcal{A}$) for an arbitrary occurrence of $\mathcal{A}$ in Algorithm 2. To this end, we have to change the definition of the tuple $(i_1, \ldots, i_k)$ in Section 6.3.4. In particular, we assume that $(i_1, \ldots, i_k)$ is the smallest answer in $M(w)$ such that $\{\!\{i_1, \ldots, i_j\}\!\}$ = Nodes($\mathcal{A}$) and for every $\ell > j$ the position i_ℓ is greater or equal than the largest number i_j in Nodes($\mathcal{A}$). Moreover, we leave all sets $G(i_1), \ldots, G(i_j)$ untouched and recompute only the sets $G(i_\ell)$ for $\ell > j$. Since the relation R_j is available for every node that is described by (I1), we can compute i_{j+1} and the new set $G(i_{j+1})$ analogously to the computation for Complete($\{\emptyset\}$) by applying Lemma 6.3.27. Again, this computation satisfies that (I1) holds for the set Nodes(Complete($\mathcal{A}$)). At the newly computed node i_{j+1} we compute the set $G(i_{j+1})$ in the same way as for the case Complete($\{\emptyset\}$), i.e.,

$$G(i_{j+1}) = \{(q_2, A \cup \{\!\{(i_{j+1}, q_2)\}\!\}) \mid \exists (q_1, q_2, \{q_2\}) \in R_j(i_{j+1}), (q, A) \in G(i_j), \text{ such that } q_1 \in \delta^*(q, w[i_j + 1..i_{j+1}])\}.$$

However, notice that the semantics of the sets $G(i_j)$ (see (I2)) differ from Section 6.3.4. Finally, we define

$$\text{Complete}(\mathcal{A}) = \{A \mid (q, A) \in G(i_k)\}.$$

The computation of Complete($\mathcal{A}$) needs time $O(|S| \cdot k! + |Q|^3 \cdot |S|^2 \cdot 2^k \cdot k \log n)$ which can be proved analogously to Lemma 6.3.29. It remains to show that (I2) is satisfied after the computation. Observe that, when Complete($\mathcal{A}$) is finished, we have that Nodes(Complete($\mathcal{A}$)) = $\{\!\{i_1, \ldots, i_j, i_{j+1}, \ldots, i_k\}\!\}$ where $i_{j+1}, \ldots, i_k$ were recomputed

by the procedure. Thus, for all sets $G(i_\ell)$ with $\ell \leq j$, (I2) is satisfied by assumption. Next, we show that (I2) is satisfied for all all sets $G(i_\ell)$ where $\ell > j$. Therefore, we prove that, if $G(i_j)$ satisfies (I2) then $G(i_{j+1})$ that is given by the above computation satisfies (I2) as well. First, notice that $\mathrm{Nodes}(A) \subseteq \mathrm{Nodes}(\mathcal{A})$ by definition. By applying (I2) on $G(i_j)$ we know that $|A| = j$, i.e., $|A \cup \{\!\{(i_{j+1}, q_2)\}\!\}| = j+1$. Using the same argument we also know that there exists an annotated answer A_1^{full} w.r.t. some run r_1 such that $(q, A) \in G(i_j)$ is compatible with A_1^{full} and r_2. Then, we know that, by the definition of R, there is another annotated answer A_2^{full} w.r.t. some run r_2 such that $(q_1, q_2, \{q_2\}) \in R_j(i_{j+1})$ is compatible with A_2^{full} and r_2. Since we know that q_1 is reachable from q according to the right subword of w, it is straightforward to obtain an annotated answer A^{full} w.r.t. some run r such that $(q_1, q_2, \{q_2\})$ and (q, A) are compatible with A^{full} and r. It follows that (I2) is satisfied for $G(i_{j+1})$ which concludes the description of $\mathrm{Complete}(\mathcal{A})$.

Back($\mathcal{A}$)*:* The procedure $\mathrm{Back}(\mathcal{A})$ is implemented as follows:

$$\mathrm{Back}(\mathcal{A}) = \begin{cases} \{(A \mid (q, A) \in G(i_{j-1})\}, & \text{if } j \geq 2, \\ \emptyset, & \text{otherwise.} \end{cases}$$

If (I1) and (I2) are satisfied before calling $\mathrm{Back}(\mathcal{A})$ then they are satisfied afterwards because we do not touch any relation R and G. Since $G(i_{j-1})$ is already computed, each call of $\mathrm{Back}(\mathcal{A})$ needs linear time in the size of $G(i_{j-1})$, i.e., $O(|S| \cdot k!)$ by Theorem 6.3.28. It remains to prove that our definition is correct. Therefore, observe that every $A \in \mathrm{Back}(\mathcal{A})$ is an incomplete answer of size $j-1$ (by (I2) for $G(i_{j-1})$). Using the same argument, $\mathrm{Back}(\mathcal{A})$ contains all possible incomplete answers A in $M(w)$ with $\mathrm{Nodes}(A) = \{\!\{i_1, \ldots, i_{j-1}\}\!\}$. This concludes the description of $\mathrm{Back}(\mathcal{A})$.

Nextnode($\mathcal{A}$)*:* Let i_j be a maximal node in $\mathrm{Nodes}(\mathcal{A})$. $\mathrm{Nextnode}(\mathcal{A})$ checks whether there is an annotated answer $\mathcal{A}^{\text{full}}$ with $\mathrm{Nodes}(A^{\text{full}}) = \{\!\{i_1, \ldots, i_{j-1}, i_{j+1}, \ldots, i_k\}\!\}$ where $i_{j+1}, \ldots, i_k$ are nodes that are larger than i_j. The procedure returns the following set of incomplete answers:

$$\mathrm{Nextnode}(\mathcal{A}) = \begin{cases} \{(A \mid (q, A) \in G(i_{j+1})\}, & \text{if } \mathcal{A}^{\text{full}} \text{ exists}, \\ \emptyset, & \text{otherwise.} \end{cases}$$

Notice that $\mathrm{Nextnode}(\mathcal{A})$ recomputes only the node i_{j+1} and the set $G(i_{j+1})$. In Section 6.3.4, we showed how to compute the node i_{j+1} for the case $\mathrm{Complete}(\mathcal{A})$. In the case of $\mathrm{Nextnode}(\mathcal{A})$, the proof is analogous except from the following detail: one has to ensure that i_{j+1} is strictly larger than i_j (if the node exists). This can be done by skipping the step where we check whether $i_{j+1} = i_j$ in the beginning of the computation. The definition of $G(i_{j+1})$ is equal to the aforementioned case of $\mathrm{Complete}(\mathcal{A})$. Analogously to the case of $\mathrm{Complete}(\mathcal{A})$ the computation satisfies (I1) and (I2) afterwards. In total, $\mathrm{Nextnode}(\mathcal{A})$ can be computed in time $O(|S| \cdot k! + |Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log n)$.

Next($\mathcal{A}$)*:* In Lemma 6.3.13, we proved that $\mathrm{Next}(\mathcal{A})$ is correct when $\mathrm{Complete}(\mathcal{A})$, $\mathrm{Back}(\mathcal{A})$, and $\mathrm{Nextnode}(\mathcal{A})$ can be implemented correctly. Above, we proved that

Nextnode($\mathcal{A}$) needs time $O(|S|\cdot k!+|Q|^3\cdot|S|^2\cdot 2^k\cdot\log n)$, Back($\mathcal{A}$) can be implemented in time $O(|S|\cdot k!)$, and Complete($\mathcal{A}$) runs in time $O(|S|\cdot k!+|Q|^3\cdot|S|^2\cdot 2^k\cdot k\log n)$. During a call of Next($\mathcal{A}$) there are at most $O(k)$ calls of Nextnode($\mathcal{A}$) and Back($\mathcal{A}$) and there is only one call of Complete($\mathcal{A}$). Altogether, Next($\mathcal{A}$) needs time $O(|S|\cdot k!+|Q|^3\cdot|S|^2\cdot 2^k\cdot k\log n)$ in total. Again, the term $|S|\cdot k!$ is due to the size of the sets G of growing answers. □

Finally, we can obtain our main result.

6.3.32 Theorem. IncrementalEnumeration *for a k-NSFA M and a word w with $|w|=n$ can be solved with an auxiliary data structure of size $O(|Q|^2\cdot|S|\cdot 2^k\cdot n)$ which can be computed in time $O(|Q|^3\cdot|S|^2\cdot 2^k\cdot n)$, maintained in time $O(|Q|^3\cdot|S|^2\cdot 2^k\cdot\log n)$ per update, and guarantees $O(|S|\cdot k!+|Q|^3\cdot|S|^2\cdot 2^k\cdot k\log n)$ enumeration delay between answers.*

Proof. During preprocessing our algorithm builds an auxiliary data structure that consists of the tree $\mathrm{N}_w^{\mathrm{aux}}$ and the relation T^+. By Lemma 6.3.7 the auxiliary data structure has size $O(|Q|^2\cdot|S|\cdot 2^k\cdot n)$ and can be computed in time $O(|Q|^3\cdot|S|^2\cdot 2^k\cdot n)$. By the same lemma it follows that updates can be maintained in time $O(|Q|^3\cdot|S|^2\cdot 2^k\cdot\log n)$. By Lemma 6.3.15 we know that we can use Algorithm 2 to correctly enumerate all answers in $M(w)$ if Complete($\mathcal{A}$), Back($\mathcal{A}$), and Nextnode($\mathcal{A}$) can be computed correctly within the required time. In Theorem 6.3.30 we showed that Complete($\{\emptyset\}$) can be implemented correctly such that it runs in time $O(|S|\cdot k!+|Q|^3\cdot|S|^2\cdot 2^k\cdot k\log n)$. Next($\mathcal{A}$) (including Complete($\mathcal{A}$), Back($\mathcal{A}$), and Nextnode($\mathcal{A}$)) can be computed in time $O(|S|\cdot k!+|Q|^3\cdot|S|^2\cdot 2^k\cdot k\log n)$ by Lemma 6.3.31. □

6.4 Incremental Enumeration for Trees

We now extend the algorithm from Section 6.3 to trees. More precisely, we show that, for a k-NSTA M (with states Q and selecting tuples S) and a tree t, we can solve IncrementalEnumeration with $O(|S|\cdot k!+|Q|^3\cdot|S|^2\cdot 2^k\cdot k\log^2|t|)$ delay, using an auxiliary data structure of size $O(|Q|^2\cdot|S|\cdot 2^k\cdot|t|)$ which can be updated within time $O(|Q|^3\cdot|S|^2\cdot 2^k\cdot\log^2|t|)$. This generalizes the following result by Balmin et al. [BPV04].

6.4.1 Theorem ([BPV04]). IncrementalEvaluation *for an NTA N and a tree t can be solved with an auxiliary data structure of size $O(|Q|^2\cdot|t|)$ which can be updated within time $O(|Q|^3\cdot\log^2\cdot|t|)$.*

We generalize Theorem 6.4.1 in two directions. First, we examine k-ary queries instead of boolean queries. Second, we show that the answer set of the query can be enumerated within a small delay. The main observation in this section is that the algorithm from Section 6.3 can be used together with the techniques in [BPV04].

In their paper, Balmin et al. presented an algorithm that simulates the evaluation of an NTA $N=(Q,\Sigma,\delta,F)$ on a tree t by maintaining a set of NFAs over the set of maximal heavy paths of t (see Definition 2.2.5). We briefly review the original algorithm of Balmin et al. in the following.

Let t_v be the subtree of t that is rooted at node v. In the preprocessing, the algorithm computes, for every $v \in t$, a new label that contains the original label of v and all states q of the NTA such that there exists a run λ on t_v and λ visits v in q. These new labels can be computed in time $O(|Q| \cdot |\delta| \cdot |t|)$ by one bottom-up pass through the tree. Moreover, the algorithm maintains an NFA for every maximal heavy path of t. Each of these NFAs operates on the new labels and simulates a part of the tree automaton by allowing only transitions that are compatible with the tree automaton. In addition, the algorithms builds for every NFA an auxiliary data structure that is analogous to $\mathrm{N}_w^{\mathrm{aux}}$ from Section 6.3.1.

If an update occurs then the auxiliary data structure of the NFA that is responsible for the maximal heavy path of v receives an update. The update is processed analogously to Section 6.3.1. However, the change to this structure can trigger changes in all auxiliary data structures of NFAs for heavy paths that are closer to the root. Since the number of maximal heavy paths that are crossed on the way is at most logarithmic in $|t|$ by Lemma 2.2.6, processing an update for incremental NTA evaluation boils down to processing $\log |t|$ updates on words for NFAs. (This also explains the complexity upper bound in Theorem 6.4.1.)

We fix the following notation for the remainder of the section. We denote by $M = ((Q, \Sigma, \delta, F), S)$ a k-NSTA, by t the binary input tree, and by Q_S the set $\cup_{s \in S} \mathrm{set}(s)$. In Section 6.4.1, we explain the preprocessing of our algorithm for IncrementalEnumeration when given M and t as input. Moreover, we define the auxiliary data structure that is constructed during the preprocessing and show how it is updated. Afterwards, we present the enumeration procedure for the set $M(t)$.

6.4.1 Preprocessing an Auxiliary Data Structure for Trees

Analogously to Balmin et al. our algorithm computes new labels for t, except that it stores some additional information about selecting states.

6.4.2 Definition. For each $v \in t$ the set $\mathrm{Reach}(v) \subseteq (Q \times 2^{Q_S})$ is inductively defined as follows:

- if v is a leaf then $\mathrm{Reach}(v) = \{(q, I) \mid (\mathrm{lab}(v) \to q) \in \delta, I = \{q\} \cap Q_S\}$,
- if v has a left child v_1 and a right child v_2 then
  $\mathrm{Reach}(v) = \{(q, I) \mid ((q_1, q_2, \mathrm{lab}(v)) \to q) \in \delta,$
  $(q_1, I_1) \in \mathrm{Reach}(v_1), (q_2, I_2) \in \mathrm{Reach}(v_2)$ and $I = I_1 \cup I_2 \cup \big(\{q\} \cap Q_S\big)\}$.

