Cross-Model Queries and Schemas: Complexity and Learning

Radu Ciucanu

Université Lille 1 – Sciences et Technologies
Ecole Doctorale Sciences pour l'Ingénieur
Institut National de Recherche en Informatique et Automatique

Cross-Model Queries and Schemas: Complexity and Learning

Thèse

soutenue le 1er juillet 2015
pour l'obtention du Doctorat, spécialité Informatique
par

Radu Ciucanu

Composition du jury :

Rapporteurs :	Sihem Amer-Yahia	CNRS
	Frank Neven	Universiteit Hasselt
Examinateurs :	Bogdan Cautis	Université Paris Sud
	Dan Olteanu	University of Oxford
Directrice de thèse :	Angela Bonifati	Université Lille 1
Co-encadrant :	Aurélien Lemay	Université Lille 3
Présidente du jury :	Laurence Duchien	Université Lille 1

Abstract

Specifying a database query using a formal query language is typically a challenging task for non-expert users. In the context of *big data*, this problem becomes even harder because it requires the users to deal with database instances of large size and hence difficult to visualize. Such instances usually lack a schema to help the users specify their queries, or have an incomplete schema as they come from disparate data sources.

In this thesis, we address the problem of query specification for non-expert users. We identify two possible approaches for tackling this problem: learning queries from examples and translating the data in a format that the user finds easier to query. Our contributions are aligned with these two complementary directions and span over three of the most popular data models: XML, relational, and graph. This thesis consists of two parts, dedicated to (i) schema definition and translation, and to (ii) learning schemas and queries.

In the first part, we define schema formalisms for unordered XML. We also characterize the computational complexity of the underlying problems of interest involving schemas and twig queries, and we show that the proposed schema languages are capable of expressing many practical languages of unordered trees and enjoy desirable computational properties. Moreover, we study the complexity of the data exchange problem from a relational source to a graph-shaped target, with heterogeneous schema mappings defined between them and additional constraints on the target graph. We prove the intractability of this setting, which holds even under significant restrictions.

In the second part, we investigate the problem of learning from examples, both for schemas and queries. We start with the unordered XML schemas proposed in the first part of the thesis, we show that they are not learnable in general, and consequently, we identify two learnable cases. Then, we focus on relational joins with equality predicates, and we also allow disjunction and projection in the queries. We characterize the frontier between tractability and intractability for several problems of interest including the learnability of these classes of queries. We also develop an interactive scenario of proposing examples to the user, which minimizes both the number of user interactions and the learning time. Finally, we investigate the problem of learning path queries defined by regular expressions on graph databases. We identify fundamental difficulties of this problem, we formalize an adapted learnability definition, and we show that the path queries on graphs are learnable. We also study an interactive setting that is similar to the one for relational joins. The interactive scenario that we develop for relational joins and path queries on graphs is immediately applicable to assisting non-expert users in the process of query specification, the main motivation of this thesis.

Résumé

La spécification de requêtes est généralement une tâche complexe pour les utilisateurs non-experts. Le problème devient encore plus difficile quand les utilisateurs ont besoin d'interroger des bases de données de grande taille et donc difficiles à visualiser. Le schéma pourrait aider à cette spécification, mais celui-ci manque souvent ou est incomplet quand les données viennent de sources hétérogènes.

Dans cette thèse, nous abordons le problème de la spécification de requêtes pour les utilisateurs non-experts. Nous identifions deux approches pour attaquer ce problème : apprendre les requêtes à partir d'exemples ou transformer les données dans un format plus facilement interrogeable par l'utilisateur. Nos contributions suivent ces deux directions complémentaires et concernent trois modèles de données parmi les plus populaires : les bases de données relationnelles, XML et orientées graphe. Cette thèse comprend deux parties, consacrées à (i) la définition et la transformation de schémas, et (ii) l'apprentissage de schémas et de requêtes.

Dans la première partie, nous définissons des formalismes de schémas pour les documents XML non-ordonnés. Nous prouvons que nos formalismes ont des bonnes propriétés computationnelles, tout en couvrant une grande partie des schémas utilisés en pratique. En outre, nous étudions la complexité du problème d'échange de données entre une source relationnelle et une cible orientée graphe, en considérant également des contraintes sur la cible. Nous prouvons la dureté de ce problème, qui surgit même sous fortes restrictions.

Dans la deuxième partie, nous investiguons le problème de l'apprentissage des schémas et des requêtes à partir d'exemples. D'abord, nous prouvons que les schémas XML proposés dans la première partie ne sont pas apprenables en général et nous identifions deux classes qui sont apprenables. Puis, nous étudions l'apprentissage des requêtes de jointures relationnelles et nous caractérisons les cas apprenables. De plus, nous développons un scénario interactif pour proposer des exemples à l'utilisateur, en minimisant le nombre d'interactions et le temps nécessaire pour apprendre. Finalement, nous investiguons l'apprentissage des requêtes de chemins définies par des expressions régulières sur les graphes. Nous identifions des obstacles fondamentaux du problème, nous formalisons une définition adaptée et nous prouvons dans quelles conditions l'apprentissage est possible pour les requêtes de chemins. Ensuite, nous étudions un scénario interactif similaire à celui du cas relationnel. Ce scénario interactif que nous avons developpé pour deux classes de requêtes (les jointures relationnelles et les chemins sur les graphes) permet effectivement d'aider des utilisateurs non-experts à définir des requêtes, ce qui est la motivation principale de cette thèse.

Contents

Introduction

Specifying a database query using a formal query language is typically a challenging task for non-expert users. Paradigms such as *query by example* [Zlo75] have been proposed for the relational databases to assist unfamiliar users to specify their queries. However, such paradigms have two underlying assumptions: (i) the data is of sufficiently small size to be visualized, and (ii) a schema is available to guide the user to formulate her query. In this thesis, we enlarge the spectrum of the data size and formats by considering formulation of queries in the realm of *big data*. In this case, the problem of query specification becomes even harder because it requires the users to deal with database instances of large size and hence difficult to visualize. Such instances usually lack a schema to help the users specify their queries, or have an incomplete schema as they come from disparate data sources.

The problem of *assisting non-expert users to specify their queries* has been recently raised by Jagadish et al. [JCE+07, NJ11]. More concretely, they have observed that "constructing a database query is often challenging for the user, commonly takes longer than the execution of the query itself, and does not use any insights from the database".

In this thesis, we address the problem of query specification for non-expert users. We identify two possible approaches for tackling this problem: (i) *learning* queries from examples and (ii) *translating the data* in a format that the user finds easier to query. Our contributions are aligned with these two complementary directions and span over three of the most popular data models: *XML*, *relational*, and *graph*. Since we consider heterogeneous data models, we address mainly the *Variety* aspect of *big data*, while putting less accent on the *Volume* and *Velocity* aspects.

We present in Figure 1 an outline of the thesis, which consists of *two parts* and *five chapters*. Each block corresponds to one chapter while each color corresponds to a part: the green blocks correspond to chapters of the first part (**Schema definition and translation**) while the red ones correspond to chapters of the second part (**Learning schemas and queries**). Moreover, notice the three columns of Figure 1, which correspond to the three data models that we consider in this thesis: XML, relational, and graph.

Furthermore, Figure 1 has also the goal of motivating the title of this thesis i.e., "*Cross-model queries and schemas: complexity and learning*". More precisely, the "cross-model" aspect is suggested by the three columns of the figure, the "complexity" and "learning" aspects correspond to the green and red blocks of the figure, respectively, while the "queries" and "schemas" are

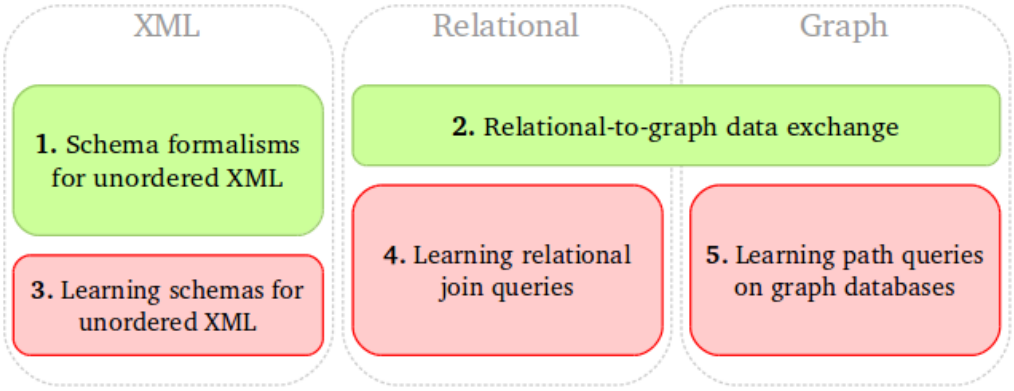


Figure 1: Outline of the two parts and five chapters of the thesis.

central concepts of all chapters. The choice of the precise problems that we investigate throughout the chapters depends on the existing related works, as we briefly show next for both parts of the thesis.

Part I. Schema definition and translation. This part consists of two chapters. In the first one, we study schemas for unordered XML. In the second one, we investigate a particular case of data exchange, with a relational source and a graph target database. Next, we briefly introduce the context of these two contributions.

Motivated by the general setting of learning queries from examples and by the increasing popularity of the XML data model, we were initially interested in the problem of *learning XML queries*. This problem has been already addressed [CGLN07, LNG06, SW12]. The closest work to common standards such as XPath and XQuery is [SW12], who investigated the learnability of *twig queries*, a highly-practical and commonly-used subclass of XPath. While evaluating the algorithms from [SW12] on XML benchmarks, we observed that they may return *overspecialized queries*, which include fragments implied by the schema. This happens because they do not exploit the schema of the XML documents, thus a natural idea would be to include the schema as input of query learning. One particularity of the twig queries is that they disregard the relative order among siblings in an XML document, hence it would be interesting to investigate the impact of schemas for *unordered XML*, which also ignore the relative order among siblings. Interestingly, we were not able to find in the literature any such practical schema language, which motivated us to develop **schema formalisms for unordered XML**. Thus, a first main contribution of this thesis is our work on defining schemas for unordered XML and thoroughly analyzing their computational properties.

We believe that our work on schemas for unordered XML is applicable in the future to *query minimization* [AYCLS02] i.e., given a query and a schema, find a smaller yet equivalent query in the presence of the schema. In particular, the most relevant part of our study towards this application is the computational complexity analysis of the problems of interest involving twig queries and schemas i.e., twig query satisfiability, implication, and containment in the presence of schema.

Furthermore, as already pointed out at the beginning of the introduction, an interesting problem of interest is to translate data from one format to another. This leads us to studying the problem of *data exchange*, which aims at translating data structured under a source schema according to a target schema and a set of constraints known as *schema mappings* [FKMP05]. Such a problem has been studied in settings where both the source and target schemas belong to the same data model, in particular relational and nested relational [PVM+02, FKMP05], XML [AL08], or graph [BPR13]. Settings in which the source and the target schema are of heterogeneous data models have not been considered so far, apart from combinations of relational and nested relational schemas in schema mapping tools [PVM+02, KPSS14]. We focus on the problem of exchanging data between relational sources and graph-shaped target databases, which might occur in several interoperability scenarios in the Semantic Web, such as ontology-based data access [PLC+08] and direct mappings [SAM12]. Thus, another contribution of the thesis is our work on **relational-to-graph data exchange**, on which we put less emphasis compared to the other four contributions since we show the intractability of the studied setting for very restricted classes of mappings.

Part II. Learning schemas and queries. This part consists of three chapters motivated by our interest in the setting of learning from examples.

First, we study the problem of **learning schemas for unordered XML** and we focus on the unordered schema formalisms introduced in the first part. This work has applications in *data integration* since automatically inferring good-quality XML schemas from document examples is an important step towards applying them in the process of data integration [Flo05]. This is clearly a data-centric application, therefore unordered XML schemas are the most appropriate. Additionally, we point out that our schema formalisms for unordered XML have been very recently extended to capture schemas for graph databases [SBG+15]. Thus, one could also leverage our techniques for learning unordered XML schemas to the goal of automatically inferring graph schemas and then employ them in the process of data integration.

Second, we investigate the problem of learning relational queries. This

problem has been very recently addressed in the context of quantified Boolean queries for the nested relational model [AAP+13]. Moreover, some of the problems strongly related to learning and already studied in the literature are the reverse engineering of join queries [ZEPS13], query by output [TCP09], synthesizing view definitions [DSPGMW10], and definability [Ban78, Par78]. However, we identified the problem of **learning relational join queries** from simple user interactions as a novel research problem. Our thorough study of this problem is a major contribution of this thesis.

Third, we focus on graph queries defined by *regular expressions*, which are fundamental for graph query languages [Bar13, Woo12] and employed in practice in the definition of the SPARQL property paths. Regular expressions define *path queries* that essentially select nodes iff one can navigate between them via a path labeled by a word in the language of a given regular expression. While the problem of executing such path queries has been extensively studied recently [BBG13, LM13, KL12], we are the first to address the problem of learning them from examples. Thus, another major contribution of this thesis is our work on **learning path queries on graph databases**.

The techniques that we developed for the last two mentioned directions are immediately applicable to *assisting non-expert users in the process of query specification*, which is the main motivation of the thesis. In particular, we formalized an interactive query learning scenario that we instantiated for each of the two considered classes of queries. Moreover, since we aim at inferring the queries while minimizing the amount of user feedback, our research is also applicable to *crowdsourcing* scenarios [FKK+11], in which such minimization typically entails lower financial costs. Indeed, we can imagine scenarios where crowdworkers are interactively provided with small fragments of the database that they should label as positive or negative examples for the goal of query learning.

Outline and contributions

In this section, we summarize the contributions for each of the five chapters of the thesis. We start with the **first part**, which consists of two chapters dedicated to schema definition and translation, respectively.

Chapter 1. We study schema languages for unordered XML i.e., having no relative order among siblings. First, we propose *unordered regular expressions* (UREs), essentially regular expressions with *unordered concatenation* instead of standard concatenation, which define languages of unordered words to model the allowed content of a node. However, unrestricted UREs

are computationally too expensive as we show the intractability of two fundamental decision problems for UREs: membership of an unordered word to the language of a URE and containment of two UREs. Consequently, we propose a practical and tractable restriction of UREs, *disjunctive interval multiplicity expressions* (DIMEs). Next, we employ DIMEs to define languages of unordered trees and propose two schema languages: *disjunctive interval multiplicity schema* (DIMS), and its restriction, *disjunction-free interval multiplicity schema* (IMS). We study the complexity of the following static analysis problems: schema satisfiability, membership of a tree to the language of a schema, schema containment, as well as twig query satisfiability, implication, and containment in the presence of schema. Finally, we study the expressive power of the proposed schema languages and compare them with yardstick languages of unordered trees (FO, MSO, and Presburger constraints) and DTDs under commutative closure. We show that the proposed schema languages are capable of expressing many practical languages of unordered trees and enjoy desirable computational properties.