In other words, if we denote by M^q the NTA $(Q, \Sigma, \delta, \{q\})$ then $\mathrm{Reach}(v)$ contains all pairs (q, I) such that $t_v \in L(M^q)$, $I \subseteq Q_S$, and there exists a run of M^q on t_v that uses all states in I. Using above definition one can compute the sets $\mathrm{Reach}(v)$ for all $v \in t$ in time $O(|Q| \log |Q| \cdot |\delta| \cdot |S| \cdot 2^k \cdot |t|)$.

Next, we define a new labeling function $\mathrm{lab}'(v)$ for every node $v \in t$. Recall that we denote the *maximal heavy path of* t by $\mathrm{hp}(t)$ and the set of all maximal heavy paths of nodes in t by $\mathrm{HPaths}(t)$ (see Definition 2.2.5).

6.4.3 Definition. Let $v_n \cdots v_1$ be a path in $\mathrm{HPaths}(t)$ such that v_1 is a leaf and v_n is the node that is closest to the root. Then, lab' is inductively defined as follows:

- for $i = 1$, $\mathrm{lab}'(v_1) := \mathrm{lab}(v_1)$,
- for $i > 1$, let v'_{i-1} be the child of v_i not on $\mathrm{hp}(v_i)$,
  - if v'_{i-1} is the right child of v_i then $\mathrm{lab}'(v_i) := (\mathrm{lab}(v_i), \mathrm{Reach}(v'_{i-1}))$, and
  - if v'_{i-1} is the left child of v_i then $\mathrm{lab}'(v_i) := (\mathrm{Reach}(v'_{i-1}), \mathrm{lab}(v_i))$.

We now define an NFA N_p for every path $p \in \mathrm{HPaths}(t)$ such that N_p processes the word $\mathrm{lab}'(v_1) \cdots \mathrm{lab}'(v_n)$ (where $p = v_n \cdots v_1$). All NFAs N_p are defined over a common finite state set Q_N, a common finite alphabet Σ_N, and a common transition function δ_N.

6.4.4 Definition. Let M be a k-NSTA, t be a binary tree, and p be a path in $\mathrm{HPaths}(t)$. Then, the NFA N_p is defined as the tuple $(Q_N, \Sigma_N, \delta_N, q_0, F_p)$ where $Q_N = Q \uplus \{q_0\}$ is the finite state set, q_0 is a fresh initial state, Σ_N is a finite alphabet, and δ_N is the transition function which is defined as follows:

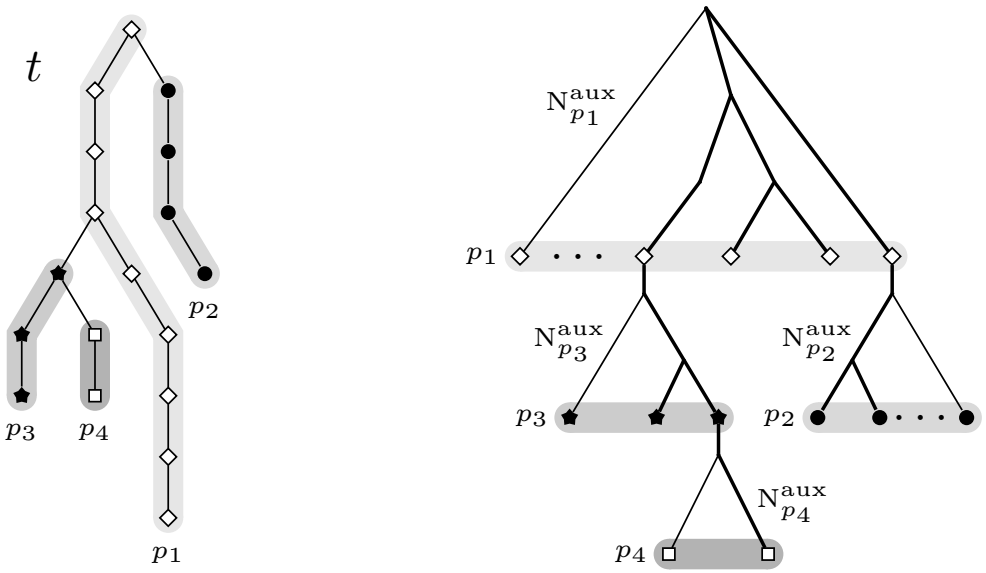
- $\delta_N(q_0, a) := Q_0$ where $Q_0 = \{q \mid (a \to q) \in \delta\}$,
- $\delta_N(q, (a, R)) := \cup_{(q', I) \in R} \delta(q, q', a)$ for all $a \in \Sigma$, and
- $\delta_N(q, (R, a)) := \cup_{(q', I) \in R} \delta(q', q, a)$ for all $a \in \Sigma$.

The set of accepting states F_p is equal to Q, if $p \neq \mathrm{hp}(t)$; and F, otherwise.

Intuitively, every NFA N_p simulates the tree automaton on the path p by allowing only transitions that are compatible with the tree automaton. We remark that our definition of the automata N_p (and of δ_N, respectively) is analogous to Balmin et al., except that we consider a labeling function that contains some additional information. The alphabet Σ_N, which is defined by the new labeling function lab', is of size $O(|\Sigma| \cdot |Q| \cdot 2^{|Q_S|})$. However, Balmin et al. showed that the alphabet and the transition function of the NFAs do not have to be stored explicitly. Instead, our algorithm stores only the sets $\mathrm{Reach}(v)$ for every node $v \in t$ and computes δ_N from the tree automaton on-the-fly.

Let $\mathrm{hp}(t)$ be the maximal heavy path of t. The NFA $N_{\mathrm{hp}(t)}$ accepts the word $\mathrm{lab}'(v_1) \cdots \mathrm{lab}'(v_n)$ if and only if the k-NSTA M accepts t. For all other paths $p \in \mathrm{HPaths}(t)$ the NFAs N_p are needed to propagate the updates in t. This concludes the description of the NFAs that are maintained during the algorithm.

The algorithm builds an auxiliary tree $\mathrm{N}_p^{\mathrm{aux}}$ for every NFA N_p and maximal heavy path p (analogously to Definition 6.3.1). Moreover, it computes the accompanying relations T_p^+ (see Definition 6.3.4) using the $\bowtie^+$-operation (see Definition 6.3.5) as before. The only difference to Section 6.3.2 is how it initializes the relations in the leaves. Notice that the leaf nodes of $\mathrm{N}_p^{\mathrm{aux}}$ are now associated with nodes in t which



(a) An example tree t.

(b) Part of the auxiliary data structure for t.

Figure 6.4: The algorithm traverses the auxiliary trees $\mathrm{N}^{\mathrm{aux}}_{p_i}$ for all maximal heavy paths $p_i \in \mathrm{HPaths}(t)$ which are linked according to their alignment in t.

can be the root of a subtree t_v. (Again, we use the convention that leaves $v_{xx} \in \mathrm{N}^{\mathrm{aux}}_p$ are also nodes in t). For nodes $v_{xx} \in \mathrm{N}^{\mathrm{aux}}_p$ the relation T^+_p is initialized as follows:

$$\mathrm{T}^+_p(v_{xx}) = \{(q_1, q_2, I) \mid q_2 \in \delta_N(q_1, \mathrm{lab}'(v_{xx})) \text{ and } I = (\cup_{(a,J)\in \mathrm{lab}'(v_{xx})} J) \cup (\{q_2\} \cap Q_S)\}.$$

For all other nodes $v_{xy} \in \mathrm{N}^{\mathrm{aux}}_p$ the relation T^+_p is defined exactly as in Section 6.3.2. This finishes the description of the preprocessing. In total, the auxiliary data structure includes t, the set $\mathrm{HPaths}(t)$, and the auxiliary trees $\mathrm{N}^{\mathrm{aux}}_p$ with relations T^+_p for each NFA N_p.

6.4.5 Example. In Figure 6.4(a) we illustrate a tree t with its maximal heavy paths $\mathrm{HPaths}(t) = \{p_1, p_2, p_3, p_4\}$. The auxiliary trees $\mathrm{N}^{\mathrm{aux}}_p$ for every $p \in \mathrm{HPaths}(t)$ are depicted in Figure 6.4(b). Nodes of maximal heavy paths in the left tree correspond to nodes of the same shape on the right. The set of auxiliary trees $\mathrm{N}^{\mathrm{aux}}_p$ can also be seen as a "tree of trees" for that the root of $\mathrm{N}^{\mathrm{aux}}_{p_4}$ provides information for a leaf node of $\mathrm{N}^{\mathrm{aux}}_{p_3}$, the root of $\mathrm{N}^{\mathrm{aux}}_{p_3}$ provides information for a leaf node of $\mathrm{N}^{\mathrm{aux}}_{p_1}$, et cetera. The set of auxiliary trees can be maintained under updates by propagating updates bottom-up in this "tree of trees". For example, assume that the highest node of p_4 is relabeled. Then, this corresponds to a relabeling of the rightmost leaf of $\mathrm{N}^{\mathrm{aux}}_{p_4}$. This change is then propagated on all nodes on the path to the root of $\mathrm{N}^{\mathrm{aux}}_{p_1}$, going through the auxiliary trees $\mathrm{N}^{\mathrm{aux}}_{p_3}$ and $\mathrm{N}^{\mathrm{aux}}_{p_1}$.

In principle, this procedure is very similar to the incremental evaluation algorithm of tree automata as described by Balmin et al. [BPV04]. The only difference

is that we maintain more involved sets Reach(v) (which explains the extra 2^k factor in complexity). Moreover, notice that $k = 0$ in [BPV04].

6.4.6 Lemma. *Given a k-NSTA M and a binary tree t, the auxiliary data structure has size $O(|Q|^2 \cdot |S| \cdot 2^k \cdot |t|)$, can be computed in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot |t|)$, and updated in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log^2 |t|)$.*

Proof. Together all trees $\mathrm{N}_p^{\mathrm{aux}}$ with $p \in \mathrm{HPaths}(t)$ have $|t|$ leaf nodes with at most $\log |t|$ depth, i.e., $O(|t|)$ nodes in total. For every node v in some $\mathrm{N}_p^{\mathrm{aux}}$ the transition relation T_p^+ is of size $O(|Q|^2 \cdot |S| \cdot 2^k)$. Therefore, the auxiliary data structure has size $O(|Q|^2 \cdot |S| \cdot 2^k \cdot |t|)$. The transition relation T_p^+ is built bottom-up for every tree $\mathrm{N}_p^{\mathrm{aux}}$. For leaf nodes the relation T_p^+ can be computed in time and space $O(|Q|^2)$. Afterwards, we need to compute $|t| \bowtie^+$-operations where each join needs time $O(|Q|^3 \cdot |S|^2 \cdot 2^k)$. In total, the auxiliary data structure is built in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot |t|)$.

Now, assume that an update occurs at a node $v \in t$. Let $p \in \mathrm{HPaths}(t)$ be the maximal heavy path that contains v. Then, every relation on the path from v to the root of $\mathrm{N}_p^{\mathrm{aux}}$ is updated, i.e., $\log |t|$ many relations. This needs $\log |t| \bowtie^+$-operations and yields time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log |t|)$ for the tree $\mathrm{N}_p^{\mathrm{aux}}$. Afterwards, the computation in the tree $\mathrm{N}_p^{\mathrm{aux}}$ triggers an update in every tree $\mathrm{N}_{p'}^{\mathrm{aux}}$ with $p' \in \mathrm{HPaths}(t)$ such that p' is crossed by the path from v to the root of t. By Lemma 2.2.6 there exist at most $\log |t|$ such paths p'. Therefore, we update at most $\log |t|$ trees $\mathrm{N}_{p'}^{\mathrm{aux}}$ where each update costs $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log |t|)$. In total, it needs time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log^2 |t|)$ to update all trees $\mathrm{N}_p^{\mathrm{aux}}$ on the path from v to the root of t. □

6.4.2 Enumerating Query Answers for Trees

Let $M = ((Q, \Sigma, \delta, F), S)$ be a k-NSTA and t be the binary input tree with $|t| = n$. To enumerate the set $M(t)$ we use a procedure that is very similar to Algorithm 2 in Section 6.3. The main differences are that (1) we now have to maintain several auxiliary trees $\mathrm{N}_p^{\mathrm{aux}}$ and relations T_p^+ (see Section 6.4.1), and (2) the enumeration procedure has to check, for every leaf in some tree $\mathrm{N}_p^{\mathrm{aux}}$, whether it has also reached a leaf node in t or has to proceed to the next tree $\mathrm{N}_{p'}^{\mathrm{aux}}$. To this end, the algorithm traverses the "tree of trees" (see, e.g., Figure 6.4(b)) that consists of all auxiliary trees $\mathrm{N}_p^{\mathrm{aux}}$ with $p \in \mathrm{HPaths}(t)$. We formally define this structure as follows.

6.4.7 Definition. Let HPaths(t) be the set of maximal heavy paths for the tree t. For every $p \in \mathrm{HPaths}(t)$ let $\mathrm{N}_p^{\mathrm{aux}}$ be the auxiliary tree for the NFA N_p. The auxiliary tree $\mathrm{N}_t^{\mathrm{aux}}$ consists of all $\mathrm{N}_p^{\mathrm{aux}}$ which are aligned and linked together as follows:

- the root of $\mathrm{N}_t^{\mathrm{aux}}$ is the root of $\mathrm{N}_{\mathrm{hp}(t)}^{\mathrm{aux}}$, and
- for every maximal heavy path p and leaf node $v \in \mathrm{N}_p^{\mathrm{aux}}$
  - if v has a child in t that is root of another heavy path p', then the child of v in $\mathrm{N}_t^{\mathrm{aux}}$ is the root of the tree $\mathrm{N}_{p'}^{\mathrm{aux}}$,
  - otherwise, v is a leaf in $\mathrm{N}_t^{\mathrm{aux}}$.

In other words, $\mathrm{N}_t^{\mathrm{aux}}$ can be seen as the union of all trees $\mathrm{N}_p^{\mathrm{aux}}$ with extra edges between some $\mathrm{N}_p^{\mathrm{aux}}$. Moreover, notice that a node in $\mathrm{N}_t^{\mathrm{aux}}$ is also a node in a tree $\mathrm{N}_p^{\mathrm{aux}}$ and that a leaf node in $\mathrm{N}_p^{\mathrm{aux}}$ is also a node in t.