Chapter 2. We investigate the problem of data exchange in a heterogeneous setting, where the source is a relational database, the target is a graph database, and the schema mappings are defined across them. We study the classical problems considered in data exchange, namely the existence of solutions and query answering. We show that both problems are intractable in the presence of target constraints, already under significant restrictions.

The **second part** of the thesis consists of three chapters dedicated to learning queries and schemas. Before introducing the specific contributions of these chapters, we would like to describe some of their common aspects.

Learning queries from examples essentially means that we ask the user to label fragments of the database as *positive* or *negative examples* (depending on whether or not she would like them as part of the query result) and then we run a *learning algorithm* that ideally returns the query that the user has in mind. This general problem statement applies for both learning relational joins and learning path queries on graphs. Moreover, it can be easily adapted for *learning schemas* for unordered XML, the difference being that the user labels documents and the learning algorithm returns a schema.

In all three learning chapters, we employ definitions of *learnability* inspired by the well-known framework of *language identification in the limit* [Gol78], which requires a learning algorithm to be *polynomial* in the size of the input, *sound* (i.e., always return a concept consistent with the examples given by the user or a special *null* value if such concept does not exist) and *complete* (i.e.,

able to produce every concept with a sufficiently rich set of examples). Since the three considered data models underline different fundamental difficulties, each of the three learning chapters uses its own learnability definition, which is a slightly-adapted version of the aforementioned standard one. Moreover, we always justify our choices with theoretical arguments.

Chapter 3. We investigate the problem of learning schemas from examples given by the user and we focus on the unordered schema formalisms introduced in Chapter 1. We prove that the schemas defined by DIMEs are not learnable in the general case, where both positive and negative examples are allowed. Consequently, we identify two practical restrictions that are learnable: one from positive examples only, and another one from both positive and negative examples. Furthermore, for both learnable cases, the proposed learning algorithms return minimal schemas consistent with the examples.

Chapter 4. We study the problem of learning relational join queries. We consider arbitrary n-ary join predicates across an arbitrary number m of relations, without assuming any prior knowledge of the integrity constraints across the involved relations. We introduce a set of strategies that let us inspect the search space and aggressively prune what we call "uninformative" tuples, and directly present to the user the *informative* ones i.e., those that allow to quickly find the goal query that the user has in mind. We focus on the inference of joins with equality predicates, and we also allow disjunctive join predicates and projection in the queries. We precisely characterize the frontier between tractability and intractability for the following problems of interest in these settings: consistency checking, learnability, and deciding the informativeness of a tuple. Next, we propose several strategies for presenting tuples to the user in a given order that lets minimize the number of interactions. We show the efficiency and scalability of our approach through an experimental study on both benchmark and synthetic datasets.

Chapter 5. We investigate the problem of learning *path queries* defined by *regular expressions*, we identify fundamental difficulties of our problem setting, we formalize what it means to be *learnable*, and we prove that the class of queries under study enjoys this property. We also investigate an interactive scenario where we start with an empty set of examples and we identify the *informative nodes* i.e., those that contribute to the learning process. Then, we ask the user to label these nodes and iterate the learning process until she is satisfied with the learned query. Finally, we present an experimental study on both real and synthetic datasets devoted to gauging the effectiveness of our learning algorithm and the improvement of the interactive approach.

We would highlight Chapter 4, Chapter 5, and Chapter 1 as the main contributions of the thesis. For this purpose, we want to point out that in Figure 1 there is a correlation between the height of a box and how important we believe to be the respective contribution. Indeed, in Chapter 4 and Chapter 5 we investigate the theoretical foundations of learning relational joins and path queries, respectively, from both static and interactive points of view. Moreover, we implemented our algorithms to build system prototypes that assist non-expert users to specify queries. Among the three theoretical chapters, Chapter 1 (which studies novel schema formalisms for unordered XML) clearly provides the most complete and interesting theoretical results.

As a remark related to the readability of the thesis, we mention that the chapters are generally independent and can be read in an arbitrary order. The only exception is related to the XML chapters, for which we recommend reading Chapter 1 before Chapter 3.

Publications

It is important to point out that this thesis is based on several publications in international journals, conferences, and workshops, as we detail next.

Chapter 1 (*Schema formalisms for unordered XML*) corresponds to our article in the Theory of Computing Systems journal [BCS14a], which is the extended version our WebDB'13 paper [BCS13].

Chapter 2 (*Relational-to-graph data exchange*) corresponds to our paper in an EDBT/ICDT'15 workshop on querying graph databases [BBC15].

Chapter 3 (*Learning schemas for unordered XML*) is based on our DBPL'13 paper [CS13].

Chapter 4 (*Learning relational join queries*) is an extended version of our EDBT'14 research paper [BCS14b], currently under journal submission [BCS14d]. We demonstrated the underlying system in VLDB'14 [BCS14c].

Chapter 5 (*Learning path queries on graph databases*) corresponds to our EDBT'15 research paper [BCL15b], and moreover, we demonstrated our system in EDBT'15 [BCL15a].

Additionally, we presented in [Ciu13] and [BCLS14] our general vision on the topic of learning queries.

Part I

Schema definition and translation

Chapter 1

Schema formalisms for unordered XML

In this chapter, we investigate schema languages for unordered XML having no relative order among siblings. First, we propose *unordered regular expressions* (UREs), essentially regular expressions with *unordered concatenation* instead of standard concatenation, that define languages of unordered words to model the allowed content of a node (i.e., collections of the labels of children). However, unrestricted UREs are computationally too expensive as we show the intractability of two fundamental decision problems for UREs: membership of an unordered word to the language of a URE and containment of two UREs. Consequently, we propose a practical and tractable restriction of UREs, *disjunctive interval multiplicity expressions* (DIMEs).

Next, we employ DIMEs to define languages of unordered trees and propose two schema languages: *disjunctive interval multiplicity schema* (DIMS), and its restriction, *disjunction-free interval multiplicity schema* (IMS). We study the complexity of the following static analysis problems: schema satisfiability, membership of a tree to the language of a schema, schema containment, as well as twig query satisfiability, implication, and containment in the presence of schema. Finally, we study the expressive power of the proposed schema languages and compare them with yardstick languages of unordered trees (FO, MSO, and Presburger constraints) and DTDs under commutative closure. Our results show that the proposed schema languages are capable of expressing many practical languages of unordered trees and enjoy desirable computational properties.

1.1 Context

When XML is used for *document-centric* applications, the relative order among the elements is typically important e.g., the relative order of paragraphs and chapters in a book. On the other hand, in case of *data-centric* XML applications, the order among the elements may be unimportant [ABV12]. In this chapter we focus on the latter use case. As an example, take a trivialized fragment of an XML document containing the DBLP repository in

Figure 1.1. While the order of the elements title, author, and year may differ from one publication to another, it has no impact on the semantics of the data stored in this semi-structured database.

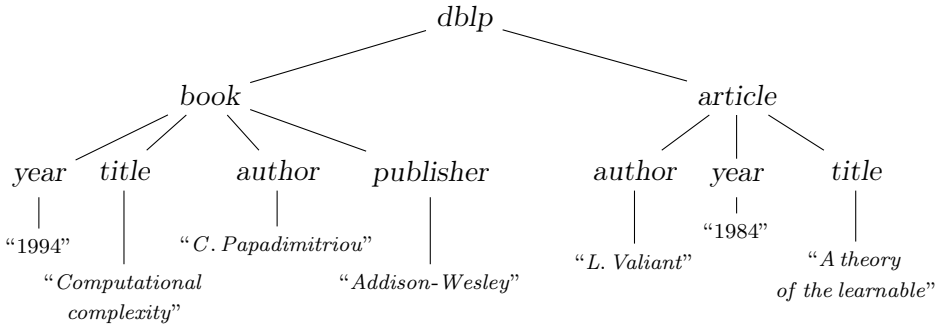


Figure 1.1: A trivialized DBLP repository.

Typically, a *schema* for XML defines for every node its *content model* i.e., the children nodes it must, may, and cannot contain. For instance, in the DBLP example, one would require every article to have exactly one title, one year, and one or more authors. A book may additionally contain one publisher and may also have one or more editors instead of authors. A schema has numerous important uses. For instance, it allows to validate a document against a schema and identify potential errors. A schema also serves as a reference for a user who does not know yet the structure of the XML document and attempts to query or modify its content.

The *Document Type Definition* (DTD), the most widespread XML schema formalism for (ordered) XML [BNVdB04, GM13], is essentially a set of rules associating with each label a regular expression that defines the admissible sequences of children. The DTDs are best fitted for ordered content because they use regular expressions, a formalism that defines sequences of labels. However, when unordered content model needs to be defined, there is a tendency to use *over-permissive* regular expressions. For instance, the DTD below corresponds to the one used in practice for the DBLP repository[1]:

$$\begin{aligned} \mathit{dblp} &\to (\mathit{article} \mid \mathit{book})^* \\ \mathit{article} &\to (\mathit{title} \mid \mathit{year} \mid \mathit{author})^* \\ \mathit{book} &\to (\mathit{title} \mid \mathit{year} \mid \mathit{author} \mid \mathit{editor} \mid \mathit{publisher})^* \end{aligned}$$

This DTD allows an article to contain any number of titles, years, and authors. A book may also have any number of titles, years, authors, editors,

[1] http://dblp.uni-trier.de/xml/dblp.dtd

and publishers. These regular expressions are clearly over-permissive because they allow XML documents that do not follow the intuitive guidelines set out earlier e.g., an XML document containing an article with two titles and no author should not be valid.

While it is possible to capture unordered content models with regular expressions, a simple pumping argument shows that their size may need to be exponential in the number of possible labels of the children. In case of the DBLP repository, this number reaches values up to 12, which basically precludes any practical use of such regular expressions. This suggests that over-permissive regular expressions may be employed for the reasons of conciseness and readability, a consideration of great practical importance.

The use of over-permissive regular expressions, apart from allowing documents that do not follow the guidelines, has other negative consequences e.g., in static analysis tasks that involve the schema. Take for example the following two twig queries [AYCLS02, W3C]:

$$/dblp/book[author = \text{“}C.\ Papadimitriou\text{”}]$$
$$/dblp/book[author = \text{“}C.\ Papadimitriou\text{”}][title]$$

The first query selects the elements labeled book, children of dblp and having an author containing the text “*C. Papadimitriou.*” The second query additionally requires that book has a title. Naturally, these two queries should be equivalent because every book should have a title. However, the DTD above does not capture properly this requirement, and consequently the two queries are not equivalent w.r.t. this DTD.

In this chapter, we investigate schema languages for unordered XML. First, we study languages of *unordered words*, where an unordered word can be seen as a multiset of symbols. We consider *unordered regular expressions* (UREs), which are essentially regular expressions with *unordered concatenation* “$\|$” instead of standard concatenation. The unordered concatenation can be seen as union of multisets, and consequently, the star “$*$” can be seen as the Kleene closure of unordered languages. Similarly to a DTD which associates to each label a regular expression to define its (ordered) content model, an unordered schema uses UREs to define for each label its unordered content model. For instance, take the following schema (satisfied by the tree in Figure 1.1):

$$\begin{aligned} dblp &\to article^* \,\|\, book^* \\ article &\to title \,\|\, year \,\|\, author^+ \\ book &\to title \,\|\, year \,\|\, publisher^? \,\|\, (author^+ \mid editor^+) \end{aligned}$$

The above schema uses UREs and captures the intuitive requirements for the DBLP repository. In particular, an article must have exactly one title, exactly one year, and at least one author. A book may additionally have a publisher and may have one or more editors instead of authors. Note that, unlike the DTD defined earlier, this schema does not allow documents having an article with several titles or without any author.

Using UREs is equivalent to using DTDs with regular expressions interpreted under the *commutative closure* [BM99, NS]: essentially, a word matches the commutative closure of a regular expression if there exists a permutation of the word that matches the regular expression in the standard way. Deciding this problem is known to be NP-complete [KT10] for arbitrary regular expressions. We show that the problem of testing the membership of an unordered word to the language of a URE is NP-complete even for a restricted subclass of UREs that allows unordered concatenation and the option operator "?" only. Not surprisingly, testing the containment of two UREs is also intractable. These results are of particular interest because they are novel and do not follow from complexity results for regular expressions, where the order plays typically an essential role [SM73, MS94]. Consequently, we focus on finding restrictions rendering UREs tractable and capable of capturing practical languages in a simple and concise manner.

The first restriction is to disallow repetitions of a symbol in a URE, thus banning expressions of the form $a \,\|\, a^?$ because the symbol a is used twice. Instead we add general interval multiplicities $a^{[1,2]}$ which offer a way to specify a range of occurrences of a symbol in an unordered word without repeating a symbol in the URE. While the complexity of the membership of an unordered word to the language of a URE with interval multiplicities and without symbol repetitions has recently been shown to be in PTIME [SBG+15], testing containment of two such UREs remains intractable. We, therefore, add limitations on the nesting of the disjunction and the unordered concatenation operators and the use of intervals, which yields the proposed class of *disjunctive interval multiplicity expressions* (DIMEs). DIMEs enjoy good computational properties: both the membership and the containment problems become tractable. Also, we believe that despite the imposed restriction DIMEs remain a practical class of UREs. For instance, all UREs used in the schema for the DBLP repository above are DIMEs.

Next, we employ DIMEs to define languages of unordered trees and propose two schema languages: *disjunctive interval multiplicity schema* (DIMS), and its restriction, *disjunction-free interval multiplicity schema* (IMS). Naturally, the above schema for the DBLP repository is a DIMS. We study the complexity of several basic decision problems: schema satisfiability, membership of a tree to the language of a schema, containment of two schemas, twig

Problem of interest	*DTD*	*DIMS*	*disj.-free DTD*	*IMS*
Schema satisfiability	PTIME [BKW98, Sch04]	PTIME (Pr. 1.6.1)	PTIME [BKW98, Sch04]	PTIME (Pr. 1.6.1)
Membership	PTIME [BKW98, Sch04]	PTIME (Pr. 1.6.2)	PTIME [BKW98, Sch04]	PTIME (Pr. 1.6.2)
Schema containment	PSPACE-c† [Sch04] PTIME [BKW98]	PTIME (Pr. 1.6.1)	coNP-h† [MNS09] PTIME [BKW98]	PTIME (Pr. 1.6.1)
Query satisfiability‡	NP-c [BFG08]	NP-c (Pr. 1.6.3)	PTIME [BFG08]	PTIME (Th. 1.7.9)
Query implication‡	EXPTIME-c [NS06]	EXPTIME-c (Pr. 1.6.4)	PTIME (Th. 1.7.11)	PTIME (Th. 1.7.9)
Query containment‡	EXPTIME-c [NS06]	EXPTIME-c (Pr. 1.6.4)	coNP-c (Th. 1.7.11)	coNP-c (Th. 1.7.10)

† when non-deterministic regular expressions are used. ‡ for twig queries.