In the following, we illustrate how we have to change Algorithm 2 from Section 6.3 such that it runs on the tree $\mathrm{N}_t^{\mathrm{aux}}$ and enumerates the set $M(t)$. We start by redefining the relevant relation R for trees. The following definition differs from Definition 6.3.17 only at the root nodes of the trees $\mathrm{N}_p^{\mathrm{aux}}$.

6.4.8 Definition. Let $p = v_n \cdots v_1 \in \mathrm{HPaths}(t)$ (where v_1 is a leaf) and let r be the root node of $\mathrm{N}_p^{\mathrm{aux}}$:

- if $p = \mathrm{hp}(t)$ then $R(r) = \{(q_0, q_F, \mathrm{set}(s)) \in T_p^+(r) \mid q_F \in F, s \in S\}$,
- if $p \neq \mathrm{hp}(t)$ then consider the parent v_p of v_n in t,
  - if v_n is a left child then $R(r) = \{(q_0, q_1, I') \mid \exists (q_0, q_1, I) \in \mathrm{T}_p^+(r)$ with $I' \subseteq I, \exists (q_2, q, J) \in R(v_p)$, $q \in \delta(q_1, q_2, \mathrm{lab}(v_p))$, and $J \subseteq I' \cup \{q\}\}$,
  - if v_n is a right child then $R(r) = \{(q_0, q_2, I') \mid \exists (q_0, q_2, I) \in \mathrm{T}_p^+(r)$ with $I' \subseteq I, \exists (q_1, q, J) \in R(v_p)$, $q \in \delta(q_1, q_2, \mathrm{lab}(v_p))$, and $J \subseteq I' \cup \{q\}\}$.

Intuitively, the tuples in the above relation are associated with partial runs of the tree automaton that are relevant for constructing an answer to the query $M(t)$.

We also adapt the output order of the algorithm for trees because nodes that can be selected by the tree automaton no longer correspond to leaves of $\mathrm{N}_t^{\mathrm{aux}}$. Instead, also internal nodes of $\mathrm{N}_t^{\mathrm{aux}}$ (i.e., leaves of individual $\mathrm{N}_p^{\mathrm{aux}}$) can be selected. We extend the output order by comparing individual nodes in $\mathrm{N}_t^{\mathrm{aux}}$ in terms of their postfix order, i.e., we define that $u \preceq_{\mathrm{t}} v$ if $u = v$ or u comes before v in the (left-to-right) post-order traversal of $\mathrm{N}_t^{\mathrm{aux}}$. Again, we define $\prec_{\mathrm{t}}$ as the strict version of $\preceq_{\mathrm{t}}$ and we extend $\preceq_{\mathrm{t}}$ to tuples analogously. Since $\mathrm{N}_t^{\mathrm{aux}}$ contains all nodes of t, the order $\preceq_{\mathrm{t}}$ (resp. $\prec_{\mathrm{t}}$) is also well-defined on the set of nodes in t.

In the remainder of this section, we prove that Algorithm 2 adapted with the aforementioned changes correctly enumerates all answers in $M(t)$. Therefore, we adapt the definition of an *(incomplete) annotated answer* (see Section 6.3.3) for trees. Formally, an *annotated answer* of a k-NSTA $M = (N, S)$ on t is a multiset A^{full} over $\mathrm{Nodes}(t) \times Q$ of the form

$$\{\!\{(i_1, q_1), \ldots, (i_k, q_k)\}\!\}$$

such that there is an accepting run λ of N on t and a tuple $(q_1, \ldots, q_k) \in S$ where λ visits i_ℓ in q_ℓ for every $\ell \in \{1, \ldots, k\}$. For a node $v \in \mathrm{N}_t^{\mathrm{aux}}$ let t_v be the subtree of $\mathrm{N}_t^{\mathrm{aux}}$ that is rooted at v. For an annotated answer $A = \{\!\{(i_1, q_1), \ldots, (i_k, q_k)\}\!\}$ and a set of nodes $V \subseteq \mathrm{Nodes}(\mathrm{N}_t^{\mathrm{aux}})$, the *projection of A onto $[V]$*, denoted $A_{[V]}$, is defined as the multiset $\{\!\{(i, q_i) \mid \exists v \in V :$ such that $i \in t_v\}\!\}$. All remaining notions for annotated answers are defined analogously to the word case.

The proof of the following lemma is analogous to the proof of Lemma 6.3.15 (when using the tree definitions for annotated answers).

6.4.9 Lemma. *Let M be a k-NSTA and t be a binary tree. Then, **Enum**(M, t) correctly enumerates all answers in $M(t)$.*

The First Answer for Trees Next, we illustrate how the set of smallest answers in $M(t)$ can be computed, i.e., we show how to implement Complete($\{\emptyset\}$) for trees. To this end, we adapt the notion of a *growing annotated answer up to position j* to trees. Again, we consider the definition for words (see Definition 6.3.16) but interpret it over Nodes(t) $\times$ Q:

6.4.10 Definition. Let $q \in Q$ be a state in Q, A be a multiset over Nodes(t) $\times$ Q, and $j \in \{1, \ldots, n\}$. Then, (q, A) is a *growing annotated answer up to node j* if there is an accepting run λ of N on t such that

- λ visits j in q, and
- there is an annotated answer A^{full} w.r.t. λ such that, for every $p \in Q$ and $i \in \text{Nodes}(t)$,
  - if $i \prec_{\text{t}} j$ then $A^{\text{full}}((i,p)) = A((i,p))$,
  - if $i = j$ then $A((i,p)) \leq A^{\text{full}}((i,p))$, and
  - if $i \succ_{\text{t}} j$ then $A((i,p)) = 0$.

Now, we are able to prove that the definition of the *relevant relation* (see Definition 6.4.8) for trees is correct. That is, we prove a statement similar to the one in Lemma 6.3.18 for trees.

For a node $v_{xy} \in \text{N}_p^{\text{aux}}$ where $p = v_n \cdots v_1$ (v_1 is a leaf) we define the *projected path* pp(v_{xy}) as $v_x \cdots v_y$. Notice that in pp(v_{xy}) we reverse the order on nodes $v_i \in p$ since we always have that $x \leq y$. Therefore, the definition of projected paths fits to the input words for the automata N_p because they read the labels of the path p bottom-up by definition.

6.4.11 Lemma. *Let $v_{xy} \in \text{N}_p^{aux}$, pp$(v_{xy}) = v_x \cdots v_y$, and $V = \{v_x, \ldots, v_y\}$. Then, $R(v_{xy})$ is the set of all tuples (q_1, q_2, I) such that there is an annotated answer A^{full} w.r.t. some run λ with $\lambda(v_y) = q_2$, $I = Nodes(A^{full}_{[V]})$, and, for $q_1 \neq q_0$, $\lambda(v_x) = q_1$.*

Proof. Since for every tuple $(q_1, q_2, I) \in R(v_{xy})$ there is a tuple $(q_1, q_2, J) \in \text{T}_p^+$ for some sets I and J, it holds analogously to Balmin et al. [BPV04] that there is a run λ on t such that $\lambda(v_y) = q_2$, and, for $q_1 \neq q_0$, $\lambda(v_x) = q_1$. The proof that the set I is correct is a straightforward induction on the position of a path in HPaths(t) compared to t. □

Next, we adapt the definition of compatibility (see Definition 6.3.22) to trees. The notion of compatibility for a tuple in R and an annotated answer is defined accordingly to the semantics of R for words (see Lemma 6.4.11), otherwise the definition is analogous to the word case.

6.4.12 Definition (Compatibility). Let $v_{xy} \in \mathrm{N}_t^{\mathrm{aux}}$, $\mathrm{pp}(v_{xy}) = v_x \cdots v_y$, and $V = \{v_x, \ldots, v_y\}$. For an annotated answer A^{full} w.r.t. some run λ, a tuple

- $(q_1, q_2, I) \in R(v_{xy})$ *is compatible with* A^{full} *and* λ if $\lambda(v_y) = q_2$, $I = \mathrm{Nodes}(A^{\mathrm{full}}_{[V]})$, and, for $q_1 \neq q_0$, $\lambda(v_x) = q_1$.

Furthermore, for (q, A) a growing answer up to position i,

- (q, A) is *compatible with* A^{full} *and* λ if $\lambda(i) = q$, $A_{[1,i-1]} = A^{\mathrm{full}}_{[1,i-1]}$ and $A_{[i,i]} \subseteq A^{\mathrm{full}}_{[i,i]}$.

Finally, $(q_1, q_2, I) \in R(v_{xy})$ *is compatible with* (q, A) if there exists an annotated answer A^{full} w.r.t. some run λ such that both (q_1, q_2, I) and (q, A) are compatible with A^{full} and λ.

Now, we are able to define the nodes i_1 (see Lemma 6.3.20) and i_{j+1} (see Proposition 6.3.23) for trees. Let $(i_1, \ldots, i_k)$ be the smallest answer in $M(t)$ in the following.

6.4.13 Lemma. *Let u be the smallest node in t (w.r.t. $\preceq_t$) such that $R(u)$ has a tuple (q_1, q_2, I) with $q_2 \in I$. Then, u is the node i_1 in t.*

The set of growing answers for i_1 is defined as

$$G(i_1) := \{(q_2, \{\!\{(i_1, q_2)\}\!\}) \mid (q_1, q_2, I) \in R(i_1) \text{ and } q_2 \in I\}.$$

6.4.14 Proposition. *The node $i_{j+1} \succeq_t i_j$ is the smallest node in t for which there is a tuple $(q_1, q_2, I) \in R(i_{j+1})$ with $q_2 \in I$ that is compatible with some $(q, A) \in G(i_j)$.*

Once we have i_{j+1}, we define

$$G(i_{j+1}) := \{(q_2, A \cup \{\!\{(i_{j+1}, q_2)\}\!\}) \mid \text{there is a tuple } (q_1, q_2, I) \in R(i_{j+1}) \text{ with } q_2 \in I \text{ that is compatible with some } (q, A) \in G(i_j)\}.$$

To compute the sets $G(i_j)$ we traverse the auxiliary data structure analogously to the word case, except that we traverse the tree $\mathrm{N}_t^{\mathrm{aux}}$. During this traversal the notion of *j-relevance* (see Definition 6.3.26) was central for the word case. The definition of j-relevance on trees is essentially the same as for words, except that it uses the definition of the relation R and the notion of compatibility for trees.

6.4.15 Definition. For $v \in \mathrm{N}_t^{\mathrm{aux}}$ and $j \in \{0, \ldots, k\}$, the set of *j-relevant tuples*, denoted $R_j(v)$, is defined as follows:

- $R_0(v) := R(v)$, and
- for each $j \geq 1$
  - if $v \prec_t i_j$ then $R_j(v) := R_{j-1}(v)$,
  - otherwise, $R_j(v) := \{(q_1, q_2, I) \in R(v) \mid (q_1, q_2, I)$ is compatible with a growing answer $(q, A) \in G(i_j)\}$.

As already mentioned, we traverse the tree $\mathrm{N}_t^{\mathrm{aux}}$ in the same way as the tree $\mathrm{N}_w^{\mathrm{aux}}$. However, the tree $\mathrm{N}_t^{\mathrm{aux}}$ has depth $O(\log^2 |t|)$ whereas $\mathrm{N}_w^{\mathrm{aux}}$ has depth $O(\log |w|)$. Therefore, it needs time $O(\log^2 |t|)$ to reach the node i_{j+1} from i_j. Moreover, it can happen that a node i_j is not a leaf node in $\mathrm{N}_t^{\mathrm{aux}}$. However, maintaining the j-relevant relation along the path from i_j to i_{j+1} can be done analogously as for words. In the following, we provide a version of Lemma 6.3.27 for trees.

6.4.16 Lemma. *For a node $v \in \mathrm{N}_t^{aux}$ the relation $R_j(v)$ can be computed in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k)$ in each of the following cases:*

(1) v is a leaf in N_t^{aux}, $v = i_j$, and $G(i_j)$ and $R_{j-1}(i_j)$ are known,

(2) v has a parent v_p, $i_j \preceq_t v_p$, and $R_j(v_p)$ is known, and

(3) v has a child v_c, $i_j \prec_t v$, and $R_j(v_c)$ and $R_{j-1}(v)$ are known.

Proof. We prove cases (1) to (3) separately. By Definition 6.4.15 we have to show that $R_j(v)$ for each case (1) to (3) is the set of all tuples (q_1, q_2, I) such that

- $(q_1, q_2, I) \in R(v)$, and
- (q_1, q_2, I) is compatible with a growing answer $(q, A) \in G(i_j)$.

(1) The proof is analogous to the proof of case (1) in Lemma 6.3.27.

(2) To prove (2) we distinguish two subcases (a) and (b).

(a) If v and v_p are nodes in the same tree $\mathrm{N}_p^{\mathrm{aux}}$ then the proof is analogous to the proof of case (2) in Lemma 6.3.27.

(b) Otherwise, we know that v is the root of some $\mathrm{N}_p^{\mathrm{aux}}$. (Notice that this is where Definition 6.4.8 differs from Definition 6.3.17.) Let p be a path $v_n \cdots v_1$ such that v_1 is a leaf in t. Then, we distinguish two subcases whether v_n is a left of a right child in t. Since both cases are analogous to each other, we prove only the case where v_n is a left child in the following. In this case, we define

$$R_j(v) = \{(q_0, q_1, I') \mid \exists (q_0, q_1, I) \in \mathrm{T}_p^+(v) \text{ with } I' \subseteq I, \\ \exists (q_2, q, I) \in R_j(v_p), q \in \delta(q_1, q_2, \mathrm{lab}(v_p)) \text{ and } I' \cup \{q\} = J\}.$$

Since $(q_2, q, I) \in R(v_p)$ we know that (q_2, q, I) is compatible with a growing answer $(q, A) \in G(i_j)$. Since $i_j \preceq_{\mathrm{t}} v_p$ it holds that $\mathrm{Nodes}(A_{[\{v\}]}) = \emptyset$. It follows that (q_0, q_1, I') is compatible with (q, A) and, therefore, that the tuple is relevant. Now, assume that there is a tuple $(q_0, q, I') \in R(v)$ that is compatible with a growing answer $(q, A) \in G(i_j)$ but that is not in $R_j(v)$. Since $(q_0, q, I') \in R(v)$ it holds that, by Definition 6.4.8, there is a tuple $(q_0, q, I) \in \mathrm{T}_p^+(v)$ with $I' \subseteq I$. By the Definition of compatibility we know that there also exists a tuple $(q_2, q, I) \in R_j(v_p)$ that is compatible with $(q, A) \in G(i_j)$ and that $q \in \delta(q_1, q_2, \mathrm{lab}(v_p))$ such that $I' \cup \{q\} = J$. Altogether, this contradicts the assumption that $(q_0, q, I') \notin R_j(v)$. We can compute $R_j(v)$ in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k)$.