Table 1.1: Summary of complexity results.

query satisfiability, implication, and containment in the presence of schema. We present in Table 1.1 a summary of the complexity results and we observe that DIMSs and IMSs enjoy the same computational properties as general DTDs and disjunction-free DTDs, respectively.

The lower bounds for the decision problems for DIMSs and IMSs are generally obtained with easy adaptations of their counterparts for general DTDs and disjunction-free DTDs. To obtain the upper bounds we develop several new tools. We propose to represent DIMEs with *characterizing tuples* that can be efficiently computed and allow deciding in polynomial time the membership of a tree to the language of a DIMS and the containment of two DIMSs. Also, we develop *dependency graphs* for IMSs and a generalized definition of an *embedding* of a query. These two tools help us to reason about query satisfiability, query implication, and query containment in the presence of IMSs. Our constructions and results for IMSs allow also to characterize the complexity of query implication and query containment in the presence of disjunction-free DTDs, which, to the best of our knowledge, have not been previously studied.

Finally, we compare the expressive power of the proposed schema languages with yardstick languages of unordered trees (FO, MSO, and Presburger constraints) and DTDs under commutative closure. We show that the proposed schema languages are capable of expressing many practical languages of unordered trees.

It is important to mention that this chapter has been published in the journal Theory of Computing Systems [BCS14a] and a preliminary version has been presented in [BCS13]. More precisely, [BCS14a] substantially extends [BCS13] by showing novel intractability results for some subclasses

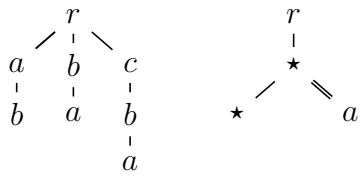
of unordered regular expressions and by extending the expressibility of the tractable subclasses. While in [BCS13] we have considered only simple multiplicities $(*, +, ?)$, in [BCS14a] and in this chapter we deal with arbitrary interval multiplicities of the form $[n, m]$.

Organization. In Section 1.2 we introduce some basic notions. In Section 1.3 we study the reasons of intractability of unordered regular expressions while in Section 1.4 we present the tractable subclass of *disjunctive interval multiplicity expressions* (DIMEs). In Section 1.5 we define two schema languages: the *disjunctive interval multiplicity schemas* (DIMSs) and its restriction, the *disjunction-free interval multiplicity schemas* (IMSs), and the related problems of interest. In Section 1.6 and Section 1.7 we analyze the complexity of the problems of interest for DIMSs and IMSs, respectively. In Section 1.8 we discuss the expressiveness of the proposed formalisms. In Section 1.9 we present related work.

1.2 Basic notions

We assume an alphabet Σ that is a finite set of symbols. We also assume that Σ has a total order $<_\Sigma$ that can be tested in constant time.

Trees. We model XML documents with unordered labeled trees. Formally, a *tree* t is a tuple $(N_t, root_t, lab_t, child_t)$, where N_t is a finite set of nodes, $root_t \in N_t$ is a distinguished root node, $lab_t : N_t \to \Sigma$ is a labeling function, and $child_t \subseteq N_t \times N_t$ is the parent-child relation. We assume that the relation $child_t$ is acyclic and require every non-root node to have exactly one predecessor in this relation. By *Tree* we denote the set of all trees.



(a) Tree t_0. (b) Twig query q_0.

Figure 1.2: A tree and a twig query.

Queries. We work with the class of twig queries, which are essentially unordered trees whose nodes may be additionally labeled with a distinguished

wildcard symbol $\star \notin \Sigma$ and that use two types of edges, child (/) and descendant (//), corresponding to the standard XPath axes. Note that the semantics of the //-edge is that of a proper descendant (and not that of descendant-or-self). Formally, a *twig query* q is a tuple $(N_q, root_q, lab_q, child_q, desc_q)$, where N_q is a finite set of nodes, $root_q \in N_q$ is the root node, $lab_q : N_q \to \Sigma \cup \{\star\}$ is a labeling function, $child_q \subseteq N_q \times N_q$ is a set of child edges, and $desc_q \subseteq N_q \times N_q$ is a set of descendant edges. We assume that $child_q \cap desc_q = \varnothing$ and that the relation $child_q \cup desc_q$ is acyclic and we require every non-root node to have exactly one predecessor in this relation. By *Twig* we denote the set of all twig queries. Twig queries are often presented using the abbreviated XPath syntax [W3C] e.g., the query q_0 in Figure 1.2(b) can be written as $r/\star[\star]//a$.

Embeddings. We define the semantics of twig queries using the notion of embedding which is essentially a mapping of nodes of a query to the nodes of a tree that respects the semantics of the edges of the query. Formally, for a query $q \in Twig$ and a tree $t \in Tree$, an *embedding* of q in t is a function $\lambda : N_q \to N_t$ such that:

1. $\lambda(root_q) = root_t$,
2. for every $(n, n') \in child_q$, $(\lambda(n), \lambda(n')) \in child_t$,
3. for every $(n, n') \in desc_q$, $(\lambda(n), \lambda(n')) \in (child_t)^+$ (the transitive closure of $child_t$),
4. for every $n \in N_q$, $lab_q(n) = \star$ or $lab_q(n) = lab_t(\lambda(n))$.

We write $t \preccurlyeq q$ if there exists an embedding of q in t. Later on, in Section 1.7.2 we generalize this definition of embedding as a tool that permits us characterizing the problems of interest.

As already mentioned, we use the notion of embedding to define the semantics of twig queries. In particular, we say that t *satisfies* q if there exists an embedding of q in t and we write $t \models q$. By $L(q)$ we denote the set of all trees satisfying q.

Note that we do not require the embedding to be injective i.e., two nodes of the query may be mapped to the same node of the tree. Figure 1.3 presents all embeddings of the query q_0 in the tree t_0 from Figure 1.2.

Unordered words. An *unordered word* is essentially a multiset of symbols i.e., a function $w : \Sigma \to \mathbb{N}_0$ mapping symbols from the alphabet to natural numbers. We call $w(a)$ the number of occurrences of the symbol a in w. We also write $a \in w$ as a shorthand for $w(a) \neq 0$. An empty word ε is an

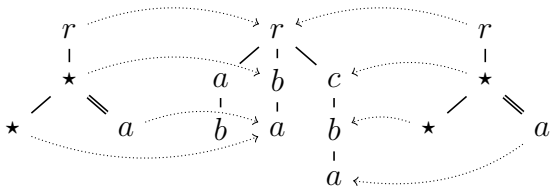


Figure 1.3: Embeddings of q_0 in t_0.

unordered word that has 0 occurrences of every symbol i.e., $\varepsilon(a) = 0$ for every $a \in \Sigma$. We often use a simple representation of unordered words, writing each symbol in the alphabet the number of times it occurs in the unordered word. For example, when the alphabet is $\Sigma = \{a, b, c\}$, $w_0 = aaacc$ stands for the function $w_0(a) = 3$, $w_0(b) = 0$, and $w_0(c) = 2$. Additionally, we may write $w_0 = a^3c^2$ instead of $w_0 = aaacc$.

We use unordered words to model collections of children of XML nodes. As it is usually done in the context of XML validation [SV02, SS07], we assume that the XML document is encoded in unary i.e., every node takes the same amount of memory. Thus, we use a unary representation of unordered words, where each occurrence of a symbol occupies the same amount of space. However, we point out that none of the presented results changes with a binary representation. In particular, the intractability of the membership of an unordered word to the language of a URE (Theorem 1.3.1) also holds with a binary representation of unordered words.

Consequently, the *size* of an unordered word w, denoted $|w|$, is the sum of the numbers of occurrences in w of all symbols in the alphabet. For instance, the size of $w_0 = aaacc$ is $|w_0| = 5$.

The (*unordered*) *concatenation* of two unordered words w_1 and w_2 is defined as the multiset union $w_1 \uplus w_2$ i.e., the function defined as $(w_1 \uplus w_2)(a) = w_1(a) + w_2(a)$ for every $a \in \Sigma$. For instance, $aaacc \uplus abbc = aaaabbccc$. Note that ε is the identity element of the unordered concatenation $\varepsilon \uplus w = w \uplus \varepsilon = w$ for every unordered word w. Also, given an unordered word w, by w^i we denote the concatenation $w \uplus \ldots \uplus w$ (i times).

A *language* is a set of unordered words. The unordered concatenation of two languages L_1 and L_2 is a language $L_1 \uplus L_2 = \{w_1 \uplus w_2 \mid w_1 \in L_1, w_2 \in L_2\}$. For instance, if $L_1 = \{a, aac\}$ and $L_2 = \{ac, b, \varepsilon\}$, then $L_1 \uplus L_2 = \{a, ab, aac, aabc, aaacc\}$.

Unordered regular expressions. Analogously to regular expressions, which are used to define languages of ordered words, we propose unordered regular expressions to define languages of unordered words. Essentially, an

unordered regular expression (URE) defines unordered words by using Kleene star "$*$", disjunction "$|$", and unordered concatenation "$\|$". Formally, we have the following grammar:

$$E ::= \epsilon \mid a \mid E^* \mid (E\text{“}|\text{”}E) \mid (E\text{“}\|\text{”}E),$$

where $a \in \Sigma$. The semantics of UREs is defined as follows:

$$\begin{aligned}
&L(\epsilon) = \{\varepsilon\},\\
&L(a) = \{a\},\\
&L(E_1 \mid E_2) = L(E_1) \cup L(E_2),\\
&L(E_1 \,\|\, E_2) = L(E_1) \uplus L(E_2),\\
&L(E^*) = \{w_1 \uplus \ldots \uplus w_i \mid w_1, \ldots, w_i \in L(E) \wedge i \geqslant 0\}.
\end{aligned}$$

For instance, the URE $(a \,\|\, (b \mid c))^*$ accepts the unordered words having the number of occurrences of a equal to the total number of b's and c's.

The grammar above uses only one *multiplicity* $*$ and we introduce macros for two other standard and commonly used multiplicities:

$$E^+ := E \,\|\, E^*, \qquad E^? := E \mid \epsilon.$$

The URE $(a \,\|\, b^?)^+ \,\|\, (a \mid c)^?$ accepts the unordered words having at least one a, at most one c, and a number of b's less or equal than the number of a's.

Interval multiplicities. While the multiplicities $*$, $+$, and $?$ allow to specify unordered words with multiple occurrences of a symbol, we additionally introduce *interval multiplicities* to allow to specify a range of allowed occurrences of a symbol in an unordered word. More precisely, we extend the grammar of UREs by allowing expressions of the form $E^{[n,m]}$ and $E^{[n,m]^?}$, where $n \in \mathbb{N}_0$ and $m \in \mathbb{N}_0 \cup \{\infty\}$. Their semantics is defined as follows:

$$\begin{aligned}
&L(E^{[n,m]}) = \{w_1 \uplus \ldots \uplus w_i \mid w_1, \ldots, w_i \in L(E) \wedge n \leqslant i \leqslant m\},\\
&L(E^{[n,m]^?}) = L(E^{[n,m]}) \cup \{\varepsilon\}.
\end{aligned}$$

In the remainder, we write simply *interval* instead of *interval multiplicity*. Furthermore, we view the following standard multiplicities as macros for intervals:

$$* := [0, \infty], \qquad + := [1, \infty], \qquad ? := [0, 1].$$

Additionally, we introduce the single occurrence multiplicity 1 as a macro for the interval $[1, 1]$.

Note that the intervals do not add expressibility to general UREs, but they become useful if we impose some restrictions. For example, if we disallow repetitions of a symbol in a URE and ban expressions of the form $a \parallel a^?$, we can however write $a^{[1,2]}$ to specify a range of occurrences of a symbol in an unordered word without repeating a symbol in the URE.

1.3 Intractability of unordered regular expressions

In this section, we study the reasons of the intractability of UREs w.r.t. the following two fundamental decision problems: *membership* and *containment*. In Section 1.3.1 we show that membership is NP-complete even under significant restrictions on the UREs while in Section 1.3.2 we show that the containment is Π_2^{P}-hard (and in 3-EXPTIME). We notice that the proofs of both results rely on UREs allowing repetitions of the same symbol. Consequently, we disallow such repetitions and we show that this restriction does not avoid intractability of the containment (Section 1.3.3). We observe that the proof of this result employs UREs with arbitrary use of disjunction and intervals, and therefore, in Section 1.4 we impose further restrictions and define the disjunctive interval multiplicity expressions (DIMEs), a subclass for which we show that the two problems of interest become tractable.

1.3.1 Membership

In this section, we study the problem of deciding the *membership* of an unordered word to the language of a URE. First of all, note that this problem can be easily reduced to testing the membership of a vector to the Parikh image of a regular language, known to be NP-complete [KT10], and vice versa. We show that deciding the membership of an unordered word to the language a URE remains NP-complete even under significant restrictions on the class of UREs, a result which does not follow from [KT10].

Theorem 1.3.1 *Given an unordered word w and an expression E of the grammar $E ::= a \mid E^? \mid (E \text{“}\parallel\text{”} E)$, deciding whether $w \in L(E)$ is NP-complete.*

Proof To show that this problem is in NP, we point out that a nondeterministic Turing machine guesses a permutation of w and checks whether it is accepted by the NFA corresponding to E with the unordered concatenation replaced by standard concatenation. We recall that w has unary representation.

Next, we prove the NP-hardness by reduction from $\text{SAT}_{\text{1-in-3}}$ i.e., given a 3CNF formula, determine whether there exists a valuation such that each clause has exactly one true literal (and exactly two false literals). The $\text{SAT}_{\text{1-in-3}}$ problem is known to be NP-complete [Sch78]. The reduction works as follows. We take a 3CNF formula $\varphi = c_1 \wedge \ldots \wedge c_k$ over the variables $\{x_1, \ldots, x_n\}$. We take the alphabet $\{d_1, \ldots, d_k, v_1, \ldots, v_n\}$. Each d_i corresponds to a clause c_i (for $1 \leqslant i \leqslant k$) and each v_j corresponds to a variable x_j (for $1 \leqslant j \leqslant n$). We construct the unordered word $w_\varphi = d_1 \ldots d_k v_1 \ldots v_n$ and the expression $E_\varphi = X_1 \parallel \ldots \parallel X_n$, where for $1 \leqslant j \leqslant n$:

$$X_j = (v_j \parallel d_{t_1} \parallel \ldots \parallel d_{t_l})^? \parallel (v_j \parallel d_{f_1} \parallel \ldots \parallel d_{f_m})^?,$$

and $d_{t_1}, \ldots, d_{t_l}$ (with $1 \leqslant t_1, \ldots, t_l \leqslant k$) correspond to the clauses that use the literal x_j, and $d_{f_1}, \ldots d_{f_m}$ (with $1 \leqslant f_1, \ldots, f_m \leqslant k$) correspond to the clauses that use the literal $\neg x_j$. For example, for the formula $\varphi_0 = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$, we construct $w_{\varphi_0} = d_1 d_2 v_1 v_2 v_3 v_4$ and

$$E_{\varphi_0} = (v_1 \parallel d_1)^? \parallel (v_1 \parallel d_2)^? \parallel v_2^? \parallel (v_2 \parallel d_1)^? \parallel (v_3 \parallel d_1 \parallel d_2)^? \parallel v_3^? \parallel v_4^? \parallel (v_4 \parallel d_2)^?.$$

We claim that $\varphi \in \text{SAT}_{\text{1-in-3}}$ iff $w_\varphi \in L(E_\varphi)$. For the *only if* case, let $V : \{x_1, \ldots, x_n\} \to \{true, false\}$ be the $\text{SAT}_{\text{1-in-3}}$ valuation of φ. We use V to construct the derivation of w_φ in $L(E_\varphi)$: for $1 \leqslant j \leqslant n$, we take $(v_j \parallel d_{t_1} \parallel \ldots \parallel d_{t_l})$ from X_j if $V(x_j) = true$, and $(v_j \parallel d_{f_1} \parallel \ldots \parallel d_{f_m})$ from X_j otherwise. Since V is a $\text{SAT}_{\text{1-in-3}}$ valuation of φ, each d_i (with $1 \leqslant i \leqslant k$) occurs exactly once, hence $w_\varphi \in L(E_\varphi)$. For the *if* case, we assume that $w_\varphi \in L(E_\varphi)$. Since $w_\varphi(v_j) = 1$, we infer that w_φ uses exactly one of the expressions of the form $(v_j \parallel \ldots)^?$. Moreover, since $w_\varphi(d_i) = 1$, we infer that the valuation encoded in the derivation of w_φ in $L(E_\varphi)$ validates exactly one literal of each clause in φ, and therefore, $\varphi \in \text{SAT}_{\text{1-in-3}}$. Clearly, the described reduction works in polynomial time. □

1.3.2 Containment

In this section, we study the problem of deciding the *containment* of two UREs. It is well known that regular expression containment is a PSPACE-complete problem [SM73], but we cannot adapt this result to characterize the complexity of the containment of UREs because the order plays an essential role in the reduction. In this section, we prove that deciding the containment of UREs is Π_2^{P}-hard and we show an upper bound which follows from the complexity of deciding the satisfiability of Presburger logic formulas [Opp78, SSM08].

Theorem 1.3.2 *Given two UREs E_1 and E_2, deciding $L(E_1) \subseteq L(E_2)$ is* 1) Π_2^{P}*-hard and* 2) *in 3-EXPTIME.*

Proof 1) We prove the Π_2^{P}-hardness by reduction from the problem of checking the satisfiability of $\forall^*\exists^*$QBF formulas, a classical Π_2^{P}-complete problem. We take a $\forall^*\exists^*$QBF formula

$$\psi = \forall x_1, \ldots, x_n.\ \exists y_1, \ldots, y_m.\ \varphi,$$

where $\varphi = c_1 \wedge \ldots \wedge c_k$ is a quantifier-free CNF formula. We call the variables $x_1, \ldots, x_n$ *universal* and the variables $y_1, \ldots, y_m$ *existential*.

We take the alphabet $\{d_1, \ldots, d_k, t_1, f_1, \ldots, t_n, f_n\}$ and we construct two expressions, E_ψ and E'_ψ. First, $E_\psi = d_1 \parallel \ldots \parallel d_k \parallel X_1 \parallel \ldots \parallel X_n$, where for $1 \leqslant i \leqslant n$ $X_i = ((t_i \parallel d_{a_1} \parallel \ldots \parallel d_{a_l}) \mid (f_i \parallel d_{b_1} \parallel \ldots \parallel d_{b_s}))$, and $d_{a_1}, \ldots d_{a_l}$ (with $1 \leqslant a_1, \ldots, a_l \leqslant k$) correspond to the clauses which use the literal x_i, and $d_{b_1}, \ldots, d_{b_s}$ (with $1 \leqslant b_1, \ldots, b_s \leqslant k$) correspond to the clauses which use the literal $\neg x_i$. For example, for the formula

$$\psi_0 = \forall x_1, x_2.\ \exists y_1, y_2.\ (x_1 \vee \neg x_2 \vee y_1) \wedge (\neg x_1 \vee y_1 \vee \neg y_2) \wedge (x_2 \vee \neg y_1),$$

we construct:

$$E_{\psi_0} = d_1 \parallel d_2 \parallel d_3 \parallel ((t_1 \parallel d_1) \mid (f_1 \parallel d_2)) \parallel ((t_2 \parallel d_3) \mid (f_2 \parallel d_1)).$$

Note that there is an one-to-one correspondence between the unordered words in $L(E_\psi)$ and the valuations of the universal variables. For example, given the formula ψ_0, the unordered word $d_1^3 d_2 d_3 t_1 f_2$ corresponds to the valuation V such that $V(x_1) = \mathit{true}$ and $V(x_2) = \mathit{false}$.

Next, we construct $E'_\psi = X_1 \parallel \ldots \parallel X_n \parallel Y_1 \parallel \ldots \parallel Y_m$, where:

- $X_i = ((t_i \parallel d^*_{a_1} \parallel \ldots \parallel d^*_{a_l}) \mid (f_i \parallel d^*_{b_1} \parallel \ldots \parallel d^*_{b_s}))$, and $d_{a_1}, \ldots d_{a_l}$ (with $1 \leqslant a_1, \ldots, a_l \leqslant k$) correspond to the clauses which use the literal x_i, and $d_{b_1}, \ldots, d_{b_s}$ (with $1 \leqslant b_1, \ldots, b_s \leqslant k$) correspond to the clauses which use the literal $\neg x_i$ (for $1 \leqslant i \leqslant n$),
- $Y_j = ((d^*_{a_1} \parallel \ldots \parallel d^*_{a_l}) \mid (d^*_{b_1} \parallel \ldots \parallel d^*_{b_s}))$, and $d_{a_1}, \ldots d_{a_l}$ (with $1 \leqslant a_1, \ldots, a_l \leqslant k$) correspond to the clauses which use the literal y_j, and $d_{b_1}, \ldots, d_{b_s}$ (with $1 \leqslant b_1, \ldots, b_s \leqslant k$) correspond to the clauses which use the literal $\neg y_j$ (for $1 \leqslant j \leqslant m$).

For example, for ψ_0 above we construct:

$$E'_{\psi_0} = ((t_1 \parallel d_1^*) \mid (f_1 \parallel d_2^*)) \parallel ((t_2 \parallel d_3^*) \mid (f_2 \parallel d_1^*)) \parallel ((d_1^* \parallel d_2^*) \mid d_3^*) \parallel (\epsilon \mid d_2^*).$$

We claim that $\models \psi$ iff $E_\psi \subseteq E'_\psi$. For the *only if* case, for each valuation of the universal variables, we take the corresponding unordered word $w \in L(E_\psi)$. Since there exists a valuation of the existential variables which satisfies φ, we use this valuation to construct a derivation of w in $L(E'_\psi)$. For the *if* case, for every unordered word from $L(E_\psi)$, we take its derivation in $L(E'_\psi)$ and we use it to construct a valuation of the existential variables which satisfies φ. Clearly, the described reduction works in polynomial time.

2) The membership of the problem to 3-EXPTIME follows from the complexity of deciding the satisfiability of Presburger logic formulas, which is in 3-EXPTIME [Opp78]. Given two UREs E_1 and E_2, we compute in linear time [SSM08] two existential Presburger formulas for their Parikh images: φ_{E_1} and φ_{E_2}, respectively. Next, we test the satisfiability of the following closed Presburger logic formula: $\forall \overline{x}.\ \varphi_{E_1}(\overline{x}) \Rightarrow \varphi_{E_2}(\overline{x})$. ▫

While the complexity gap for the containment of UREs (as in Theorem 1.3.2) is currently quite important, we believe that this gap may be reduced by working on quantifier elimination for the Presburger formula obtained by translating the containment of UREs (as shown in the second part of the proof of Theorem 1.3.2). Although we believe that this problem is Π_2^{P}-complete, its exact complexity remains an open question.

1.3.3 Disallowing repetitions

The proofs of Theorem 1.3.1 and Theorem 1.3.2 rely on UREs allowing repetitions of the same symbol, which might be one of the causes of the intractability. Consequently, from now on we disallow repetitions of the same symbol in a URE. Similar restrictions are commonly used for the regular expressions to maintain practical aspects: *single occurrence regular expressions* (*SOREs*) [BNSV10], *conflict-free types* [CGPS13, CGS09, GCS08], and *duplicate-free DTDs* [MWM07]. While the complexity of the membership of an unordered word to the language of a URE without symbol repetitions has recently been shown to be in PTIME [SBG+15], testing containment of two such UREs continues to be intractable.

Theorem 1.3.3 *Given two UREs E_1 and E_2 not allowing repetitions of symbols, deciding $L(E_1) \subseteq L(E_2)$ is coNP-hard.*

Proof We show the coNP-hardness by reduction from the complement of 3SAT. Take a 3CNF formula $\varphi = c_1 \wedge \ldots \wedge c_k$ over the variables $\{x_1, \ldots, x_n\}$. We assume w.l.o.g. that each variable occurs at most once in a clause. Take the alphabet $\{a_{ij} \mid 1 \leqslant i \leqslant k, 1 \leqslant j \leqslant n, c_i \text{ uses } x_j \text{ or } \neg x_j\}$. We construct the expression $E_\varphi = X_1 \,\|\, \ldots \,\|\, X_n$, where $X_j = ((a_{t_1 j} \,\|\, \ldots \,\|\, a_{t_l j}) \mid (a_{f_1 j} \,\|\, \ldots \,\|\, a_{f_m j})$

(for $1 \leqslant j \leqslant n$), and $c_{t_1}, \ldots, c_{t_l}$ (with $1 \leqslant t_1, \ldots, t_l \leqslant k$) are the clauses which use the literal x_j, and $c_{f_1}, \ldots, c_{f_m}$ (with $1 \leqslant f_1, \ldots, f_m \leqslant k$) are the clauses which use the literal $\neg x_j$. Next, we construct $E'_\varphi = (C_1 \mid \ldots \mid C_k)^{[0,k-1]}$, where $C_i = (a_{ij_1} \mid \ldots \mid a_{ij_p})^+$ (for $1 \leqslant i \leqslant k$), and $x_{j_1}, \ldots, x_{j_p}$ (with $1 \leqslant j_1, \ldots, j_p \leqslant n$) are the variables used by the clause c_i. For example, for

$$\varphi_0 = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee \neg x_4),$$

we obtain:

$$\begin{aligned} E_{\varphi_0} =& (a_{11} \mid a_{21}) \parallel (a_{32} \mid a_{12}) \parallel ((a_{13} \parallel a_{23}) \mid a_{33}) \parallel (\epsilon \mid (a_{24} \parallel a_{34})), \\ E'_{\varphi_0} =& ((a_{11} \mid a_{12} \mid a_{13})^+ \mid (a_{21} \mid a_{23} \mid a_{24})^+ \mid (a_{32} \mid a_{33} \mid a_{34})^+)^{[0,2]}. \end{aligned}$$

Note that there is an one-to-one correspondence between the unordered words w_V in $L(E_\varphi)$ and the valuations V of the variables $x_1, \ldots, x_n$ (*). For example, for above φ_0 and the valuation V such that $V(x_1) = V(x_2) = V(x_3) = \mathit{true}$ and $V(x_4) = \mathit{false}$, the unordered word $w_V = a_{11}a_{32}a_{13}a_{23}a_{24}a_{34}$ is in $L(E_{\varphi_0})$. Moreover, given an $w_V \in L(E_\varphi)$, one can easily obtain the valuation.

We observe that the interval $[0, k-1]$ is used above a disjunction of k expressions of the form C_i and there is no repetition of symbols among the expressions of the form C_i. This allows us to state an instrumental property (**): $w \in L(E'_\varphi)$ iff there exists an $i \in \{1, \ldots, k\}$ such that none of the symbols used in C_i occurs in w. From (*) and (**), we infer that given a valuation V, $V \models \varphi$ iff $w_V \in L(E_\varphi) \setminus L(E'_\varphi)$, that yields $\varphi \in$ 3SAT iff $L(E_\varphi) \not\subseteq L(E'_\varphi)$. Clearly, the described reduction works in polynomial time. ▫

Theorem 1.3.3 shows that disallowing repetitions of symbols in a URE does not avoid the intractability of the containment. Additionally, we observe that the proof of Theorem 1.3.3 employs UREs with arbitrary use of disjunction and intervals. Consequently, in the next section we impose further restrictions that yield a class of UREs with desirable computational properties.

1.4 Disjunctive interval multiplicity expressions (DIMEs)

In this section, we present the DIMEs, a subclass of UREs for which membership and containment become tractable. First, we present an intuitive representation of DIMEs with characterizing tuples (Section 1.4.1). Next, we formally define DIMEs and show that they are precisely captured by

their characterizing tuples (Section 1.4.2). Finally, we use a compact representation of the characterizing tuples to show the tractability of DIMEs (Section 1.4.3).

1.4.1 Characterizing tuples

In this section, we introduce the notion of *characterizing tuple* that is an alternative, more intuitive representation of DIMEs, the subclass of UREs that we formally define in Section 1.4.2. Recall that by $a \in w$ we denote $w(a) \neq 0$. Given a DIME E, the *characterizing tuple* $\Delta_E = (C_E, N_E, P_E, K_E)$ is as follows.

- The *conflicting pairs of siblings* C_E consisting of all pairs of symbols in Σ such that E defines no word using both symbols simultaneously:
$$C_E = \{(a, b) \in \Sigma \times \Sigma \mid \nexists w \in L(E).\ a \in w \wedge b \in w\}.$$
- The *extended cardinality map* N_E capturing for each symbol in the alphabet the possible numbers of its occurrences in the unordered words defined by E:
$$N_E = \{(a, w(a)) \in \Sigma \times \mathbb{N}_0 \mid w \in L(E)\}.$$
- The *collections of required symbols* P_E capturing symbols that must be present in every word; essentially, a set of symbols X belongs to P_E if every word defined by E contains at least one element from X:
$$P_E = \{X \subseteq \Sigma \mid \forall w \in L(E).\ \exists a \in X.\ a \in w\}.$$
- The *counting dependencies* K_E consisting of pairs of symbols (a, b) such that in every word defined by E, the number of bs is at most the number of as. Note that if both (a, b) and (b, a) belong to K_E, then all unordered words defined by E should have the same number of a's and b's.
$$K_E = \{(a, b) \in \Sigma \times \Sigma \mid \forall w \in L(E).\ w(a) \geqslant w(b)\}.$$

As an example we take $E_0 = a^+ \parallel ((b \parallel c^?)^+ \mid d^{[5,\infty]})$ and we illustrate its characterizing tuple Δ_{E_0}. Because P_E is closed under supersets, we list only its minimal elements:

$C_{E_0} = \{(b, d), (c, d), (d, b), (d, c)\}$,
$N_{E_0} = \{(a, i) \mid i \geqslant 1\} \cup \{(b, i) \mid i \geqslant 0\} \cup \{(c, i) \mid i \geqslant 0\} \cup \{(d, i) \mid i = 0 \vee i \geqslant 5\}$,
$P_{E_0} = \{\{a\}, \{b, d\}, \ldots\}$,
$K_{E_0} = \{(b, c)\}$.