(3) To prove (3) we distinguish two subcases (a) and (b).

(a) If v and v_c are nodes in the same tree $\mathrm{N}_p^{\mathrm{aux}}$ then the proof is analogous to the proof of case (3) in Lemma 6.3.27.

(b) Otherwise, we know that v is a leaf of some $\mathrm{N}_p^{\mathrm{aux}}$. (Notice that this is where Definition 6.4.8 differs from Definition 6.3.17.) Let p be a path $v_n \cdots v_1$ such that v_1 is a leaf in t. Then, we distinguish two subcases whether v_n in t is a left child or not. Since both cases are analogous to each other, we prove only the case where v_n is a left child in the following. In this case, we define

$$R_j(v) = \{(q_2, q, I) \mid (q_2, q, I) \in R_{j-1}(v), (q_0, q_1, J) \in R_j(v_c), \\ q \in \delta(q_1, q_2, \mathrm{lab}(v)) \text{ and } I = (\{q\} \cup J) \cap \mathrm{set}(s) \text{ for some } s \in S\}.$$

Since $(q_2, q, I) \in R_{j-1}(v)$ it follows that $(q_2, q, I) \in R(v)$. It remains to show that (q_2, q, I) is compatible with a growing answer $(q, A) \in G(i_j)$. That is, we need to find an annotated answer A^{full} w.r.t. some run r_A such that (q_2, q, I) and $(q, A) \in G(i_j)$ are compatible with A^{full} and r_A. The construction of such A^{full} and r_A is analogous to case (3)(b) in Lemma 6.3.27. We can compute $R_j(v)$ in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k)$. □

Now, the following results can be proved analogously to Lemma 6.3.29 and Theorem 6.3.30.

6.4.17 Lemma. *Given N_t^{aux}, T_p^+, and $\ell \in \{1, \ldots, k\}$, the set $G(i_\ell)$ can be computed in time $O(|S| \cdot k! + |Q|^3 \cdot |S|^2 \cdot 2^k \cdot \ell \log^2 |t|)$.*

6.4.18 Theorem. *Given N_t^{aux} and T_p^+, the first answer in $M(t)$ can be computed in time $O(|S| \cdot k! + |Q|^3 \cdot |S|^2 \cdot 2^k \cdot k \log^2 |t|)$.*

From One Answer to the Next for Trees Once we are able to compute Complete($\{\emptyset\}$), which is the set of smallest answer(s) in $M(t)$, we can compute the next set of answers using the procedure **Next**($\mathcal{A}$) in Algorithm 2. The following lemma shows that **Next**($\mathcal{A}$) works correctly for trees. The proof of the next lemma is analogous to the proof of Lemma 6.3.31.

6.4.19 Lemma. *Complete, Back, and Nextnode can be implemented such that Algorithm 2 for trees correctly computes Next($\mathcal{A}$). Furthermore, Next($\mathcal{A}$) runs in $O(|S| \cdot k! + |Q|^3 \cdot |S|^2 \cdot 2^k \cdot k \log^2 |t|)$ time.*

Finally, we can obtain our main result for IncrementalEnumeration when given a k-NSTA and a binary tree as input.

6.4.20 Theorem. IncrementalEnumeration *for a k-NSTA M and a tree t can be solved with an auxiliary data structure of size $O(|Q|^2 \cdot |S| \cdot 2^k \cdot |t|)$ which can be computed in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot |t|)$, maintained in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log^2 |t|)$ per update, and guarantees $O(|S| \cdot k! + |Q|^3 \cdot |S|^2 \cdot 2^k \cdot k \log^2 |t|)$ enumeration delay between answers.*

Proof. During the preprocessing the algorithm builds the tree $\mathrm{N}_t^{\mathrm{aux}}$ and relation T_p^+. By Lemma 6.4.6 the auxiliary data structure has size $O(|Q|^2 \cdot |S| \cdot 2^k \cdot |t|)$ and can be computed in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot |t|)$. By the same lemma it follows that updates can be maintained in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log^2 |t|)$. By Lemma 6.4.9 we know that we can use Algorithm 2 for trees to correctly enumerate all answers in $M(t)$ if Complete($\mathcal{A}$), Back($\mathcal{A}$), and Nextnode($\mathcal{A}$) can be computed correctly within the required time. In Theorem 6.4.18 we showed that Complete($\{\emptyset\}$) can be implemented correctly such that it runs in time $O(|S| \cdot k! + |Q|^3 \cdot |S|^2 \cdot 2^k \cdot k \log^2 |t|)$. Next($\mathcal{A}$) can be computed in time $O(|S| \cdot k! + |Q|^3 \cdot |S|^2 \cdot 2^k \cdot k \log^2 |t|)$ by Lemma 6.4.19. □

6.5 Incremental Enumeration under Multiset Semantics

When carefully examining the sets $G(i_j)$, one can observe that they contain detailed information on the states that are used to select positions in an annotated answer. According to the semantics of node selecting automata (see Section 6.1) our algorithm deletes this additional information for the output as well as duplicate answers.

In the following, we use this information to solve the IncrementalEnumeration problem for more expressive semantics, i.e., semantics that output every answer as often as a tuple in a set $G(i_j)$ contains this answer. We denote these semantics as *multiset semantics* and define them as follows: For a k-NSFA $M = ((Q, \Sigma, \delta, q_0, F), S)$ and an input word $w = a_1 \cdots a_n \in \Sigma^*$, the result of M on w under multiset semantics is denoted $M_{\mathrm{ms}}(w)$ and is a function that maps tuples $(i_1, \ldots, i_k) \in \{1, \ldots, n\}^k$ to $\mathbb{N}$. Formally we define, for each $(i_1, \ldots, i_k) \in \{1, \ldots, n\}^k$,

$$M_{\mathrm{ms}}(w)((i_1, \ldots, i_k)) = |\{(p_1, \ldots, p_k) \in S \mid \text{there is an accepting run } r \text{ of } N \text{ on } w \text{ such that } r \text{ visits } i_\ell \text{ in } p_\ell \text{ for every } \ell \in \{1, \ldots, k\}\}|.$$

For selecting tree automata and trees, multiset semantics are defined analogously. Intuitively, the output contains a tuple $(i_1, \ldots, i_k)$ as often as there are selecting tuples and runs that select it. For example, for the 2-NSFA M in Figure 6.1 and the word $w = abcd$ we have that

$$M_{\mathrm{ms}}(w) = \{\!\{(1,2), (1,2), (2,1), (2,1), (1,3), (3,1), (2,4), (4,2)\}\!\}.$$

The difference between the algorithms under set semantics (see Sections 6.3 and 6.4) and algorithms for IncrementalEnumeration (for word and tree automata) under multiset semantics only lies in the procedure output($\mathcal{A}$).

6.6 Spanners in the Context of Information Extraction

In *Information Extraction* (*IE*) one is interested in the automatic extraction of structured information from large texts. IE was formally considered already in 1987 [GS96] (at that time with the focus on military applications). It regained importance with the arising of big data in the new millennium. Our focus is on a specific information extraction tool, called *SystemT*, which is being developed at the IBM Almaden Research Center since 2005 and aims at providing an expressive vocabulary to formalize extraction rules and being user-friendly at the same time. At the present, SystemT is used for large scale text analytics [HKK+13], in the context of cleaning inconsistencies in databases [FKRV14], and back-end analytics in an enterprise search system at IBM Research [BKL12, BKL12]. During the development of SystemT, an SQL-like declarative language, which is denoted *Annotation Query Language* (or *AQL*), was introduced to provide a formalism to express information extraction rules that can be intuitively written by the human user [RRK+08, KLR+08, CKL+10].

The theoretical foundations of AQL and SystemT were first studied in a paper by Fagin et al. [FKRV13], where the authors introduced a formal framework for AQL. The intention of this framework is to have a formal model to study the exact computational complexity of AQL in the future. The core of this framework are so-called *spanners* which are used to map an input string of length n to a set of intervals between 1 and $n+1$. When defining spanners in the framework, the authors distinguish between two kinds of representations: First, *primitive* spanner representations, i.e., *regular-expression formulas*, *variable-stack automata*, and *variable-set automata*. Second, representations via *spanner algebras* where spanners are defined by applying algebraic operations to primitive spanner representations.

In the following, our focus is on the primitive spanner representations that involve automata. Therefore, we briefly review variable-stack and variable-set automata using notions from [FKRV13] in Section 6.6.1. Afterwards, we show how to solve the IncrementalEnumeration problem for these automata. For our solution we use k-εNSFAs which are k-NFSAs that allow ε-transitions. We formally define k-εNSFAs in Section 6.6.2 and show how to adapt the algorithm from Section 6.3 for these automata. In Section 6.6.3 we finally illustrate encodings for variable-stack and variable-set automata into k-NSFAs with ε-transitions such that the algorithm from Section 6.6.2 solves IncrementalEnumeration for the encoded automata.

6.6.1 Variable-Stack and Variable-Set Automata

Given a word w of length n, the output of a variable-stack or variable-set automaton is a set of relations where each relation contains a set of intervals between 1 and $n+1$. To formally define variable-stack or variable-set automata and their output, we need the following notation from [FKRV13].

6.6.1 Definition. Let Σ be an alphabet and $w = a_1 \cdots a_n$ be a word over Σ. Then,

- a *span* on w is a tuple $[i, j\rangle$ where $1 \leq i \leq j \leq n+1$, which is associated with the subword $a_i \cdots a_{j-1}$,
- a *span relation* over w is a set of spans on w, and
- a *spanner* is an operator that transforms w into a set of span relations over w.

For example, let $w = abcd$ then $[1, 5\rangle = w$, $[2, 3\rangle = b$, and $[2, 2\rangle = \varepsilon$. The set $\{[1, 5\rangle, [2, 3\rangle, [2, 2\rangle\}$ is a span relation over w.

To represent spanners, *span variables* are used to identify the start and end of a span. We informally illustrate span variables on the example of *regular-expression formulas* [FKRV13], where expressions from the class RE are extended by an additional operator of the form $x\{r\}$. Here, x is a span variable and r is an arbitrary regular-expression formula. The semantics of regular-expression formulas associates to a word in the language of the formula the subword that is matched by r to the span represented by x. For example, consider the regular-expression formula $r_x = (aa)^* x\{a\}(aa)^*$ and the word $w = aaaaa$. Observe that $w \in L((aa)^* a (aa)^*)$ and that x can be associated to the spans $[1, 2\rangle$, $[3, 4\rangle$, and $[5, 6\rangle$ depending on how the word is matched to the regular expression. The output on w of the spanner represented by r_x is $\{\{[1, 2\rangle\}, \{[3, 4\rangle\}, \{[5, 6\rangle\}\}$.

Next, we review two formal definitions of primitive spanner representations that involve automata and were introduced in [FKRV13]. In these automata, the span variables are used together with a special *push symbol* $\vdash$ and a special *pop symbol* $\dashv$.

6.6.2 Definition. A *variable-stack automaton* V is a tuple $(Q, \Sigma, X, \delta, o, f)$ where Q is a finite set of states, Σ is a finite alphabet, X is a finite set of span variables, δ is the transition function with signature

$$Q \times (\Sigma \cup \{x\vdash \mid x \in X\} \cup \{\varepsilon, \dashv\}) \to \mathcal{P}(Q),$$

$o \in Q$ is the initial state, and $f \in Q$ is the accepting state.

The semantics of a variable-stack automaton V are defined as follows. For transitions of the form (q, a, q') and (q, ε, q'), the automaton V behaves analogously to nondeterministic finite automata (with ε-transitions). For the remaining transitions V maintains a *stack* u. A span variable x is pushed on u when a transition $(q, x\vdash, q')$ is read and the topmost variable is popped from u when a transition $(q, \dashv, q')$ is read. In addition, it is not allowed to push a span variable twice during the same computation of V. Therefore, V maintains a set $Y \subseteq X$ of *available* span variables, i.e., the set of variables that can still be pushed on the stack u. Thus, the stack u contains only symbols from X and u contains no variable twice. In the following, we represent u as a word over X and denote an arbitrary stack under these constraints by $u \downarrow X$. Notice that $|u| \leq |X|$ for every $u \downarrow X$. A *configuration of* V is a tuple (q_i, u_i, Y_i, j) where $q_i \in Q$ is the current state, $u_i \downarrow X$ is the current stack, $Y_i \subseteq X$ is the set of available span variables, and $j \in \{0, \ldots, n\}$ indicates that j symbols from the input word $w = a_1 \cdots a_n$ were already processed. A *run*

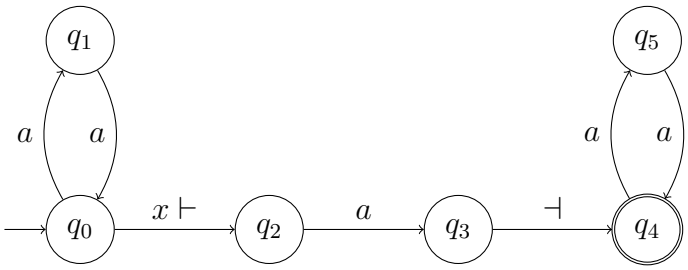


Figure 6.5: An example variable-stack automaton V_r.

of V on a word w is a sequence of configurations $c_0 \cdots c_m$ where c_0 is the *initial configuration* $(o, \varepsilon, X, 0)$ and, for all $i \in \{1, \ldots, m\}$, one of the following holds for $c_{i-1} = (q_{i-1}, u_{i-1}, Y_{i-1}, j)$ and $c_i = (q_i, u_i, Y_i, k)$:

- $u_i = u_{i-1}$, $Y_i = Y_{i-1}$, $k = j + 1$, and $(q_{i-1}, a, q_i) \in \delta$,
- $u_i = u_{i-1}$, $Y_i = Y_{i-1}$, $k = j$, and $(q_{i-1}, \varepsilon, q_i) \in \delta$,
- $u_i = u_{i-1} \cdot x$, $Y_i = Y_{i-1} \backslash \{x\}$, $k = j$, and $(q_{i-1}, x\vdash, q_i) \in \delta$ for some $x \in X$, or
- $u_i \cdot x = u_{i-1}$, $Y_i = Y_{i-1}$, $k = j$, and $(q_{i-1}, \dashv, q_i) \in \delta$ for some $x \in X$.

A run of V on w is *accepting* if and only if $c_m = (f, \varepsilon, \emptyset, n)$. For an accepting run r of V on w we define, for every variable $x \in X$, the span $\rho(x) = [i, j\rangle$ such that the variable x was pushed after a_{i-1} and before a_i was processed and x was popped after a_j and before a_{j+1} was processed. Furthermore, we define $\mu_r = \{\rho(x) \mid x \in X\}$. Then, the automaton V computes the output

$$V(w) = \{\mu_r \mid r \text{ is an accepting run of } V\}.$$

In Figure 6.5 we illustrate a variable-stack automaton V_r, which is semantically equivalent to the aforementioned regular-expression formula $r_x = (aa)^* x\{a\}(aa)^*$. Next, we review the definition of variable-set automata from [FKRV13]. These automata behave very similar to variable-stack automata, except that they maintain a set of active span variables instead of a stack. Therefore, the pop transitions in a variable-set automaton indicate which variable is popped, i.e., they bear a label $\dashv x$ for some span variable x.

6.6.3 Definition. A *variable-set automaton* V is a tuple $(Q, \Sigma, X, \delta, o, f)$ where Q is a finite set of states, Σ is a finite alphabet, X is a finite set of span variables, δ is the transition function with signature

$$Q \times (\Sigma \cup \{\varepsilon\} \cup \{x\vdash \mid x \in X\} \cup \{\dashv x \mid x \in X\}) \to \mathcal{P}(Q),$$

$o \in Q$ is the initial state, and $f \in Q$ is the accepting state.

The semantics of a variable-set automaton V are defined as follows. A *configuration of* V is a tuple (q_i, U_i, Y_i, j) where $q_i \in Q$ is the current state, $U_i \subseteq X$ is the set of *active* span variables, $Y \subseteq X$ is the set of *available* span variables, and $j \in \{0, \ldots, n\}$ indicates that j symbols from the input word were already processed. A *run of* V *on a word* $w = a_1 \cdots a_n$ is defined as a sequence of configurations $c_0 \cdots c_m$ where c_0 is the *initial configuration* $(o, \emptyset, X, 0)$ and, for all $i \in \{1, \ldots, m\}$, one of the following holds for $c_{i-1} = (q_{i-1}, U_{i-1}, Y_{i-1}, j)$ and $c_i = (q_i, U_i, Y_i, k)$:

- $U_i = U_{i-1}$, $Y_i = Y_{i-1}$, $k = j + 1$, and $(q_{i-1}, a, q_i) \in \delta$,
- $U_i = U_{i-1}$, $Y_i = Y_{i-1}$, $k = j$, and $(q_{i-1}, \varepsilon, q_i) \in \delta$,
- $U_i = U_{i-1} \cup x$, $Y_i = Y_{i-1} \backslash \{x\}$, $k = j$, and $(q_{i-1}, x \vdash, q_i) \in \delta$ for some $x \in X$, or
- $U_i = U_{i-1} \backslash \{x\}$, $Y_i = Y_{i-1}$, $k = j$, and $(q_{i-1}, \dashv x, q_i) \in \delta$ for some $x \in X$.

A run of V on w is *accepting* if and only if $c_m = (f, \emptyset, \emptyset, n)$. For an accepting run r of V on w we define, for every variable $x \in X$, the span $\rho(x) = [i, j\rangle$ such that the variable x was pushed after a_{i-1} and before a_i was processed and x was popped after a_j and before a_{j+1} was processed. Furthermore, we define $\mu_r = \{\rho(x) \mid x \in X\}$. Then, the automaton V computes the output

$$V(w) = \{\mu_r \mid r \text{ is an accepting run of } V\}.$$

6.6.2 Incremental Enumeration for k-εNFSAs

To solve IncrementalEnumeration for variable-stack and variable-set automata, we show first how to adapt the algorithm from Section 6.3.3 such that it can solve IncrementalEnumeration for k-NFSAs with ε-transitions. Therefore, we briefly review the definition of NFAs with ε-transitions in the following.

6.6.4 Definition. A *(nondeterministic) finite automaton with ε-transitions* (or *εNFA*) N is a tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is a finite alphabet, δ is the transition function with signature $Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states.

The semantics of an εNFA N are defined as usual, i.e., whenever $q_2 \in \delta(q_1, \varepsilon)$ we know that, if N is in state q_1 then reading ε takes N to state q_2. A *run of* N *on a word* $w = a_1 \cdots a_n$ is a sequence $r = (q_0, 1) \cdots (q_m, n + 1)$ such that, for every $i \in \{1, \ldots, m\}$,

- either (q_{i-1}, j) and $(q_i, j + 1)$ occurs in r and $q_i \in \delta(q_{i-1}, a_j)$,
- or (q_{i-1}, j) and (q_i, j) occurs in r and $q_i \in \delta(q_{i-1}, \varepsilon)$.

For such a run r we say that r *visits position* i *in state* q_j for $j \in \{0, \ldots, m\}$ if the pair (q_j, i) appears in r. Notice that, by this definition, r visits position i before the symbol a_i is processed, i.e., r visits positions 1 to $n + 1$. Since variable-set and variable-stack automata output spans between 1 and $n + 1$, the above definition is

accordant to these automata and will facilitate our encoding in the next section. If $q_m \in F$ then the run is *accepting*. We say that a run r of N on w is *partial* if r fulfills all conditions of a run except that r does not have to start with the initial state.

6.6.5 Definition. For $k \in \mathbb{N}$ a *k-ary nondeterministic finite selecting automaton with ε-transitions* (or *k-εNFSA*) M is a pair (N, S) where N is an εNFA over Σ with states Q and $S \subseteq Q^k$ is a set of *selecting tuples*.

When M reads a word w of length n, it computes a set of tuples in $\mathrm{Nodes}(w)^k$. More precisely, we define

$$M(w) = \{(v_1, \ldots, v_k) \mid \text{ there is an accepting run } r \text{ of } N \text{ on } w \text{ and a tuple } (p_1, \ldots, p_k) \in S \text{ such that } r \text{ visits } v_\ell \text{ in } p_\ell \text{ for every } \ell \in \{1, \ldots, k\}\}.$$

We can solve the IncrementalEnumeration problem for k-εNFSAs by a slight variation of the algorithm from Section 6.3. We fix the following notation for the remainder of this section. By $M = ((Q, \Sigma, \delta, q_0, F), S)$ we denote a k-εNFSA and by $w = a_1 \cdots a_n \in \Sigma^*$ the input word. By Q_S we denote the set of all states that appear in some selecting tuple, i.e., $Q_S = \cup_{s \in S} \mathrm{set}(s)$. We make the following two changes in the auxiliary data structure (see Section 6.3.2).

First, we build the auxiliary tree $\mathrm{N}_w^{\mathrm{aux}}$ over $n+1$ leaves because in a spanner we can select the positions 1 to $n+1$. For simplicity, we assume that $n+1$ is a power of two in the following.

Second, we adapt the initialization of the extended transition relation T^+. In particular, we compute the relation $\mathrm{T}^+(v_{xx})$ for every $x \in \{1, \ldots, n+1\}$ as follows:

- if $x \neq n+1$ then $\mathrm{T}^+(v_{xx}) = \{(q_1, q_m, I) \mid$ there exists a partial run $(q_1, x) \cdots (q_{m-1}, x)(q_m, x+1)$ with $m \geq 2$ and $I = \{q_2, \ldots, q_m\} \cap Q_S\}$,
- otherwise, $\mathrm{T}^+(v_{xx}) = \{(q_1, q_m, I) \mid$ there exists a partial run $(q_1, x) \cdots (q_m, x)$ with $m \geq 2$ and $I = \{q_2, \ldots, q_m\} \cap Q_S\}$.

We compute the relation $\mathrm{T}^+(v_{xy})$ for each inner node v_{xy} of $\mathrm{N}_w^{\mathrm{aux}}$ bottom-up exactly as we compute $\mathrm{T}^+(v_{xy})$ for k-NFSAs and words (using the operation $\bowtie^+$ from Definition 6.3.5). After the relation T^+ is computed for the k-εNFSA M, the following holds for every $v_{xy} \in \mathrm{N}_w^{\mathrm{aux}}$:

- for $y \neq n+1$ the relation $\mathrm{T}^+(v_{xy})$ contains all tuples $(q_1, q_2, I) \in (Q^2 \times 2^{Q_S})$ for which there exist a selecting tuple $s \in S$ and partial run $r = (q_1, x) \cdots (q, y)(q_2, y+1)$ on $w[x..y]$ such that $I = \mathrm{set}(r) \cap \mathrm{set}(s)$,
- otherwise, the relation $\mathrm{T}^+(v_{xy})$ contains all tuples $(q_1, q_2, I) \in (Q^2 \times 2^{Q_S})$ for which there exist a selecting tuple $s \in S$ and partial run $r = (q_1, x) \cdots (q_2, y)$ on $w[x..y-1]$ such that $I = \mathrm{set}(r) \cap \mathrm{set}(s)$.

Thus, $M(w) \neq \emptyset$ if and only if $(q_0, q_F, \mathrm{set}(s)) \in \mathrm{T}^+(v_{1(n+1)})$ for $q_F \in F$ and some $s \in S$. Moreover, the tree $\mathrm{N}_w^{\mathrm{aux}}$ is of size $O(|Q|^2 \cdot |S| \cdot 2^k \cdot n)$. The computation of $\mathrm{N}_w^{\mathrm{aux}}$ needs time $O(|Q|^4 \cdot |S|^2 \cdot 2^k \cdot n)$ since we have to compute the transitive closure over all ε-transitions in M to compute the relation T^+ for the leaf nodes in $\mathrm{N}_w^{\mathrm{aux}}$. The algorithm for the enumeration of the result set $M(w)$ is completely analogous to the enumeration procedure for k-NFSAs without ε-transitions (see Sections 6.3.3 to 6.3.5), which leads to the following result.

6.6.6 Theorem. IncrementalEnumeration *for a k-εNSFA M and a word w with $|w| = n$ can be solved with an auxiliary data structure of size $O(|Q|^2 \cdot |S| \cdot 2^k \cdot n)$ which can be computed in time $O(|Q|^4 \cdot |S|^2 \cdot 2^k \cdot n)$, maintained in time $O(|Q|^3 \cdot |S|^2 \cdot 2^k \cdot \log n)$ per update, and guarantees $O(|S| \cdot k! + |Q|^3 \cdot |S|^2 \cdot 2^k \cdot k \log n)$ enumeration delay between answers.*

6.6.3 Incremental Enumeration for Spanners

We now show how to apply the algorithm from Section 6.6.2 to solve IncrementalEnumeration for variable-stack and variable-set automata. To this end, we define an encoding of variable-stack and variable-set automata with k span variables into $(2k)$-εNFSAs. In the following, we illustrate the encoding for variable-stack automata and argue why it is correct. The encoding for variable-set automata is then almost the same.

Let $V = (Q_V, \Sigma, X, \delta_V, o, f)$ be a variable-stack automaton with a set of span variables $X = \{x_1, \ldots, x_k\}$. We simulate a run of the variable-stack automaton by encoding the current content of the stack u and the current set of available span variables Y in the states of a $(2k)$-εNFSA, i.e., the $(2k)$-εNFSA contains a state q_{uY} for every $q \in Q_V$, $u \downarrow X$, and $Y \subseteq X$. (Remember that u and Y have size at most k.) More intuitively, the $(2k)$-εNFSA contains a copy of the state set of the variable-stack automaton for every configuration of u and Y. This includes that, for every state q_{uY}, the stack u contains no span variable twice and it always holds that $u \cap Y = \emptyset$.

According to the semantics of a variable-stack automaton, u and Y remain unchanged in the next configuration when reading a Σ- or ε-labeled transition in V. Thus, the $(2k)$-εNFSA contains a transition (q_{uY}, σ, p_{uY}) for every $u \downarrow X, Y \subseteq X$, and transition $(q, \sigma, p) \in \delta_V$ with $\sigma \in \Sigma \cup \{\varepsilon\}$. We also encode $x\vdash$- and $\dashv$-labeled transitions of V by ε-transitions in the $(2k)$-εNFSA. However, since a variable-stack automaton defines the start (or end, respectively) of a span at the moment when such a transition is read, we have to ensure that the $(2k)$-εNFSA reaches a selecting state at the target state of an encoded $x\vdash$- or $\dashv$-labeled transition. Therefore, we add states $q_{uY}^{x\vdash}$ and $q_{uY}^{\dashv}$ for every state $q \in Q_V$, stack $u \downarrow X$, set $Y \subseteq X$, and variable $x \in X$. Moreover, we add a transition $(q_{uY}, \varepsilon, p_{u \cdot xY \setminus \{x\}}^{x\vdash})$ to the $(2k)$-εNFSA for every transition $(q, x\vdash, p) \in \delta_V$ and we add a transition $(q_{u \cdot xY}, \varepsilon, p_{uY}^{\dashv})$ for every transition $(q, \dashv, p) \in \delta_V$. The set of $q_{uY}^{x\vdash}$- and $q_{uY}^{\dashv}$-outgoing transitions are equal to the set of q_{uY}-outgoing transitions.