We point out that N_E may be infinite and P_E exponential in the size of E. Later on we discuss how to represent both sets in a compact manner while allowing efficient manipulation.

Then, an unordered word w *satisfies* a characterizing tuple Δ_E corresponding to a DIME E, denoted $w \models \Delta_E$, if the following conditions are satisfied:

1. $w \models C_E$ i.e., $\forall (a,b) \in C_E.\ (a \in w \Rightarrow b \notin w) \wedge (b \in w \Rightarrow a \notin w)$,
2. $w \models N_E$ i.e., $\forall a \in \Sigma.\ (a, w(a)) \in N_E$,
3. $w \models P_E$ i.e., $\forall X \in P_E.\ \exists a \in X.\ a \in w$,
4. $w \models K_E$ i.e., $\forall (a,b) \in K_E.\ w(a) \geqslant w(b)$.

For instance, the unordered word *aabbc* satisfies the characterizing tuple Δ_{E_0} corresponding to the aforementioned DIME $E_0 = a^+ \,\|\, ((b \,\|\, c^?)^+ \mid d^{[5,\infty]})$ since it satisfies all the four conditions imposed by Δ_{E_0}. On the other hand, note that the following unordered words do not satisfy Δ_{E_0}:

- *abddddd* because it contains at the same time b and d, and $(b,d) \in C_{E_0}$,
- *add* because it has two d's and $(d,2) \notin N_{E_0}$,
- *aa* because it does not contain any b or d and $\{b,d\} \in P_{E_0}$,
- *abbccc* because it has more c's than b's and $(b,c) \in K_{E_0}$.

In the next section, we define the DIMEs and show that they are precisely captured by characterizing tuples.

1.4.2 Grammar of DIMEs

An *atom* is $(a_1^{I_1} \,\|\, \ldots \,\|\, a_k^{I_k})$, where all I_i's are ? or 1. For example, $(a \,\|\, b^? \,\|\, c)$ is an atom, but $(a^{[3,4]} \,\|\, b)$ is not an atom. A *clause* is $(A_1^{I_1} \mid \ldots \mid A_k^{I_k})$, where all A_i's are atoms and all I_i's are intervals. A clause is *simple* if all I_i's are ? or 1. For example, $(a^{[2,3]} \mid (b^? \,\|\, c)^*)$ is a clause (which is not simple), $((a^? \,\|\, b) \mid c^?)$ is a simple clause while $((a^? \,\|\, b^+) \mid c)$ is not a clause.

A *disjunctive interval multiplicity expression* (DIME) is $(D_1^{I_1} \,\|\, \ldots \,\|\, D_k^{I_k})$, where for $1 \leqslant i \leqslant k$ either *1)* D_i is a simple clause and $I_i \in \{+, *\}$, or *2)* D_i is a clause and $I_i \in \{1, ?\}$. Moreover, a symbol can occur at most once in a DIME. For example, $(a \mid (b \| c^?)^+) \| (d^{[3,4]} \mid e^*)$ is a DIME while $(a \| b^?)^+ \| (a \mid c)$ is not a DIME because it uses the symbol a twice. A *disjunction-free interval multiplicity expression* (IME) is a DIME which does not use the disjunction

operator. An example of IME is $a \parallel (b \parallel c^?)^+ \parallel d^{[3,4]}$. For more practical examples of DIMEs see Examples 1.5.2 and 1.5.3 from Section 1.5.

We have tailored DIMEs to be able to capture them with characterizing tuples that permit deciding membership and containment in polynomial time (cf. Section 1.4.3). As we have already pointed out Section 1.3.3, a slightly more relaxed restriction on the nesting of disjunction and intervals leads to intractability of the containment (Theorem 1.3.3). Even though DIMEs may look very complex, the imposed restrictions are necessary to obtain lower complexity while considering fragments with practical relevance (cf. Section 1.8).

Next, we show that each DIME can be rewritten as an equivalent *reduced DIME*. Reduced DIMEs may also seem complex, but they are a building block for (i) proving that the language of a DIME is precisely captured by its characterizing tuple (Lemma 1.4.1), and (ii) computing the compact representation of the characterizing tuples that yield the tractability of DIMEs (cf. Section 1.4.3).

Before defining the reduced DIMEs, we need to introduce some additional notations. Given an atom A (resp. a clause D), we denote by Σ_A (resp. Σ_D) the set of symbols occurring in A (resp. D). Given a DIME E, by I^a_E (resp. I^A_E or I^D_E) we denote the interval associated in E to the symbol a (resp. atom A or clause D). Because we consider only expressions without repetitions, this interval is well-defined. Moreover, if E is clear from the context, we write simply I^a (resp. I^A or I^D) instead of I^a_E (resp. I^A_E or I^D_E). Furthermore, given an interval I which can be either $[n, m]$ or $[n, m]^?$, by $I^?$ we understand the interval $[n, m]^?$. In a reduced DIME E, each clause with interval D^I has one of the following three types:

1. $D^I = (A_1 \mid \ldots \mid A_k)^+$, where $k \geqslant 2$ and, for every $i \in \{1, \ldots, k\}$, A_i is an atom such that there exists $a \in \Sigma_{A_i}$ such that $I^a = 1$.

   For example, $((a \parallel b^?) \mid c)^+$ has type 1, but a^+ and $((a^? \parallel b^?) \mid c)^+$ do not.

2. $(A_1^{I_1} \mid \ldots \mid A_k^{I_k})$, where for every $i \in \{1, \ldots, k\}$ *1)* A_i is an atom such that there exists $a \in \Sigma_{A_i}$ such that $I^a = 1$ and *2)* 0 does not belong to the set represented by the interval I_i.

   For example, $(a \mid (b^? \parallel c)^{[5,\infty]})$ and a^+ have type 2, but $(a \mid (b^? \parallel c^?)^{[5,\infty]})$ and $(a^* \mid (b^? \parallel c)^{[5,\infty]})$ do not.

3. $(A_1^{I_1} \mid \ldots \mid A_k^{I_k})$, where for every $i \in \{1, \ldots, k\}$ A_i is an atom and I_i is an interval such that 0 belongs to the set represented by the interval I_i.

For example, $(a^* \mid (b \,\|\, c)^{[3,4]^?})$ and $(a^? \,\|\, b^?)^*$ have type 3, but $(a^? \,\|\, b^?)^{[3,4]}$ does not.

The reduced DIMEs easily yield the construction of their characterizing tuples. Take a clause with interval D^I from a DIME E and observe that the symbols from Σ_D are present in the characterizing tuple Δ_E as follows.

- If D^I is of type 1, then there is no symbol in Σ_D that occurs in a conflict in C_E. Otherwise, C_E consists of all pairs of distinct symbols (a, b) from Σ_D that appear in different atoms from D^I.
- If D^I is of type 1, then we have $(a, n) \in N_E$ for every $(a, n) \in \Sigma_D \times \mathbb{N}_0$. Otherwise, the possible number of occurrences of every symbol a from Σ_D can be obtained directly from the two intervals above it: the interval of D and the interval of the atom containing a. We explain in Section 1.4.3 how to precisely construct a compact representation of the potentially infinite set N_E.
- If D^I is of type 1 or 2, then every unordered word defined by E contains at least one of the symbols a from Σ_D having interval $I^a = 1$. More precisely, P_E contains all sets of symbols $X \subseteq \Sigma$ containing, for every atom of D, at least one symbol a with $I^a = 1$. For example, for $((a \,\|\, b \,\|\, c^?) \mid (d \,\|\, e))^+$, P_E consists of the sets $\{a, d\}, \{a, e\}, \{b, d\}, \{b, e\}$ and all their supersets. Otherwise, if D^I is of type 3, then there is no set in P_E containing only symbols from Σ_D.
- Regardless of the type of D^I, the counting dependencies K_E consist of all pairs of symbols (a, b) such that they appear in the same atom in D and $I^a = 1$.

To obtain reduced DIMEs, we use the following rules:

- Take a simple clause $(A_1^{I_1} \mid \ldots \mid A_k^{I_k})$.
  - $(A_1^{I_1} \mid \ldots \mid A_k^{I_k})^*$ goes to $A_1^* \,\|\, \ldots \,\|\, A_k^*$ (k clauses of type 3). Essentially, we distribute the $*$ of a disjunction of atoms with intervals to each of the atoms. For example, $(a \mid (b \,\|\, c^?))^*$ goes to $a^* \,\|\, (b \,\|\, c^?)^*$.
  - $(A_1^{I_1} \mid \ldots \mid A_k^{I_k})^+$ goes to $A_1^* \,\|\, \ldots \,\|\, A_k^*$ (k clauses of type 3) if there exists an atom with interval $A_i^{I_i}$ ($i \in \{1, \ldots, k\}$) that defines the empty word i.e., $I_i =?$ or $I^a =?$ for every symbol $a \in \Sigma_{A_i}$. If the empty word is defined, then we can basically transform the $+$ into $*$ and then distribute the $*$ as for the previous case. For example, $((a \,\|\, b^?) \mid (c \,\|\, d)^?)^+$ goes to $(a \,\|\, b^?)^* \,\|\, (c \,\|\, d)^*$.

- Take a clause $(A_1^{I_1} \mid \ldots \mid A_k^{I_k})$.
  - $(A_1^{I_1} \mid \ldots \mid A_k^{I_k})^?$ goes to $(A_1^{I_1^?} \mid \ldots \mid A_k^{I_k^?})$ (type 3). We essentially distribute the ? of a disjunction of atoms with intervals to each of the atoms. For example, $(a^{[2,3]} \mid b^+)^?$ goes to $(a^{[2,3]^?} \mid b^*)$.
  - $(A_1^{I_1} \mid \ldots \mid A_k^{I_k})$ goes to $(A_1^{I_1^?} \mid \ldots \mid A_k^{I_k^?})$ (type 3) if there exists an atom with interval $A_i^{I_i}$ $(i \in \{1, \ldots, k\})$ that defines the empty word i.e., 0 belongs to the set represented by I_i or $I^a = ?$ for every symbol $a \in \Sigma_{A_i}$. If the empty word is defined by one of the atoms, then we can basically distribute ? to all of them. For example, $(a \mid (b \parallel c)^{[0,5]})$ goes to $(a^? \mid (b \parallel c)^{[0,5]})$.
- Take an atom $(a_1^? \parallel \ldots \parallel a_k^?)$ and an interval I. Then, $(a_1^? \parallel \ldots \parallel a_k^?)^I$ goes to $(a_1^? \parallel \ldots \parallel a_k^?)^{[0,\max(I)]}$, where by $\max(I)$ we denote the maximum value from the set represented by the interval I. This step may be combined with one of the previous ones to rewrite a clause with interval as one of type 3. For example, $((a^? \parallel b^?)^{[3,6]} \mid c)$ goes to $((a^? \parallel b^?)^{[0,6]} \mid c^?)$.
- Remove symbols a (resp. atoms A or clauses D) such that I^a (resp. I^A or I^D) is $[0, 0]$.

Note that each of the rewriting steps gives an equivalent reduced expression.

Next, we assume that we work with reduced DIMEs only and show that the language defined by a DIME E comprises of all unordered words satisfying the characterizing tuple Δ_E.

Lemma 1.4.1 *Given an unordered word w and a DIME E, $w \in L(E)$ iff $w \models \Delta_E$.*

Proof The *only if* part follows from the definition of the satisfiability of Δ_E. For the *if* part, we take the tuple Δ_E corresponding to a DIME $E = D_1^{I_1} \parallel \ldots \parallel D_k^{I_k}$ and an unordered word w such that $w \models \Delta_E$. Let $w = w_1 \uplus \ldots \uplus w_k \uplus w'$, where each w_i contains all occurrences in w of the symbols from Σ_{D_i} (for $1 \leqslant i \leqslant k$). Since $w \models N_E$, we infer that there is no symbol $a \in \Sigma \backslash (\Sigma_{D_1} \cup \ldots \cup \Sigma_{D_k})$ such that $a \in w$, which implies $w' = \varepsilon$. Thus, proving $w \models E$ reduces to proving that $w_i \models D_i^{I_i}$ (for $1 \leqslant i \leqslant k$). Since E is a reduced DIME, each derivation can be constructed by reasoning on the three possible types of the $D_i^{I_i}$ (for $1 \leqslant i \leqslant k$).

Case 1. Take $D_i^{I_i} = (A_1 \mid \ldots \mid A_k)^+$ of type 1. From the semantics of the UREs, we observe that proving $w_i \models D_i^{I_i}$ is equivalent to proving that (i) w_i is non-empty and (ii) w_i can be split as $w_i = w_1' \uplus \ldots \uplus w_p'$, where every w_j' $(1 \leqslant j \leqslant p)$ satisfies an atom A_l $(1 \leqslant l \leqslant k)$. First, we point out that since w

satisfies the collections of required symbols P_E, we infer that w_i is non-empty, which implies (i). Then, since w satisfies the extended cardinality map N_E and the counting dependencies K_E, we infer that (ii) is also satisfied.

Case 2. Take $D_i^{I_i} = (A_1^{I_1} \mid \ldots \mid A_k^{I_k})$ of type 2. From the semantics of UREs, we observe that proving $w_i \models D_i^{I_i}$ is equivalent to proving that (i) w_i is non-empty and (ii) there exists an atom with interval $A_j^{I_j}$ $(1 \leqslant j \leqslant k)$ such that $w_i \models A_j^{I_j}$. Since $w \models P_E$, we infer that w_i is non-empty hence (i) is satisfied. Then, since $w \models C_E$, we infer that only the symbols from one atom A_j of D_i are present in w_i. Moreover, since $w \models N_E$ and $w \models K_E$, we infer that the number of occurrences of each symbol from Σ_{A_j} are such that $w_i \models A_j^{I_j}$. Hence, the condition (ii) is also satisfied.

Case 3. Take $D_i^{I_i} = (A_1^{I_1} \mid \ldots \mid A_k^{I_k})$ of type 3. The only difference w.r.t. the previous case is that w_i may be also empty, hence proving $w_i \models D_i^{I_i}$ is equivalent to proving only that there exists an atom with interval $A_j^{I_j}$ $(1 \leqslant j \leqslant k)$ such that $w_i \models A_j^{I_j}$, which follows similarly to the previous case. ▫

Moreover, we define the *subsumption* of two characterizing tuples, which captures the containment of DIMEs. Given two DIMEs E and E', we write $\Delta_{E'} \preccurlyeq \Delta_E$ if $C_E \subseteq C_{E'}$, $N_{E'} \subseteq N_E$, $P_E \subseteq P_{E'}$, and $K_E \subseteq K_{E'}$. Then, we obtain the following.

Lemma 1.4.2 *Given two DIMEs E and E', $L(E') \subseteq L(E)$ iff $\Delta_{E'} \preccurlyeq \Delta_E$.*

Proof First, we claim that given two DIMEs E and E': $\Delta_{E'} \preccurlyeq \Delta_E$ iff $w \models \Delta_{E'}$ implies $w \models \Delta_E$ for every w (*). The *only if* part of (*) follows directly from the definitions while the *if* part can be easily shown by contraposition. From Lemma 1.4.1 and (*) we infer the correctness of Lemma 1.4.2. ▫

Example 1.4.3 For the following DIMEs, it holds that $L(E') \subsetneq L(E)$ and $L(E) \not\subseteq L(E')$:

- Take $E = a^* \parallel b^*$ and $E' = (a \parallel b^?)^*$. Note that $K_E = \varnothing$ and $K_{E'} = \{(a, b)\}$. For instance, the unordered word b belongs to $L(E)$, but does not belong to $L(E')$.
- Take $E = a^{[3,6]^?} \mid b^*$ and $E' = a^{[3,6]} \mid b^+$. Note that $P_E = \varnothing$, and $P_{E'} = \{\{a, b\}\}$. For instance, the unordered word ε belongs to $L(E)$, but does not belong to $L(E')$.
- Take $E = (a \parallel b^?)^*$ and $E' = (a \parallel b^?)^{[0,5]}$. Note that $(a, 6)$ belongs to N_E, but not to $N_{E'}$. For instance, the unordered word a^6 belongs to $L(E)$, but does not belong to $L(E')$.

- Take $E = (a \mid b)^+$ and $E' = a^+ \mid b^+$. Note that $C_E = \varnothing$, and $C_{E'} = \{(a, b), (b, a)\}$. For instance, the unordered word ab belongs to $L(E)$, but does not belong to $L(E')$. ▫

Lemma 1.4.2 shows that two equivalent DIMEs yield the same characterizing tuple, and hence, the tuple Δ_E can be viewed as a "canonical form" for the language defined by a DIME E. Formally, we obtain the following.

Corollary 1.4.4 *Given two DIMEs E and E', $L(E) = L(E')$ iff $\Delta_E = \Delta_{E'}$.*

In the next section, we show that the characterizing tuple has a compact representation that permits us to decide the problems of membership and containment in polynomial time.

1.4.3 Tractability of DIMEs

We now show that the characterizing tuple admits a compact representation that yields the tractability of deciding membership and containment of DIMEs.

Given a reduced DIME E, note that C_E and K_E are quadratic in $|\Sigma|$ and can be easily constructed. The set C_E consists of all pairs of distinct symbols (a, b) such that they appear in different atoms in the same clause of type 2 or 3. Moreover, K_E consists of all pairs of distinct symbols (a, b) such that they appear in the same atom and $I^a = 1$.

While N_E may be infinite, it can be easily represented in a compact manner using intervals: for every symbol a, the set $\{i \in \mathbb{N}_0 \mid (a, i) \in N_E\}$ is representable by an interval. Given a symbol $a \in \Sigma$, by $\hat{N}_E(a)$ we denote the interval representing the set $\{i \in \mathbb{N}_0 \mid (a, i) \in N_E\}$ that can be easily obtained from E:

- $\hat{N}_E(a) = [0, 0]$ if a appears in no clause in E,
- $\hat{N}_E(a) = [0, \infty]$ (or simply $*$) if a appears in a clause of type 1 in E,
- $\hat{N}_E(a) = I^A$ if $I^a = 1$, A is the atom containing a, and A is the unique atom of a clause of type 2 or 3,
- $\hat{N}_E(a) = I^{A^?}$ if $I^a = 1$, A is the atom containing a, and A appears in a clause of type 2 or 3 containing at least two atoms,
- $\hat{N}_E(a) = [0, \max(I^A)]$ if $I^a = ?$, A is the atom containing a, and A appears in a clause of type 2 or 3.

For example, for $E_0 = a^+ \parallel ((b \parallel c^?)^+ \mid d^{[5,\infty]})$, we obtain the following $\hat{N}_{E_0}$:

$$\hat{N}_{E_0}(a) = +, \qquad \hat{N}_{E_0}(b) = *, \qquad \hat{N}_{E_0}(c) = *, \qquad \hat{N}_{E_0}(d) = [5,\infty]^?.$$

Naturally, testing $N_{E'} \subseteq N_E$ reduces to a simple test on $\hat{N}_{E'}$ and $\hat{N}_E$.

Representing P_E in a compact manner is more tricky. A natural idea would be to store only its $\subseteq$-minimal elements since P_E is closed under supersets. Unfortunately, there exist DIMEs having an exponential number of $\subseteq$-minimal elements. For instance, for the DIME $E_1 = ((a \parallel b) \mid (c \parallel d))^+ \parallel ((e \parallel f)^{[2,5]} \mid g^{[1,3]}) \parallel (h^* \parallel i^{[0,9]})$, the set P_{E_1} has 6 $\subseteq$-minimal elements: $\{a,c\}$, $\{a,d\}$, $\{b,c\}$, $\{b,d\}$, $\{e,g\}$, and $\{f,g\}$. The example easily generalizes to arbitrary numbers of atoms used in the clauses.

However, we observe that the exponentially-many $\subseteq$-minimal elements may contain redundant information that is already captured by other elements of the characterizing tuple. For instance, for the above DIME E_1, if we know that $\{a,c\}$ belongs to P_E, we can easily see that other $\subseteq$-minimal elements also belong to P_E. More precisely, we observe that for every unordered word w defined by E it holds that $w(a) = w(b)$, $w(c) = w(d)$ and $w(e) = w(f)$, which is captured by the counting dependencies $K_E = \{(a,b),(b,a),(c,d),(d,c),(e,f),(f,e)\}$. Hence, for the unordered words defined by E, the presence of an a implies the presence of a b, the presence of a c implies the presence of a d, etc. Consequently, if $\{a,c\}$ belongs to P_E, then $\{b,c\}$, $\{a,d\}$, and $\{b,d\}$ also belong to P_E. Similarly, if $\{e,g\}$ belongs to P_E, then $\{f,g\}$ also belongs to P_E.

Next, we use the aforementioned observation to define a compact representation of P_E. For this purpose, we introduce the auxiliary notion of symbols *implied by a DIME E in the presence of a set of symbols X*, denoted $\mathit{impl}_E(X)$:

$$\mathit{impl}_E(X) = X \cup \{a \in \Sigma \mid \exists b \in X.\ (a,b) \in K_E \text{ and } (b,a) \in K_E\}.$$

For example, for the above E_1, we have $\mathit{impl}_E(\{a,c\}) = \{a,b,c,d\}$.

Moreover, given a DIME E, by $P_E^{\subseteq\text{min}}$ we denote the set of all $\subseteq$-minimal elements of P_E. Given a subset $P \subseteq P_E^{\subseteq\text{min}}$, we say that P is:

- *non-redundant* if $\forall X \in P.\ \nexists Y \in P.\ X \subseteq \mathit{impl}_E(Y)$,
- *covering* if $\forall X \in P_E^{\subseteq\text{min}}.\ \exists Y \in P.\ X \subseteq \mathit{impl}_E(Y)$.

For example, take the above $E_1 = ((a \parallel b) \mid (c \parallel d))^+ \parallel ((e \parallel f)^{[2,5]} \mid g^{[1,3]}) \parallel (h^* \parallel i^{[0,9]})$ and recall that $P_{E_1}^{\subseteq\text{min}} = \{\{a,c\},\{a,d\},\{b,c\},\{b,d\},\{e,g\},\{f,g\}\}$. Then, we have the following:

- $\{\{b, c\}, \{f, g\}\}$ is non-redundant and covering,
- $\{\{b, c\}\}$ is non-redundant and it is not covering,
- $\{\{a, c\}, \{b, c\}, \{f, g\}\}$ is redundant and covering,
- $\{\{a, c\}, \{b, c\}\}$ is redundant and not covering.

Given a DIME E, the *compact representation of the collections of required symbols* P_E is naturally a non-redundant and covering subset of $P_E^{\subseteq \min}$. Since there may exist many non-redundant and covering subsets of $P_E^{\subseteq \min}$, we use the total order $<_\Sigma$ on the alphabet Σ to propose a deterministic construction of the compact representation $\hat{P}_E$. For this purpose, we define first some additional notations.

Given an atom A, by $\Phi(A)$ we denote the smallest label from Σ w.r.t. $<_\Sigma$ that is present in A and has interval 1:

$$\Phi(A) = \min_{<_\Sigma}\{a \in \Sigma_A \mid I^a = 1\}.$$

For example, $\Phi(a \parallel b) = a$. Then, given a clause with interval D^I, by $\Phi(D^I)$ we denote the set of all symbols $\Phi(A)$ for every atom A in D:

$$\Phi(D^I) = \{\Phi(A) \mid A \text{ is an atom in } D\}.$$

For example, $\Phi(((a \parallel b) \mid (c \parallel d))^+) = \{a, c\}$ and $\Phi(((e \parallel f)^{[2,5]} \mid g^{[1,3]})) = \{e, g\}$. Then, $\hat{P}(E)$ consists of all such sets for the clauses with intervals of type 1 or 2:

$$\hat{P}_E = \{\Phi(D^I) \mid D^I \text{ is a clause with interval of type 1 or 2 in } E\}.$$

For example, $\hat{P}_{E_1} = \{\{a, c\}, \{e, g\}\}$. Notice that the set $\{a, c\}$ is due to the clause with interval $((a \parallel b) \mid (c \parallel d))^+$ of type 1 and the set $\{e, g\}$ is due to the clause with interval $((e \parallel f)^{[2,5]} \mid g^{[1,3]})$ of type 2. Also notice that the clause with interval $(h^* \parallel i^{[0,9]})$ is of type 3, none of its symbols is required, and consequently, no set in $\hat{P}_E$ contains symbols from it.

We have introduced all elements to be able to define the compact representation of a characterizing tuple. Given a DIME E, we say that $\hat{\Delta} = (C_E, \hat{N}_E, \hat{P}_E, K_E)$ is the *compact representation* of its characterizing tuple Δ_E. Then, an unordered word w satisfies $\hat{\Delta}_E$, denoted $w \models \hat{\Delta}_E$, if

- $w \models C_E$ and $w \models K_E$ as previously defined when we have introduced $w \models \Delta_E$,
- $w \models \hat{N}_E$ i.e., $\forall a \in \Sigma.\ w(a) \in \hat{N}_E(a)$,

- $w \models \hat{P}_E$ i.e., $\forall X \in \hat{P}_E.\ \exists a \in X.\ a \in w$. Notice that we use exactly the same definition as for $w \models P_E$ and recall that $\hat{P}_E$ is in fact a non-redundant and covering subset of $P_E^{\subseteq\min}$.

Next, we show that given a DIME E, its compact characterizing tuple $\hat{\Delta}_E$ defines precisely the same set of unordered words as its characterizing tuple Δ_E.

Lemma 1.4.5 *Given an unordered word w and a DIME E, $w \models \Delta_E$ iff $w \models \hat{\Delta}_E$.*

Proof The *only if* part follows directly from the definitions. For the *if* part, proving $w \models \Delta_E$ reduces to proving that $w \models P_E$, which moreover, reduces to proving that for every X from $P_E^{\subseteq\min}$ there is a symbol a in X that occurs in w (*). Since $\hat{P}_E$ is a covering subset of $P_E^{\subseteq\min}$, we know that for every $X \in P_E^{\subseteq\min}$ there exists a set $Y \in \hat{P}_E$ such that $X \subseteq \mathit{impl}_E(Y)$. Since $w \models \hat{P}_E$ and $w \models K_E$, we infer that (*) is satisfied. ▫

Additionally, we define the *subsumption* of the compact representations of two characterizing tuples. Given two DIMEs E and E', we write $\hat{\Delta}_{E'} \preccurlyeq \hat{\Delta}_E$ if

- $C_E \subseteq C_{E'}$ and $K_E \subseteq K_{E'}$ (as for the subsumption of characterizing tuples),
- $\forall a \in \Sigma.\ \hat{N}_{E'}(a) \subseteq \hat{N}_E(a)$,
- $\forall X \in \hat{P}_E.\ \exists Y \in \hat{P}_{E'}.\ Y \subseteq \mathit{impl}_{E'}(X)$.

Next, we show that the subsumption of compact representations of characterizing tuples captures the subsumption of characterizing tuples.

Lemma 1.4.6 *Given two DIMEs E and E', $\Delta_{E'} \preccurlyeq \Delta_E$ iff $\hat{\Delta}_{E'} \preccurlyeq \hat{\Delta}_E$.*

Proof First, since P_E is closed under supersets, we observe that

$$P_E \subseteq P_{E'} \text{ iff } \forall X \in P_E^{\subseteq\min}.\ \exists Y \in P_{E'}^{\subseteq\min}.\ Y \subseteq X.$$

Moreover, the conditions $C_E \subseteq C_{E'}$ and $K_E \subseteq K_{E'}$ are part of both $\Delta_{E'} \preccurlyeq \Delta_E$ and $\hat{\Delta}_{E'} \preccurlyeq \hat{\Delta}_E$. Consequently, proving $\Delta_{E'} \preccurlyeq \Delta_E$ iff $\hat{\Delta}_{E'} \preccurlyeq \hat{\Delta}_E$ reduces to proving that, if $C_E \subseteq C_{E'}$ and $K_E \subseteq K_{E'}$, then

$$\forall X \in P_E^{\subseteq\min}.\ \exists Y \in P_{E'}^{\subseteq\min}.\ Y \subseteq X \text{ iff } \forall X \in \hat{P}_E.\ \exists Y \in \hat{P}_{E'}.\ Y \subseteq \mathit{impl}_{E'}(X).$$

For the *only if* part, take a set X from $\hat{P}_E$. Since X also belongs to $P_E^{\subseteq\min}$, we know by hypothesis that there exists a set Y in $P_{E'}^{\subseteq\min}$ such that $Y \subseteq X$. Then, construct a set Y' from Y by replacing each symbol b from Y with the smallest a w.r.t. $<_\Sigma$ such that (a, b) and (b, a) belong to $K_{E'}$. Moreover, since $K_E \subseteq K_{E'}$, we infer that $Y' \subseteq \mathit{impl}_{E'}(X)$. For the *if* part, take an X from $\hat{P}_E$ and an Y from $\hat{P}_{E'}$ s.t. $Y \subseteq \mathit{impl}_{E'}(X)$. To construct the corresponding X' in $P_E^{\subseteq\min}$ and Y' in $P_{E'}^{\subseteq\min}$ such that $Y' \subseteq X'$, we replace symbols a from X and a' from Y with symbols b in X' and b' in Y' such that (a, b) and (b, a) belong to K_E, and (a', b') and (b', a') belong to $K_{E'}$. Since $K_E \subseteq K_{E'}$, we know that such X' and Y' do exist. □

Example 1.4.7 Take $E = a^* \parallel (b \mid c)^+ \parallel d^*$ and $E' = (a \parallel b)^+ \mid (c \parallel d)^+$. Notice that $L(E') \subseteq L(E)$, $\Delta_{E'} \preccurlyeq \Delta_E$, and $\hat{\Delta}_{E'} \preccurlyeq \hat{\Delta}_E$. In particular, we have the following.