Finally, the $(2k)$-εNFSA contains a selecting tuple $(*^{x_1\vdash}_{u_1Y_1}, *^{\dashv}_{u_1Y_1}, \ldots, *^{x_k\vdash}_{u_kY_k}, *^{\dashv}_{u_kY_k})$ where $*$ are arbitrary (not necessary equal) states $q \in Q$, $u_1, \ldots, u_k \downarrow X$ and $Y_1, \ldots, Y_k \subseteq X$. A formal definition of the encoding is given in the following.

6.6.7 Definition. Let $V = (Q_V, \Sigma, X, \delta_V, o, f)$ be a variable-stack automaton where $X = \{x_1, \ldots, x_k\}$. The $(2k)$-εNFSA M_V is defined as $((Q_M, \Sigma, \delta_M, o_{\varepsilon X}, F_M), S_M)$ where

- Q_M contains states q_{uY}, $q^{x\vdash}_{uY}$, and $q^{\dashv}_{uY}$ for every state $q \in Q_V$, stack $u \downarrow X$, set $Y \subseteq X$ with $u \cap Y = \emptyset$, and variable $x \in X$,
- δ_M contains, for every $x \in X$, $u \downarrow X, Y \subseteq X$, and $\sigma \in \Sigma \cup \{\varepsilon\}$, the transitions
  - (q_{uY}, σ, p_{uY}), $(q^{x\vdash}_{uY}, \sigma, p_{uY})$, and $(q^{\dashv}_{uY}, \sigma, p_{uY})$ for every transition $(q, \sigma, p) \in \delta_V$,
  - $(q_{uY}, \varepsilon, p^{y\vdash}_{u\cdot yY\setminus\{y\}})$, $(q^{x\vdash}_{uY}, \varepsilon, p^{y\vdash}_{u\cdot yY\setminus\{y\}})$, and $(q^{\dashv}_{uY}, \varepsilon, p^{y\vdash}_{u\cdot yY\setminus\{y\}})$ for every transition $(q, y\vdash, p) \in \delta_V$ with $y \in Y$, and
  - $(q_{u\cdot xY}, \varepsilon, p^{\dashv}_{uY})$, $(q^{x\vdash}_{u\cdot xY}, \varepsilon, p^{\dashv}_{uY})$, and $(q^{\dashv}_{u\cdot xY}, \varepsilon, p^{\dashv}_{uY})$ for every transition $(q, \dashv, p) \in \delta_V$,
- F_M contains the state $f_{\varepsilon\emptyset}$ and, for every $x \in X$, states $f^{x\vdash}_{\varepsilon\emptyset}$ and $f^{\dashv}_{\varepsilon\emptyset}$, and
- $S_M = \{(*^{x_1\vdash}_{u_1Y_1}, *^{\dashv}_{u_1Y_1}, \ldots, *^{x_k\vdash}_{u_kY_k}, *^{\dashv}_{u_kY_k})\}$ where $*$ are arbitrary (not necessary equal) states $q \in Q$, $u_1, \ldots, u_k \downarrow X$ and $Y_1, \ldots, Y_k \subseteq X$.

Using this encoding, we get the output for V by interpreting the selecting tuples as a tuple of pairs, i.e., for a word $w \in \Sigma^*$,

$$V(w) = \{\{[i_1, i_2\rangle, \ldots, [i_{2k-1}, i_{2k}\rangle\} \mid (i_1, i_2, \ldots, i_{2k-1}, i_{2k}) \in M_V(w)\}.$$

In this way, we can solve IncrementalEnumeration for a variable-stack automaton and an input word of length n with preprocessing time $O(n)$, enumeration delay $O(\log n)$, and update time $O(\log n)$. However, the encoding of Definition 6.6.7 constructs a selecting automaton that can be exponential in k in the worst case. The result for IncrementalEnumeration also holds for variable-set automata via a slight but straightforward variation of the above encoding.

6.7 Incremental Enumeration: Logic, XPath, and Future Work

It is well-known that there exists a direct connection between run-based node-selecting automata and MSO-queries (see, e.g., [NPTT05, TW68]). As such we study the incremental enumeration problem for MSO queries with free node variables, over words and trees. However, it should be kept in mind that MSO queries can be non-elementary smaller than their equivalent nondeterministic node-selecting finite (tree) automaton. Therefore, our enumeration algorithm is non-elementary in

terms of the MSO formula, which cannot be avoided unless P = NP [FG04]. Although, this seems not very practical in the first place, notice that the arity of the query is usually very small in practical applications. For example, XPath queries can be modeled using only binary queries (i.e., $k = 2$).

Moreover, the incremental evaluation problem was already investigated in the context of XML [BML$^+$04] and XPath queries [BGM10]. In [BGM10] Björklund et al. presented an algorithm for maintaining whether updates preserve a property specified by an XPath query. This algorithm also uses Patnaik and Immerman's divide-and-conquer approach. However, notice that the studied XPath dialects are less expressive than tree automata, they may be exponentially more succinct, and they only consider Boolean queries such that they are not concerned with efficiently enumerating answers. Regarding core XPath queries or variants of regular XPath (see, e.g., [BK09]) and the IncrementalEnumeration problem, the result from Theorem 6.4.20 provides an $O(|Q|^2 \cdot |t|)$ upper bound on the auxiliary data structure and $O(|Q|^3 \cdot \log^2 |t|)$ enumeration delay between answers.

Towards future work, it would be interesting to see if the techniques presented in this chapter can be generalized towards graphs with bounded treewidth, using tree decompositions and the generalization in Bagan [Bag06]. A straightforward generalization of our algorithm would only be able to deal with relabelings since node insertions and deletions can have drastic impact on tree decompositions. Furthermore, a single node relabel in the graph can induce $m > 1$ relabels in the tree decomposition which would influence complexity.

We also believe that our method is promising to efficiently compute the *difference between answers*. In this case one wants to run an algorithm that computes, after an update occurred on the word or the tree, which tuples no longer satisfy the query and which ones are new in the answer set.

It could also be interesting to investigate whether we can efficiently maintain the number of answers to a query (under set or multiset semantics). Notice that the number of times that a tuple $(i_1, \ldots, i_k)$ is in the answer under multiset semantics is simply the number of tuples in $G(i_k)$. Computing the number of answers efficiently is important for deciding whether a constant-delay algorithm with linear time preprocessing would be able to output the whole output faster than our logarithmic-delay algorithm which would not require preprocessing after an update. Roughly, when the output of a query contains at most $O(n/\log n)$ outputs, the logarithmic-delay algorithm will finish more quickly than a constant-delay procedure with linear preprocessing. Moreover, the logarithmic-delay algorithm produces the first answers more quickly. Estimating the number of answers to a query can therefore help to decide which kind of procedure is desirable.

7

Conclusions

We examined the computational complexity of regular languages for data processing on the Web. One of the most used concepts for this purpose are regular expressions that are enhanced by additional operators or have to fulfill special semantical constraints to make them fit for the distinct requirements in various practical applications. In particular, we were interested in the complexity of variants of regular expressions that are used in the context of XML schema and RDF query languages. For the considered variants of regular expressions we investigated whether it is reasonable to use them in practice or whether they prevent an efficient implementation.

Regarding the examined semantical constraints on regular expressions our picture is rather negative. The first constraint that we examined was the determinism constraint for regular expressions used in XML schema languages. Although determinism is often used to make certain problems computationally easier, the use of deterministic regular expressions has some severe drawbacks. For example, not every regular language can be written as a DRE such that not every regular language can be used in a schema that requires deterministic expressions.

In **Chapter 4**, we examined the computational complexity of the problem to decide whether a regular language can be expressed by a deterministic regular expression. This problem is called the DRE-Definability problem. Our main result is a PSPACE algorithm that adapts an algorithm in [BKW98] and decides whether the language of an NFA can be written as a deterministic regular expression. Using the known PSPACE lower bound for this problem from Bex et al. [BGMN09], it followed that DRE-Definability(NFA) is PSPACE-complete. We also examined the complexity of this problem for several variants depending on the particular input. To this end, we proved that DRE-Definability(RE) is PSPACE-complete, DRE-Definability(RE(#)) is EXPSPACE-complete, and DRE-Definability(minDFA) is NLOGSPACE-hard.

Afterwards we studied the relative descriptional complexity of deterministic regular expressions. In summary our results show that, although DRE-definable lan-

guages are a proper subset of regular languages, the descriptional complexity of DREs is not better than compared to minimal DFAs or arbitrary REs. More precisely, we proved that the representation of an arbitrary regular language as a deterministic expression can be exponentially larger than the representation as a regular expression and a minimal DFA in general. We remark that, this result was already shown in the master thesis of the author [Los10, LMN12]. Compared to the result in [Los10, LMN12] we provided here a proof that is shorter, less complex, and even holds for finite languages. In addition, we proved that by applying a boolean operation on two deterministic regular expressions an exponential blow-up for the resulting DRE cannot be avoided in general.

Second, we studied the complexity of two semantical constraints that were introduced for regular expressions (i.e., property paths) in SPARQL queries: the simple walk requirement and the path counting requirement. In **Chapter 5**, we proved that, independently from each other, both constraints prevent an efficient implementation of SPARQL query evaluation. In more detail, the evaluation of SPARQL queries is NP-complete even for very restricted regular expressions and also under data complexity. Query evaluation under the path counting requirement is #P-complete even for very restricted expressions and under regular path semantics (which does not require the evaluation on simple walks).

Opposed to the studied semantical constraints, we also examined the evaluation of regular expressions with additional operators (including counting operators). Here, the studied problems were motivated by the set of allowed regular expressions in SPARQL queries. We showed that the additional operators in these expressions do not pose a critical problem for efficient query evaluation. More precisely, we proved that regular expressions with counting operators can be evaluated over graphs in polynomial time. We also observed that the restricted negation operator is tractable in SPARQL queries, though this operator should not be relaxed since query evaluation of expressions with full negation over graphs becomes intractable immediately. A result that is not mainly motivated by SPARQL query evaluation but might be interesting in another context, is that the membership problem for regular expressions with counting operators and full negation is in P.

In **Chapter 6**, we investigated enumeration algorithms that are sensitive to dynamic data. In more detail, we constructed algorithms for the evaluation of k-ary nondeterministic finite selecting (tree) automata which are evaluated over words and trees, respectively. For the IncrementalEnumeration problem for words of size n, we presented an algorithm that needs $O(n)$ preprocessing time, outputs all answers with $O(\log n)$ enumeration delay between two answers, and processes data updates in $O(\log n)$ time. The algorithm first builds an auxiliary tree over the input word in the preprocessing. It traverses the auxiliary tree from left to right to compute the answers in a kind of lexicographic order. For the IncrementalEnumeration problem for trees of size n, we constructed an algorithm that solves the problem with $O(n)$ preprocessing time, $O(\log^2 n)$ enumeration delay between answers, and $O(\log^2 n)$ time per update. The algorithm combines the algorithm for words and a technique of Balmin et al. [BPV04] that shows how to maintain membership for a tree in the language of a tree automaton under updates.

Finally, we illustrated how our algorithms can be used to solve IncrementalEnumeration for queries under multiset semantics. We also applied our results for an application from the field of information extraction that uses variable-stack and variable-set automata.

Open problems and further questions in the future Regarding deterministic regular expressions the following question might be the most interesting one that is still open:

> Is there a regular language L such that a minimal DRE for L is double-exponentially larger than a minimal RE for L?

Although we still lack the techniques to answer this question, our results in **Chapter 4** give some new insights into the problem. More precisely, we are able to specify certain properties of the structure of the minimal NFA and DFA for such a DRE-definable language L, using the result of the polynomial upper bound on the recursion depth of the BKW-Algorithm.

Another interesting open question regarding the class of DRE-definable languages is the following:

> Is there a DRE-definable language L such that a minimal RE for L is exponentially larger than the minimal DFA for L?

As far as we know all existing techniques for proving lower bounds on the relative size of arbitrary regular expressions cannot be applied for DRE-definable languages. More precisely, these techniques cannot be applied because the desired preconditions for the used languages directly imply that the language is not DRE-definable (see, e.g., the techniques used in [EZ76, GH08, GJ08]). Therefore, we believe that new techniques have to be developed to answer this question.

A detailed discussion of open problems regarding SPARQL query evaluation can be found at the end of **Chapter 5**. For the future of SPARQL the most challenging problem to solve might be to find expressive and tractable semantics that can be realized within the specification. Although we see it very positive that first changes were introduced to the SPARQL specification, the withdrawal of counting operators in property paths was not necessary. It would be interesting to see whether stronger arguments on the efficiency of SPARQL query evaluation with regular expressions (including counting operators) can reverse this change.

The presented results for IncrementalEnumeration were, as far as we know, the first to formally study this problem for queries with arbitrary arity. We therefore see our results as a proof of concept. Moreover, we believe that our results leave some space for further research on this topic. Ideas for future work regarding the IncrementalEnumeration problem were discussed in full details at the end of **Chapter 6**. In the future it would be interesting as well as challenging to prove lower bounds on these kind of algorithms or to extend the results to more general structures (e.g., structures of bounded treewidth). We also explained why it would be useful to compute the difference between two consecutive query results or to calculate the number of answers to a query.

Bibliography

[AB09] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[ABE09] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 7(2):57–73, 2009.

[ABLM14] M. Arenas, P. Barceló, L. Libkin, and F. Murlak. *Foundations of Data Exchange*. Cambridge University Press, 2014.

[ABS99] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[ACP12] M. Arenas, S. Conca, and J. Pérez. Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *Proceedings of the 21st World Wide Web Conference 2012 (WWW)*, pages 629–638. ACM, 2012.

[AI00] N. Alechina and N. Immerman. Reachability logic: An efficient fragment of transitive closure logic. *Logic Journal of the Interest Group in Pure and Applied Logics (IGPL)*, 8(3):325–337, 2000.

[AJ93] C. Álvarez and B. Jenner. A very hard log-space counting class. *Theoretical Computer Science (TCS)*, 107(1):3–30, 1993.