- $C_E = \varnothing$ is included in $C_{E'} = \{(a, c), (a, d), (b, c), (b, d), (c, a), (c, b), (d, a), (d, b)\}$,
- $\hat{N}_E(a) = \hat{N}_{E'}(a) = *, \ldots, \hat{N}_E(d) = \hat{N}_{E'}(d) = *$,
- $K_E = \varnothing$ is included in $K_{E'} = \{(a, b), (b, a), (c, d), (d, c)\}$,
- $\hat{P}_E = \{\{b, c\}\}$ and $\hat{P}_{E'} = \{\{a, c\}\}$ that compactly represent $P_E = \{\{b, c\}, \ldots\}$ and $P_{E'} = \{\{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \ldots\}$, respectively (we have listed only the $\subseteq$-minimal sets). Then, take $X = \{b, c\}$ from $\hat{P}_E$ and notice that there exists $Y = \{a, c\}$ in $\hat{P}_{E'}$ such that $Y \subseteq \mathit{impl}_{E'}(X)$ because $\mathit{impl}_{E'}(\{b, c\}) = \{a, b, c, d\}$. □

Next, we show that the compact representation is of polynomial size.

Lemma 1.4.8 *Given a DIME E, the compact representation $\hat{\Delta}_E = (C_E, \hat{N}_E, \hat{P}_E, K_E)$ of its characterizing tuple Δ_E is of size polynomial in the size of the alphabet Σ.*

Proof By construction, the sizes of C_E and K_E are quadratic in $|\Sigma|$ while the sizes of $\hat{P}_E$ and $\hat{N}_E$ are linear in $|\Sigma|$. □

The use of compact representation of characterizing tuples allows us to state the main result of this section.

Theorem 1.4.9 *Given an unordered word w and two DIMEs E and E':*

1. *deciding whether $w \in L(E)$ is in PTIME,*
2. *deciding whether $L(E') \subseteq L(E)$ is in PTIME.*

Proof The first part follows from Lemma 1.4.1, Lemma 1.4.5, and Lemma 1.4.8. The second part follows from Lemma 1.4.2, Lemma 1.4.6, and Lemma 1.4.8.

□

1.5 Interval multiplicity schemas

In this section, we employ DIMEs to define schema languages and we present the related problems of interest.

Definition 1.5.1 *A* disjunctive interval multiplicity schema (DIMS) *is a tuple $S = (root_S, R_S)$, where $root_S \in \Sigma$ is a designated root label and R_S maps symbols in Σ to DIMEs. By DIMS we denote the set of all disjunctive interval multiplicity schemas. A* disjunction-free interval multiplicity schema (IMS) *$S = (root_S, R_S)$ is a restricted DIMS, where R_S maps symbols in Σ to IMEs. By IMS we denote the set of all disjunction-free interval multiplicity schemas.*

We define the language captured by a DIMS S in the following way. Given a tree t, we first define the unordered word ch_t^n of children of a node $n \in N_t$ of t i.e., $ch_t^n(a) = |\{m \in N_t \mid (n,m) \in child_t \wedge lab_t(m) = a\}|$. Now, a tree t *satisfies* S, in symbols $t \models S$, if $lab_t(root_t) = root_S$ and for every node $n \in N_t$, $ch_t^n \in L(R_S(lab_t(n)))$. By $L(S) \subseteq Tree$ we denote the set of all trees satisfying S.

In the sequel, we present a schema $S = (root_S, R_S)$ as a set of rules of the form $a \to R_S(a)$, for every $a \in \Sigma$. If $L(R_S(a)) = \varepsilon$, then we write $a \to \epsilon$ or we simply omit writing such a rule.

Example 1.5.2 Take the content model of a semi-structured database storing information about a peer-to-peer file sharing system, having the following rules: *1)* a peer is allowed to download at most the same number of files that it uploads, and *2)* peers are split into two groups: a peer is a *vip* if it uploads at least 100 files, otherwise it is a simple *user*:

$$\begin{aligned} peers &\to user^* \parallel vip^*, \\ user &\to (upload \parallel download^?)^{[0,99]}, \\ vip &\to (upload \parallel download^?)^{[100,\infty]}. \end{aligned}$$ □

Example 1.5.3 Take the content model of a semi-structured database storing information about two types of cultural events: plays and movies. Every event has a date when it takes place. If the event is a play, then it takes place in a theater while a movie takes place in a cinema.

$$\begin{aligned} events &\to event^*, \\ event &\to date \parallel ((play \parallel theater) \mid (movie \parallel cinema)). \end{aligned}$$ □

Problems of interest. We define next the problems of interest and we formally state the corresponding decision problems parameterized by the class of schema $\mathcal{S}$ and, when appropriate, by a class of queries $\mathcal{Q}$.

- *Schema satisfiability* – checking if there exists a tree satisfying the given schema:
$$\mathrm{SAT}_{\mathcal{S}} = \{S \in \mathcal{S} \mid \exists t \in \mathit{Tree}.\ t \models S\}.$$
- *Membership* – checking if the given tree satisfies the given schema:
$$\mathrm{MEMB}_{\mathcal{S}} = \{(S, t) \in \mathcal{S} \times \mathit{Tree} \mid t \models S\}.$$
- *Schema containment* – checking if every tree satisfying one given schema satisfies another given schema:
$$\mathrm{CNT}_{\mathcal{S}} = \{(S_1, S_2) \in \mathcal{S} \times \mathcal{S} \mid L(S_1) \subseteq L(S_2)\}.$$
- *Query satisfiability by schema* – checking if there exists a tree that satisfies the given schema and the given query:
$$\mathrm{SAT}_{\mathcal{S},\mathcal{Q}} = \{(S, q) \in \mathcal{S} \times \mathcal{Q} \mid \exists t \in L(S).\ t \models q\}.$$
- *Query implication by schema* – checking if every tree satisfying the given schema satisfies also the given query:
$$\mathrm{IMPL}_{\mathcal{S},\mathcal{Q}} = \{(S, q) \in \mathcal{S} \times \mathcal{Q} \mid \forall t \in L(S).\ t \models q\}.$$
- *Query containment in the presence of schema* – checking if every tree satisfying the given schema and one given query also satisfies another given query:
$$\mathrm{CNT}_{\mathcal{S},\mathcal{Q}} = \{(p, q, S) \in \mathcal{Q} \times \mathcal{Q} \times \mathcal{S} \mid \forall t \in L(S).\ t \models p \Rightarrow t \models q\}.$$

Next, we study these problems for DIMSs and IMSs in Sections 1.6 and 1.7.

1.6 Complexity of disjunctive interval multiplicity schemas

In this section, we present the complexity results for DIMSs. First, we show the tractability of schema satisfiability and containment. Then, we provide an algorithm for deciding membership in *streaming* i.e., that processes an

XML document in a single pass and using memory depending on the height of the tree and not on its size. Finally, we point out that the complexity of query satisfiability, implication, and containment in the presence of the schema follow from existing results.

First, we show the tractability of schema satisfiability and schema containment.

Proposition 1.6.1 SAT_{DIMS} *and* CNT_{DIMS} *are in PTIME.*

Proof A simple algorithm based on dynamic programming can decide the satisfiability of a DIMS. More precisely, given a schema $S = (root_S, R_S)$, one has to determine for every symbol a of the alphabet Σ whether there exists a (finite) tree t that satisfies $S' = (a, R_S)$. Then, the schema S is satisfiable if there exist such a tree for the root label $root_S$.

Moreover, testing the containment of two DIMSs reduces to testing, for each symbol in the alphabet, the containment of the associated DIMEs, which is in PTIME (Theorem 1.4.9). □

Next, we provide an algorithm for deciding membership in *streaming* i.e., that processes an XML document in a single pass and uses memory depending on the height of the tree and not on its size. Our notion of streaming has been employed in [SV02] as a relaxation of the constant-memory XML validation against DTDs, which can be performed only for some DTDs [SV02, SS07]. In general, validation against DIMSs cannot be performed with constant memory due to the same observations as in [SV02, SS07] w.r.t. the use of recursion in the schema. Hence, we have chosen our notion of streaming to be able to have an algorithm that works for the entire class of DIMSs. We assume that the input tree is given in XML format, with arbitrary ordering of sibling nodes. Moreover, the proposed algorithm has *earliest rejection* i.e., if the given tree does not satisfy the given schema, the algorithm outputs the result as early as possible. For a tree t, $height(t)$ is the height of t defined in the usual way. We employ the standard RAM model and assume that subsequent natural numbers are used as labels in Σ.

Proposition 1.6.2 MEMB_{DIMS} *is in PTIME. There exists an earliest rejection streaming algorithm that checks membership of a tree t in a DIMS S in time $O(|t| \times |\Sigma|^2)$ and using space $O(height(t) \times |\Sigma|^2)$.*

Proof We propose Algorithm 1 for deciding the membership of a tree t to the language of a DIMS S. The input tree t is given in XML format, with some arbitrary ordering of sibling nodes. We assume a well-formed stream $\widetilde{t} \subset \{open, close\} \times \Sigma$ representing a tree t and a procedure $read(\widetilde{t})$ that

returns the next pair (θ, b) in the stream, where $\theta \in \{open, close\}$ and $b \in \Sigma$. The algorithm works for every arbitrary ordering of sibling nodes. To validate a tree t against a DIMS $S = (root_S, R_S)$, one has to run Algorithm 1 after reading the opening tag of the root.

For a given node, the algorithm constructs the compact representation of the characterizing tuple of its label (line 1), which requires space $O(|\Sigma|^2)$ (cf. Lemma 1.4.8). The algorithm also stores for a given node the number of occurrences of each label in Σ among its children. This is done using the array *count*, which requires space $O(\Sigma)$. Initially, all values in the array *count* are set at 0 (lines 2-3) and they are updated after reading the open tag of the children (lines 4-6). During the execution, the algorithm maintains a stack whose height is the depth of the currently visited node. Naturally, the bound on space required is $O(height(t) \times |\Sigma|^2)$.

The algorithm has earliest rejection since it rejects a tree as early as possible. More precisely, this can be done after reading the opening tag for nodes that violate the maximum value for the allowed cardinality for their label (lines 7-8) or violate some conflicting pair of siblings (lines 9-10). If it is not the case, the algorithm recursively validates the corresponding subtree (lines 11-12). After reading all children of the current node, the algorithm checks whether the components of the characterizing tuple are satisfied: the extended cardinality map (lines 14-15), the collections of required symbols (lines 16-17), and the counting dependencies (lines 18-19). Notice that since we have checked the conflicting pairs of siblings after reading each opening tag, we do not need to check them again after reading all children. However, we still need to check the extended cardinality map at this moment to see whether the number of occurrences of each label is in the allowed interval. When we have read the opening tag, we were able to reject only if the maximum value for the allowed number of occurrences has been already violated. As for the collections of required symbols and the counting dependencies, we are able to establish whether they are satisfied or not after reading all children. If none of the constraints imposed by the characterizing tuple is violated, the algorithm returns true (line 20). As we have already shown with Lemma 1.4.1 and Lemma 1.4.5, the compact representation of the characterizing tuple captures precisely the language of a given DIME. Consequently, the algorithm returns true after reading the root node iff the given tree satisfies the given schema. □

We continue with complexity results that follow from known facts. Query satisfiability for DTDs is NP-complete [BFG08] and we adapt the result for DIMSs.

Proposition 1.6.3 $\mathrm{SAT}_{DIMS,Twig}$ *is NP-complete.*

Algorithm 1 Streaming algorithm for testing membership.

algorithm *validate*(a)
Parameters: DIMS S, stream $\widetilde{t}$
Input: the label $a \in \Sigma$ of the current node
Output: *true* if the subtree rooted at the current node is valid w.r.t. S, *false* otherwise

1: **let** $(C, \hat{N}, \hat{P}, K)$ be the compact representation of the characterizing tuple of $R_S(a)$
2: **for** $b \in \Sigma$ **do**
3: **let** $count[b] = 0$
4: $(\theta, b) = read(\widetilde{t})$
5: **while** $\theta = open$ **do**
6: $count[b] := count[b] + 1$
7: **if** $count[b] > \max(\hat{N}(b))$ **then**
8: **return** *false*
9: **if** $\exists c \in \Sigma.\ (b, c) \in C \wedge count[c] \neq 0$ **then**
10: **return** *false*
11: **if** $validate(b) = false$ **then**
12: **return** *false*
13: $(\theta, b) = read(\widetilde{t})$
14: **if** $\exists b \in \Sigma.\ count[b] \notin \hat{N}(b)$ **then**
15: **return** *false*
16: **if** $\exists X \in \hat{P}.\ \forall b \in X.\ count[b] = 0$ **then**
17: **return** *false*
18: **if** $\exists (b, c) \in K.\ count[b] < count[c]$ **then**
19: **return** *false*
20: **return** *true*

Proof Proposition 4.2.1 from [BFG08] implies that satisfiability of twig queries in the presence of DTDs is NP-hard. We adapt the proof and we obtain the following reduction from SAT to $\text{SAT}_{DIMS,Twig}$: we take a CNF formula $\varphi = \bigwedge_{i=1}^{n} C_i$ over the variables $x_1, \ldots, x_m$, where each C_i is a disjunction of literals. We take $\Sigma = \{r, t_1, f_1, \ldots, t_m, f_m, c_1, \ldots, c_n\}$ and we construct:

- The DIMS S having the root label r and the rules:
  - $r \to (t_1 \mid f_1) \parallel \ldots \parallel (t_m \mid f_m)$,
  - $t_j \to c_{j_1} \| \ldots \| c_{j_k}$, where $c_{j_1}, \ldots, c_{j_k}$ correspond to the clauses using x_j (for $1 \leqslant j \leqslant m$),
  - $f_i \to c_{j_1} \| \ldots \| c_{j_k}$, where $c_{j_1}, \ldots, c_{j_k}$ correspond to the clauses using $\neg x_j$ (for $1 \leqslant j \leqslant m$).
- The twig query $q = r[//c_1] \ldots [//c_n]$.