[AP11] M. Arenas and J. Pérez. Querying semantic web data with SPARQL. In *Proceedings of the 30th Symposium on Principles of Database Systems (PODS)*, pages 305–316. ACM, 2011.

[Apa11] Apache Software Foundation. Jena semantic web framework. http://jena.apache.org/, 2011.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[AV99] S. Abiteboul and V. Vianu. Regular path queries with constraints. *Journal of Computer and System Sciences (JCSS)*, 58(3):428–452, 1999.

[Bag06] G. Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *Proceedings of the 15th Conference on Computer Science Logic (CSL)*, pages 167–181. Springer, 2006.

[BBG13] G. Bagan, A. Bonifati, and B. Groz. A trichotomy for regular simple path queries on graphs. In *Proceedings of the 32nd Symposium on Principles of Database Systems (PODS)*, pages 261–272. ACM, 2013.

[BDHS96] P. Buneman, S. B. Davidson, G. G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 505–516. ACM, 1996.

[BEGO71] R. Book, S. Even, S. Greibach, and G. Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, 20(2):149–153, 1971.

[Ber73] C. Berge. *Graphs and Hypergraphs*. North-Holland Publishing Company, 1973.

[BF11] M. Bojanczyk and D. Figueira. Efficient evaluation for a temporal logic on changing XML documents. In *Proceedings of the 30th Symposium on Principles of Database Systems (PODS)*, pages 259–270. ACM, 2011.

[BGM10] H. Björklund, W. Gelade, and W. Martens. Incremental XPath evaluation. *ACM Transactions on Database Systems (TODS)*, 35(4):29:1–29:43, 2010.

[BGMN09] G. J. Bex, W. Gelade, W. Martens, and F. Neven. Simplifying XML Schema: effortless handling of nondeterministic regular expressions. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 731–744. ACM, 2009.

[BGNV10] G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. *ACM Transactions on the Web*, 4(4):14:1–14:32, 2010.

[BHBL09] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.

[BK93] A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science (TCS)*, 120(2):197–213, 1993.

[BK09] M. Benedikt and C. Koch. XPath leashed. *ACM Computing Surveys (CSUR)*, 41(1):3:1–3:54, 2009.

[BKL12] Z. Bao, B. Kimelfeld, and Y. Li. Automatic suggestion of query-rewrite rules for enterprise search. In *Proceedings of the 35th International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 591–600. ACM, 2012.

[BKW92] A. Brüggemann-Klein and D. Wood. Deterministic regular languages. In *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 173–184. Springer, 1992.

[BKW98] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998.

[BL09] T. Berners-Lee. Linked Data. http://www.w3.org/DesignIssues/LinkedData.html, June 2009.

[BLFM05] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. Technical report, The Internet Society, January 2005.

[BML+04] D. Barbosa, A.O. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. Efficient incremental validation of XML documents. In *Proceedings of the 20th International Conference on Data Engineering (ICDE)*, pages 671–682, 2004.

[BNdB04] G. J. Bex, F. Neven, and J. Van den Bussche. DTDs versus XML Schema: A practical study. In *Proceedings of the 7th International Workshop on the Web and Databases (WebDB)*, pages 79–84, 2004.

[BNSV10] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren. Inference of concise regular expressions and DTDs. *ACM Transactions on Database Systems (TODS)*, 35(2):11:1–11:47, 2010.

[BNV07] G. J. Bex, F. Neven, and S. Vansummeren. Inferring XML Schema Definitions from XML data. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 998–1009. ACM, 2007.

[BPSM+08] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language XML 1.0 (fifth edition). Technical report, World Wide Web Consortium (W3C), November 2008. W3C Recommendation at http://www.w3.org/TR/xml/.

[BPV04] A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of XML documents. *ACM Transactions on Database Systems (TODS)*, 29(4):710–751, 2004.

[Brz64] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[CC08] H. Chen and L. Chen. Inclusion test algorithms for one-unambiguous regular expressions. In *Proceedings of the 5th International Colloquium on Theoretical Aspects of Computing (ICTAC)*, pages 96–110. Springer, 2008.

[CCSY01] C. Câmpeanu, K. Culik, K. Salomaa, and S. Yu. State complexity of basic operations on finite languages. In *Proceedings of the 4th International Workshop on Implementing Automata (WIA)*, pages 60–70. Springer, 2001.

[CDLM13] W. Czerwinski, C. David, K. Losemann, and W. Martens. Deciding definability by deterministic regular expressions. In *Proceedings of the 16th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, pages 289–304. Springer, 2013.

[CGLV00a] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *Principles of Knowledge Representation and Reasoning (KR)*, pages 176–185. Morgan Kaufmann, 2000.

[CGLV00b] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. View-based query processing for regular path queries with inverse. In *Proceedings of the 19th Symposium on Principles of Database Systems (PODS)*, pages 58–66. ACM, 2000.

[CGLV02] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Rewriting of regular expressions and regular path queries. *Journal of Computer and System Sciences (JCSS)*, 64(3):443–465, 2002.

[CGS09a] D. Colazzo, G. Ghelli, and C. Sartiani. Efficient asymmetric inclusion between regular expression types. In *Proceedings of the 12th International Conference on Database Theory (ICDT)*, pages 174–182. ACM, 2009.

[CGS09b] D. Colazzo, G. Ghelli, and C. Sartiani. Efficient inclusion for a class of XML types with interleaving and counting. *Information Systems (IS)*, 34(7):643–656, 2009.

[CHM11] P. Caron, Y. Han, and L. Mignot. Generalized one-unambiguity. In *Proceedings of the 15th International Conference on Developments in Language Theory (DLT)*, pages 129–140. Springer, 2011.

[Cho02] B. Choi. What are real DTDs like? In *Proceedings of the 5th International Workshop on the Web and Databases (WebDB)*, pages 43–48, 2002.

[CKL+10] L. Chiticariu, R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, and S. Vaithyanathan. SystemT: An algebraic approach to declarative information extraction. In *Proceedings of the 48th Annual Meeting of*

the Association for Computational Linguistics (ACL), pages 128–137. ACL, 2010.

[CM90] M. P. Consens and A. O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *Proceedings of the 9th Symposium on Principles of Database Systems (PODS)*, pages 404–416. ACM, 1990.

[CM01] J. Clark and M. Murata. Relax NG specification. Technical report, The Organization for the Advancement of Structured Information Standards (OASIS), December 2001. http://www.relaxng.org/spec-20011203.html.

[CMW87] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 323–330. ACM, 1987.

[Col07] T. Colcombet. A combinatorial theorem for trees. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 901–912. Springer, 2007.

[Cor12] O. Corby. Corese 3.0. http://wimmics.inria.fr/corese, March 2012.

[Cou09] B. Courcelle. Linear delay enumeration and monadic second-order logic. *Discrete Applied Mathematics*, 157(12):2675–2700, 2009.

[CS93] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2(2):121–147, 1993.

[CWL14] R. Cyganiak, D. Wood, and M. Lanthaler. Resource Description Framework RDF 1.1. Technical report, World Wide Web Consortium (W3C), February 2014. W3C Recommendation at http://www.w3.org/TR/rdf11-concepts/.

[Die12] R. Diestel. *Graph Theory (fourth edition)*. Springer, 2012.

[DS93] G. Dong and J. Su. First-order incremental evaluation of Datalog queries. In *Proceedings of the 4th International Workshop on Database Programming Languages (DBPL)*, pages 295–308. Springer, 1993.

[DS05] M. Duerst and M. Suignard. Internationalized Resource Identifiers (IRIs). Technical report, The Internet Society, January 2005.

[DS11] A. Durand and Y. Strozecki. Enumeration complexity of logical query problems with second-order variables. In *Proceedings of the 20th Conference on Computer Science Logic (CSL)*, pages 189–202. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011.

[DT01] A. Deutsch and V. Tannen. Optimization properties for classes of conjunctive regular path queries. In *Proceedings of the 8th International Workshop on Database Programming Languages (DBPL)*, pages 21–39. Springer, 2001.

[EKSW04] K. Ellul, B. Krawetz, J. Shallit, and M. Wang. Regular expressions: new results and open problems. *Journal of Automata, Languages and Combinatorics (JALC)*, 9(2-3):233–256, 2004.

[EZ76] A. Ehrenfeucht and H. Zeiger. Complexity measures for regular expressions. *Journal of Computer and System Sciences (JCSS)*, 12(2):134–146, 1976.

[FFLS00] M. F. Fernández, D. Florescu, A. Y. Levy, and D. Suciu. Declarative specification of web sites with STRUDEL. *The VLDB Journal*, 9(1):38–55, 2000.

[FG04] M. Frick and M. Grohe. The complexity of first-order and monadic second-order logic revisited. *Annals of Pure and Applied Logic*, 130(1–3):3–31, 2004.

[FGK03] M. Frick, M. Grohe, and C. Koch. Query evaluation on compressed trees (extended abstract). In *Proceedings of the 18th IEEE Symposium on Logic in Computer Science (LICS)*, page 188. IEEE Computer Society, 2003.

[FKRV13] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Spanners: a formal framework for information extraction. In *Proceedings of the 32nd Symposium on Principles of Database Systems (PODS)*, pages 37–48. ACM, 2013.

[FKRV14] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Cleaning inconsistencies in information extraction via prioritized repairs. In *Proceedings of the 33nd Symposium on Principles of Database Systems (PODS)*, pages 164–175. ACM, 2014.

[FLS98] D. Florescu, A. Y. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *Proceedings of the 17th Symposium on Principles of Database Systems (PODS)*, pages 139–148. ACM, 1998.

[FR13] D. D. Freydenberger and D. Reidenbach. Inferring descriptive generalisations of formal languages. *Journal of Computer and System Sciences (JCSS)*, 79(5):622–639, 2013.

[FW04] D. Fallside and P. Walmsley. XML Schema Part 0: Primer (second edition). Technical report, World Wide Web Consortium (W3C), October 2004. W3C Recommendation at http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.

[Gel10] W. Gelade. Succinctness of regular expressions with interleaving, intersection and counting. *Theoretical Computer Science (TCS)*, 411(31–33):2987–2998, 2010.

[GGM12] W. Gelade, M. Gyssens, and W. Martens. Regular expressions with counting: weak versus strong determinism. *SIAM Journal on Computing (SICOMP)*, 41(1):160–190, 2012.

[GH08] H. Gruber and M. Holzer. Finite automata, digraph connectivity, and regular expression size. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 39–50. Springer, 2008.

[GH09] H. Gruber and M. Holzer. Tight bounds on the descriptional complexity of regular expressions. In *Proceedings of the 13th International Conference on Developments in Language Theory (DLT)*, pages 276–287. Springer, 2009.

[GIM+13] W. Gelade, T. Idziaszek, W. Martens, F. Neven, and J. Paredaens. Simplifying XML schema: Single-type approximations of regular tree languages. *Journal of Computer and System Sciences (JCSS)*, 79(6):910–936, 2013.

[GJ08] H. Gruber and J. Johannsen. Optimal lower bounds on regular expression size using communication complexity. In *Proceedings of the 11th International Conference on Foundations of Software Science and Computational Structures (FOSSACS)*, pages 273–286. Springer, 2008.

[Glu61] V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1–53, 1961.

[GMN09] W. Gelade, W. Martens, and F. Neven. Optimizing schema languages for XML: Numerical constraints and interleaving. *SIAM Journal on Computing (SICOMP)*, 38(5):2021–2043, 2009.

[GMS12] B. Groz, S. Maneth, and S. Staworko. Deterministic regular expressions in linear time. In *Proceedings of the 31st Symposium on Principles of Database Systems (PODS)*, pages 49–60. ACM, 2012.

[GN08] W. Gelade and F. Neven. Succinctness of the complement and intersection of regular expressions. In *Proceedings of the 25th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 325–336. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2008.

[GN12] W. Gelade and F. Neven. Succinctness of the complement and intersection of regular expressions. *ACM Transactions on Computational Logic (TOCL)*, 13(1):4:1–4:19, 2012.

[Gro12] B. Groz. *XML Security Views: Queries, Updates and Schemas.* PhD thesis, Université Lille 1, 2012.

[GS96] R. Grishman and B. Sundheim. Message understanding conference- 6: A brief history. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING)*, pages 466–471, 1996.

[GSMT+12] S. Gao, C. M. Sperberg-McQueen, H. S. Thompson, N. Mendelsohn, D. Beech, and M. Maloney. W3C XML Schema Definition Language (XSD) 1.1 part 1: Structures. Technical report, World Wide Web Consortium (W3C), April 2012. W3C Recommendation at http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/.

[HKK+13] O. Hassanzadeh, A. Kementsietsidis, B. Kimelfeld, R. Krishnamurthy, F. Ozcan, and I. Pandis. Next generation data analytics at IBM research. *Proceedings of the VLDB Endowment*, 6(11):1174–1175, 2013.

[HMU13] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Pearson Education, 2013.

[Hov09] D. Hovland. Regular expressions with numerical constraints and automata with counters. In *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing (ICTAC)*, pages 231–245. Springer, 2009.

[Hov12] D. Hovland. The inclusion problem for regular expressions. *Journal of Computer and System Sciences (JCSS)*, 78(6):1795–1813, 2012.

[HS12] S. Harris and A. Seaborne. SPARQL 1.1 query language. Technical report, World Wide Web Consortium (W3C), January 2012. W3C Working Draft at http://www.w3.org/TR/2012/WD-sparql11-query-20120105/.

[HS13] S. Harris and A. Seaborne. SPARQL 1.1 query language. Technical report, World Wide Web Consortium (W3C), March 2013. W3C Recommendation at http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[HT84] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing (SICOMP)*, 13(2):338–355, 1984.

[Imm88] N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing (SICOMP)*, 17(5):935–938, 1988.

[Jir08] G. Jirásková. On the state complexity of complements, stars, and reversals of regular languages. In *Proceedings of the 12th International Conference on Developments in Language Theory (DLT)*, pages 431–442. Springer, 2008.

[JJS05] J. Jirásek, G. Jirásková, and A. Szabari. State complexity of concatenation and complementation. *International Journal of Foundations of Computer Science*, 16(3):511–529, 2005.

[Jon75] N. D. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Sciences (JCSS)*, 11(1):68–85, 1975.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations (The IBM Research Symposia Series)*, pages 85–103, 1972.

[KFB08] A. Kampman, C. Fluit, and J. Broekstra. Sesame - java rdf framework. http://rdf4j.org/, August 2008.

[Kle56] S. C. Kleene. Representations of events in nerve sets and finite automata. *Automata Studies*, pages 3–42, 1956.

[KLR+08] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu. SystemT: a system for declarative information extraction. *SIGMOD Record*, 37(4):7–13, 2008.

[KM13] C. Konrad and F. Magniez. Validating XML documents in the streaming model with external memory. *ACM Transactions on Database Systems (TODS)*, 38(4):27, 2013.

[KRV14] E. V. Kostylev, J. L. Reutter, and D. Vrgoč. Containment of data graph queries. In *Proceedings of the 17th International Conference on Database Theory (ICDT)*, pages 131–142. OpenProceedings.org, 2014.

[KS13a] W. Kazana and L. Segoufin. Enumeration of first-order queries on classes of structures with bounded expansion. In *Proceedings of the 32nd Symposium on Principles of Database Systems (PODS)*, pages 297–308. ACM, 2013.

[KS13b] W. Kazana and L. Segoufin. Enumeration of monadic second-order queries on trees. *ACM Transactions on Computational Logic (TOCL)*, 14(4):25:1–25:12, 2013.

[KSM95] S. Kannan, Z. Sweedyk, and S. R. Mahaney. Counting and random generation of strings in regular languages. In *Proceedings of the 6th Annual Symposium on Discrete Algorithms (SODA)*, pages 551–557. ACM/SIAM, 1995.

[KT03] P. Kilpeläinen and R. Tuhkanen. Regular expressions with numerical occurrence indicators — preliminary results. In *Proceedings of the 8th Symposium on Programming Languages and Software Tools (SPLST)*, pages 163–173. University of Kuopio, 2003.

[KT07] P. Kilpeläinen and R. Tuhkanen. One-unambiguity of regular expressions with numeric occurrence indicators. *Information and Computation (IC)*, 205(6):890–916, 2007.

[KW80] C. Kintala and D. Wotschke. Amounts of nondeterminism in finite automata. *Acta Informatica*, 13:199–204, 1980.

[LBC14] P. Lu, J. Bremer, and H. Chen. Deciding determinism of regular languages. *Theory of Computing Systems*, pages 1–43, 2014.

[LM12] K. Losemann and W. Martens. The complexity of evaluating path expressions in SPARQL. In *Proceedings of the 31st Symposium on Principles of Database Systems (PODS)*, pages 101–112. ACM, 2012.

[LM13] K. Losemann and W. Martens. The complexity of regular expressions and property paths in SPARQL. *ACM Transactions on Database Systems (TODS)*, 38(4):24:1–24:39, 2013.

[LM14] K. Losemann and W. Martens. MSO queries on trees: enumerating answers under updates. In *Proceedings of the Joint Meeting of the 23rd Annual Conference on Computer Science Logic and the 29th Annual Symposium on Logic in Computer Science (CSL-LICS)*, pages 67:1–67:10. ACM, 2014.

[LMN12] K. Losemann, W. Martens, and M. Niewerth. Descriptional complexity of deterministic regular expressions. In *Proceedings of the 37th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 643–654. Springer, 2012.

[LMV13] L. Libkin, W. Martens, and D. Vrgoč. Querying graph databases with XPath. In *Proceedings of the 16th International Conference on Database Theory (ICDT)*, pages 129–140. ACM, 2013.

[Los10] K. Losemann. Boolesche Operationen auf deterministischen regulären Ausdrücken. Master's thesis, TU Dortmund, October 2010.

[Los12] K. Losemann. Foundations of regular expressions in XML schema languages and SPARQL. In *Proceedings of the ACM SIGMOD/PODS PhD Symposium*, pages 39–44. ACM, 2012.

[LP84] A. S. Lapaugh and C. Papadimitriou. The even path problem for graphs and digraphs. *Networks*, 14(4):507–513, 1984.

[LRV13] L. Libkin, J. L. Reutter, and D. Vrgoč. Trial for RDF: adapting graph query languages for RDF data. In *Proceedings of the 32nd Symposium on Principles of Database Systems (PODS)*, pages 201–212. ACM, 2013.

[LTV13] L. Libkin, T. Tan, and D. Vrgoč. Regular expressions with binding over data words for querying graph databases. In *Proceedings of the 17th International Conference on Developments in Language Theory (DLT)*, pages 325–337. Springer, 2013.

[LV12] L. Libkin and D. Vrgoč. Regular path queries on graphs with data. In *Proceedings of the 15th International Conference on Database Theory (ICDT)*, pages 74–85. ACM, 2012.

[LY02] Y. A. Liu and F. Yu. Solving regular path queries. In *Proceedings of the 6th International Conference on Mathematics of Program Construction (MPC)*, pages 195–208. Springer, 2002.

[Man73] R. Mandl. Precise bounds associated with the subset construction on various classes of nondeterminism finite automata. In *Proceedings of the 7th Annual Princeton Conference on Information Science and Systems*, pages 263–267. Princeton University Press, 1973.

[Mar06] W. Martens. *Static analysis of XML transformation and schema languages*. PhD thesis, Hasselt University, 2006.

[MBPS05] S. Maneth, A. Berlea, T. Perst, and H. Seidl. XML type checking with macro tree transducers. In *Proceedings of the 24th Symposium on Principles of Database Systems (PODS)*, pages 283–294. ACM, 2005.

[MN05] W. Martens and F. Neven. On the complexity of typechecking top-down XML transformations. *Theoretical Computer Science (TCS)*, 336(1):153–180, 2005.

[MN07] W. Martens and J. Niehren. On the minimization of XML Schemas and tree automata for unranked trees. *Journal of Computer and System Sciences (JCSS)*, 73(4):550–583, 2007.

[MNS04] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. In *Proceedings of the 29th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 889–900. Springer, 2004.

[MNS07] W. Martens, F. Neven, and T. Schwentick. Simple off the shelf abstractions of XML Schema. *SIGMOD Record*, 36(3):15–22, 2007.

[MNS09] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for XML schemas and chain regular expressions. *SIAM Journal on Computing (SICOMP)*, 39(4):1486–1530, 2009.

[MNS10] W. Martens, M. Niewerth, and T. Schwentick. Schema design for XML repositories: Complexity and tractability. In *Proceedings of the 29th Symposium on Principles of Database Systems (PODS)*, ACM, 2010.

[MS72] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proceedings of the 13th Annual Symposium on Switching and Automata Theory (SWAT)*, pages 125–129. IEEE Computer Society, 1972.

[MS04] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM (JACM)*, 51(1):2–45, 2004.

[MSV03] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *Journal of Computer and System Sciences (JCSS)*, 66(1):66–97, 2003.

[MSVT94] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia. Complexity models for incremental computation. *Theoretical Computer Science (TCS)*, 130(1):203–236, 1994.

[MW95] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing (SICOMP)*, 24(6):1235–1258, 1995.

[Nev99] F. Neven. *Design and Analysis of Query Languages for Structured Documents*. PhD thesis, Limburgs Universitair Centrum, 1999.

[NPTT05] J. Niehren, L. Planque, J.-M. Talbot, and S. Tison. N-ary queries by tree automata. In *Proceedings of the 10th International Symposium on Database Programming Languages (DBPL)*, pages 217–231. Springer, 2005.

[NS06] F. Neven and T. Schwentick. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *Logical Methods in Computer Science (LMCS)*, 2(3):1–30, 2006.

[PAG09] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3):16:1–16:45, 2009.

[PAG10] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. *Journal of Web Semantics (JWS)*, 8(4):255–270, 2010.

[Pap94] C. M. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

[PI97] S. Patnaik and N. Immerman. Dyn-FO: A parallel, dynamic complexity class. *Journal of Computer and System Sciences (JCSS)*, 55(2):199–209, 1997.

[PS02] G. Pighizzini and J. Shallit. Unary language operations, state complexity and Jacobsthal's function. *International Journal of Foundations of Computer Science*, 13(1):145–159, 2002.

[Reg] Regular expression library. RegExLib.com.

[RG02] R. Ramakrishnan and J. Gehrke. *Database Management Systems (third edition)*. McGraw-Hill, 2002.

[RRK+08] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An algebraic approach to rule-based information extraction. In *Proceedings of the 24th International Conference on Data Engineering (ICDE)*, pages 933–942. IEEE Computer Society, 2008.

[Sav70] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences (JCSS)*, 4(2):177–192, 1970.

[Seg13] L. Segoufin. Enumerating with constant delay the answers to a query. In *Proceedings of the 16th International Conference on Database Theory (ICDT)*, pages 10–20. ACM, 2013.

[Sim90] I. Simon. Factorization forests of finite height. *Theoretical Computer Science (TCS)*, 72(1):65–94, 1990.

[SML10] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *Proceedings of the 13th International Conference on Database Theory (ICDT)*, pages 4–33. ACM, 2010.

[ST83] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences (JCSS)*, 26(3):362–391, 1983.

[Sto74] L. Stockmeyer. *The complexity of decision problems in automata theory and logic*. PhD thesis, Massachusetts Institute of Technology, 1974.

[SV02] L. Segoufin and V. Vianu. Validating streaming XML documents. In *Proceedings of the 21th Symposium on Principles of Database Systems (PODS)*, pages 53–64. ACM, 2002.

[SW11] R. Sedgewick and K. Wayne. *Algorithms (fourth edition)*. Addison-Wesley, 2011.

[SWY04] A. Salomaa, D. Wood, and S. Yu. On the state complexity of reversals of regular languages. *Theoretical Computer Science (TCS)*, 320(2-3):315–329, 2004.

[SY97] K. Salomaa and S. Yu. NFA to DFA transformation for finite languages over arbitrary alphabets. *Journal of Automata, Languages and Combinatorics (JALC)*, 2(3):177–186, 1997.

[Sze88] R. Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

[TW68] J. W. Thatcher and J. B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, 1968.

[Val79a] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science (TCS)*, 8(2):189–201, 1979.

[Val79b] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing (SICOMP)*, 8(3):410–421, 1979.

[vEB97] P. van Emde Boas. The convenience of tilings. In *Complexity, Logic and Recursion Theory*, volume 187 of *Lecture Notes in Pure and Applied Mathematics*, pages 331–363. Marcel Dekker Inc., 1997.

[Wil] G. T. Williams. RDF::Query - a complete SPARQL 1.1 Query and Update implementation for the use with RDF::Trine. http://search.cpan.org/dist/RDF-Query/lib/RDF/Query.pm.

[Woo03] P. T. Wood. Containment for XPath fragments under DTD constraints. In *Proceedings of the 9th International Conference on Database Theory (ICDT)*, pages 297–311. Springer, 2003.

[Yan90] M. Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the 9th Symposium on Principles of Database Systems (PODS)*, pages 230–242. ACM, 1990.

[Yu01] S. Yu. State complexity of regular languages. *Journal of Automata, Languages and Combinatorics (JALC)*, 6(2):221, 2001.

[YZS94] S. Yu, Q. Zhuang, and K. Salomaa. The state complexities of some basic operations on regular languages. *Theoretical Computer Science (TCS)*, 125(2):315 – 328, 1994.

List of Notations

Σ alphabet, 9

$\cdot\backslash\cdot\cdot$ Brzozowski derivative, 12

$\#$ shortcut for counting operators in regular expressions, 24

$[\cdot]$ equivalence class, 11

Δ infinite set of symbols, 69

$\bowtie$ join of two relations, 15

$\{\!\{\cdots\}\!\}$ multiset, 9

$!$ restricted negation operator, 24

$|\cdot|$ size (length) of $\cdot$, 11

δ transition function, 10

δ_ε transition function with ε-transitions, 148

$\bullet$ wildcard symbol, 24

$\langle\cdot\rangle$ nesting operator, 103

CHARE chain regular expression, 25

DFA deterministic finite automaton, 10

DFA^w deterministic finite automaton with wildcards, 70

DRE deterministic regular expression, 27

DTD Document Type Definition, 19

G_r Glushkov automaton, 31

G_r^ω Glushkov automaton with wildcards, 71

hp$(\cdot)$ heavy path of $\cdot$, 14

HPaths(t) set of all maximal heavy paths of nodes in t, 14

IRI Internationalized Resource Identifier, 21

k-NFSTA k-ary nondeterministic finite selecting tree automaton, 110

k-NFSA k-ary nondeterministic finite selecting automaton, 109

lab$(\cdot)$ label of $\cdot$, 12

$L(\cdot)$ language defined by $\cdot$, 10

lev$_k(D, q)$ level k automaton of D for the state q, 36

$[N]$ minimal DFA for an NFA N, 11

NFA nondeterministic finite automaton, 10

εNFA nondeterministic finite automaton with ε-transitions, 148

NFA^w nondeterministic finite automaton with wildcards, 70

Nodes(w) set of nodes for a word w, 109

NRE nested regular expression, 103

NTA (bottom-up) nondeterministic tree automaton, 109

num(r) numbered regular expression, 31

$\mathcal{P}(N)$ power set automaton of N, 10

$\mathcal{P}(S)$ power set of a set S, 9

RDF Resource Description Framework, 20

RE class of regular expressions, 11

$r(i)$ state reached by reading the first i symbols from the input, 10

$r_{\#}$ annotated expression of r, 26

r run of a word automaton, 10

$\mathrm{set}(\cdot)$ set of symbols occurring in $\cdot$, 109

SGML Standard Generalized Markup Language, 20

SPARQL SPARQL Protocol and RDF Query Language (recursive acronym), 22

t_v subtree of the node v in t, 14

UFA unambiguous finite automaton, 10

UFA^w unambiguous finite automaton with wildcards, 70

UPA Unique Particle Attribution, 26

URI Uniform Resource Identifier, 20

XML Extensible Markup Language, 18

XSD XML Schema Definition, 19

Index