For example, for the formula $\varphi_0 = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$ we obtain the DIMS S containing the rules:

$$r \to (t_1 \mid f_1) \parallel (t_2 \mid f_2) \parallel (t_3 \mid f_3) \parallel (t_4 \mid f_4),$$
$$t_1 \to c_1, \qquad f_1 \to c_2, \qquad t_2 \to \epsilon, \qquad f_2 \to c_1,$$
$$t_3 \to c_1 \parallel c_2, \qquad f_3 \to \epsilon, \qquad t_4 \to \epsilon, \qquad f_4 \to c_2.$$

and the query $q = /r[//c_1][//c_2]$. The formula φ is satisfiable iff $(S, q) \in \text{SAT}_{DIMS,Twig}$. The described reduction works in polynomial time in the size of the input formula.

For the NP upper bound, we reduce $\text{SAT}_{DIMS,Twig}$ to $\text{SAT}_{DTD,Twig}$ (i.e., the problem of satisfiability of twig queries in the presence of DTDs), known to be in NP (Theorem 4.4 from [BFG08]). Given a DIMS S, we construct a DTD D having the same root label as S and whose rules are obtained from the rules of S by replacing the unordered concatenation with standard (ordered) concatenation. Then, take a twig query q. We claim that there exists an (unordered) tree satisfying q and S iff there exists an (ordered) tree satisfying q and D. For the *if* part, take an ordered tree t satisfying q and D, remove the order to obtain an unordered tree t', and observe that t' satisfies S. For the *only if* part, take an unordered tree t satisfying q and S. From the construction of D, we infer that there exists an ordered tree t' (obtained via some ordering of the sibling nodes of t) satisfying both q and D. We recall that the twig queries disregard the relative order among the siblings.

□

The complexity results for query implication and query containment in the presence of DIMSs follow from the EXPTIME-completeness proof from [NS06] for twig query containment in the presence of DTDs.

Proposition 1.6.4 $\text{IMPL}_{DIMS,Twig}$ *and* $\text{CNT}_{DIMS,Twig}$ *are EXPTIME-complete.*

Proof The EXPTIME-hardness proof of twig containment in the presence of DTDs (Theorem 4.5 from [NS06]) has been done using a reduction from the *Two-player corridor tiling* problem and a technique introduced in [MS04]. In the proof from [NS06], when testing the containment $p \subseteq_S q$, p is chosen such that it satisfies every tree in S, hence $\text{IMPL}_{DTD,Twig}$ is also EXPTIME-complete. Furthermore, Lemma 3 in [MS04] can be adapted to twig queries and DIMS: for every $S \in DIMS$ and twig queries $q_0, q_1, \ldots, q_m$ there exists $S' \in DIMS$ and twig queries q and q' such that $q_0 \subseteq_S q_1 \cup \ldots \cup q_m$ iff $q \subseteq_{S'} q'$. Moreover, the DTD in [NS06] can be captured with a DIMS constructible in polynomial time: take the same reduction as in [NS06] and then replace the standard concatenation with unordered concatenation. Hence, we infer that $\text{CNT}_{DIMS,Twig}$ and $\text{IMPL}_{DIMS,Twig}$ are also EXPTIME-hard.

For the EXPTIME upper bound, we reduce $\text{CNT}_{DIMS,Twig}$ to $\text{CNT}_{DTD,Twig}$ (i.e., the problem of twig query containment in the presence of DTDs), known to be in EXPTIME (Theorem 4.4 from [NS06]). Given a DIMS S, we construct a DTD D having the same root label as S and whose rules are obtained from the rules of S by replacing the unordered concatenation with standard (ordered) concatenation. Then, take two twig queries p and q. We claim that $p \subseteq_S q$ iff $p \subseteq_D q$ and show the two parts by contraposition. For the *if* part, assume $p \not\subseteq_S q$, hence there exists an unordered tree t that satisfies q and S, but not p. From the construction of D, we infer that there exists an ordered tree t' (obtained via some ordering of the sibling nodes of t) that satisfies q and D, but not p. For the *only if* part, assume $p \not\subseteq_D q$, hence there exists an ordered tree t that satisfies q and D, but not p. By removing the order of t, we obtain an unordered tree t' that satisfies q and S, but not p. We recall that the twig queries disregard the relative order among the siblings. The membership of $\text{CNT}_{DIMS,Twig}$ to EXPTIME yields that $\text{IMPL}_{DIMS,Twig}$ is also in EXPTIME (it suffices to take as p the universal query). □

1.7 Complexity of disjunction-free interval multiplicity schemas

Although *query satisfiability* and *query implication* in the presence of schema are intractable for DIMSs, we prove that they become tractable for IMSs

(Section 1.7.4). We also show a considerably lower complexity for *query containment* in the presence of schema: coNP-completeness for IMSs instead of EXPTIME-completeness for DIMSs (Section 1.7.4). Additionally, we point out that our results for IMSs allow also to characterize the complexity of query implication and query containment in the presence of *disjunction-free DTDs* (i.e., restricted DTDs using regular expressions without disjunction operator), which, to the best of our knowledge, have not been previously studied (Section 1.7.5). To prove our results, we develop a set of tools that we present next: *dependency graphs* (Section 1.7.1), *generalized definition of embedding* (Section 1.7.2), *family of characteristic graphs* (Section 1.7.3).

1.7.1 Dependency graphs

Recall that IMSs use IMEs, which are essentially expressions of the form $A_1^{I_1} \parallel \ldots \parallel A_k^{I_k}$, where $A_1, \ldots, A_k$ are atoms, and $I_1, \ldots, I_k$ are intervals. Given an IME E, let $\mathit{symbols}^{\forall}(E)$ be the set of symbols present in all unordered words in $L(E)$, and $\mathit{symbols}^{\exists}(E)$ the set of symbols present in at least one unordered word in $L(E)$:

$$\mathit{symbols}^{\forall}(E) = \{a \in \Sigma \mid \forall w \in L(E).\ a \in w\},$$
$$\mathit{symbols}^{\exists}(E) = \{a \in \Sigma \mid \exists w \in L(E).\ a \in w\}.$$

Given an IME E, notice that $\mathit{symbols}^{\forall}(E) \subseteq \mathit{symbols}^{\exists}(E)$, and moreover, the sets $\mathit{symbols}^{\forall}(E)$ and $\mathit{symbols}^{\exists}(E)$ can be easily constructed from E. For example, given $E_0 = (a \parallel b^?)^{[5,6]} \parallel c^+$, we have $\mathit{symbols}^{\forall}(E_0) = \{a, c\}$ and $\mathit{symbols}^{\exists}(E_0) = \{a, b, c\}$.

Definition 1.7.1 *Given an IMS $S = (\mathit{root}_S, R_S)$, the* existential dependency graph *of S is the directed rooted graph $G_S^{\exists} = (\Sigma, \mathit{root}_S, E_S^{\exists})$ with the node set Σ, the distinguished root node root_S, and the set of edges $E_S^{\exists}$ such that $(a, b) \in E_S^{\exists}$ if $b \in \mathit{symbols}^{\exists}(R_S(a))$. Furthermore, the* universal dependency graph *of S is the directed rooted graph $G_S^{\forall} = (\Sigma, \mathit{root}_S, E_S^{\forall})$ such that $(a, b) \in E_S^{\forall}$ if $b \in \mathit{symbols}^{\forall}(R_S(a))$.*

Example 1.7.2 Take the IMS S containing the rules:

$$r \to (a^? \parallel b)^{[1,10]} \parallel c, \qquad a \to d^?, \qquad b \to a^{[2,3]} \parallel c^* \parallel d^+.$$

In Figure 1.4 we present the existential dependency graph of S and the universal dependency graph of S. ▫

Figure 1.4: Existential dependency graph $G_S^{\exists}$ and universal dependency graph $G_S^{\forall}$ for Example 1.7.2.

Given an IMS S and a symbol a, we say that a is *reachable* (or *useful*) in S if there exists a tree in $L(S)$ which has a node labeled by a. Moreover, we say that an IMS is *trimmed* if it contains rules only for the reachable symbols. For every satisfiable IMS S, there exists an equivalent trimmed version which can be obtained by removing the rules for the symbols involved in *unreachable components* in $G_S^{\forall}$ (in the spirit of [AGW01]). Notice that the unreachable components of $G_S^{\forall}$ correspond in fact to cycles in $G_S^{\forall}$. In the sequel, we assume w.l.o.g. that all IMSs that we manipulate are satisfiable and trimmed.

1.7.2 Generalizing the embedding

We generalize the notion of embedding previously defined in Section 1.2. Note that in the rest of the section we use the term *dependency graphs* when we refer to both existential and universal dependency graphs. First, an *embedding* of a query q in a dependency graph $G = (\Sigma, root, E)$ is a function $\lambda : N_q \to \Sigma$ such that:

1. $\lambda(root_q) = root$,
2. for every $(n, n') \in child_q$, $(\lambda(n), \lambda(n')) \in E$,
3. for every $(n, n') \in desc_q$, $(\lambda(n), \lambda(n')) \in E^+$ (the transitive closure of E),
4. for every $n \in N_q$, $lab_q(n) = \star$ or $lab_q(n) = \lambda(n)$.

If there exists an embedding of q in G, we write $G \preccurlyeq q$. Next, a *simulation* of a dependency graph $G = (\Sigma, root, E)$ in a tree t is a relation $R \subseteq \Sigma \times N_t$ such that:

1. $(root, root_t) \in R$,
2. for every $(a, n) \in R$, $(a, a') \in E$, there exists $n' \in N_t$ such that $(n, n') \in child_t$ and $(a', n') \in R$,

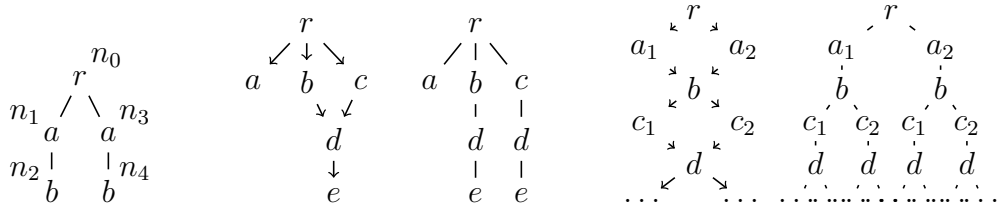
3. for every $(a, n) \in R$. $lab_t(n) = a$.

Note that R is a total relation for the nodes of the graph reachable from the root i.e., for every $a \in \Sigma$ reachable from *root* in G, there exists a node $n \in N_t$ such that $(a, n) \in R$. If there exists a simulation from G to t, we write $t \preccurlyeq G$. Additionally, note that given a graph containing cycles reachable from the root, there does not exist any (finite) tree where it can be simulated. However, we point out that in the remainder we use the notion of simulation only for universal dependency graphs that are supposed to come from trimmed IMSs, hence they do not have such cycles.

Given two dependency graphs $G_1 = (\Sigma, root, E_1)$ and $G_2 = (\Sigma, root, E_2)$, G_1 is a *subgraph* of G_2 if $E_1 \subseteq E_2$. For a dependency graph $G = (\Sigma, root, E)$, we define the partial order $\leqslant_G$ on the subgraphs of G: given G_1 and G_2 two subgraphs of G, $G_1 \leqslant_G G_2$ if G_1 is a subgraph of G_2. Note that the relation $\leqslant_G$ is reflexive, antisymmetric, and transitive, thus being an ordering relation. Moreover, it is well-founded and it has a minimal element $G_0 = (\Sigma, root, \varnothing)$. The following result can be easily shown by a structural induction using the order $\leqslant_G$.

Lemma 1.7.3 *For every IMS S, its universal dependency graph can be simulated in every tree t which belongs to the language of S.*

A *path in a dependency graph* $G = (\Sigma, root, E)$ is a non-empty sequence of vertices starting at *root* such that for every two consecutive vertices in the sequence, there is a directed edge between them in G. By $Paths(G) \subseteq \Sigma^+$ we denote the set of all paths in G. The set of paths is finite only for graphs without cycles reachable from the root. For instance, the paths of the graph G_1 in Figure 1.5(b) are $Paths(G_1) = \{r, ra, rb, rc, rbd, rcd, rbde, rcde\}$.



(a) Tree t_1.

(b) Graph G_1 and its unfolding.

(c) Graph G_2 and its unfolding.

Figure 1.5: A tree and two graphs with their corresponding unfoldings.

Similarly, a *path in a tree* t is a non-empty sequence of nodes starting at $root_t$ such that every two consecutive nodes in the sequence are in the relation

$child_t$. By $Paths(t) \subseteq N_t^+$ we denote the set of all paths in t. Then, we define $LabPaths(t) \subseteq \Sigma^+$ as the set of sequences of labels of nodes from all paths in t. For instance, for the tree t_1 from Figure 1.5(a) we have $Paths(t_1) = \{n_0, n_0n_1, n_0n_1n_2, n_0n_3, n_0n_3n_4\}$ and $LabPaths(t_1) = \{r, ra, rab\}$. Note that $|LabPaths(t)| \leqslant |Paths(t)|$. The *unfolding* of a dependency graph $G = (\Sigma, root, E)$, denoted u_G, is a tree $u_G = (N_{u_G}, root_{u_G}, lab_{u_G}, child_{u_G})$ such that:

- $N_{u_G} = Paths(G)$,
- $root_{u_G} \in N_{u_G}$ is the root of u_G,
- $(p, p \cdot a) \in child_{u_G}$, for every path $p, p \cdot a \in Paths(G)$ (note that "$\cdot$" stands for standard ordered concatenation),
- $lab_{u_G}(root_{u_G}) = root$, and $lab_{u_G}(p \cdot a) = a$, for every path $p \cdot a \in Paths(G)$.

The unfolding of a graph is finite only when the graph has no cycle reachable from the root, because otherwise $Paths(G)$ is infinite, hence u_G is infinite. In the remainder, we use the unfolding only for graphs having no cycle reachable from the root (in order to have finite unfoldings). In such a case, the unfolding can be seen as the *smallest* tree u_G (w.r.t. the number of nodes) having $LabPaths(u_G) = Paths(G)$. The idea of the unfolding is to transform the dependency graph G into a tree having the *child* relation instead of directed edges. There are nodes duplicated in order to avoid nodes with more than one incoming edge. For instance, in Figure 1.5(b) we take the graph G_1 and construct its unfolding u_{G_1}. Moreover, notice that the size of the unfolding may be exponential in the size of the graph, for example for the graph G_2 from Figure 1.5(c).

We also extend the definition of embedding and propose the embedding from a tree to another tree i.e., given two trees t and t', we say that t' can be embedded in t (denoted $t \preccurlyeq t'$) if the query $(N_{t'}, root_{t'}, lab_{t'}, child_{t'}, \varnothing)$ can be embedded in t. Similarly, we can define the embedding from a tree to a dependency graph. Note that two embeddings can be *composed*, for example:

- $\forall t, t' \in Tree.\ \forall q \in Twig.\ (t \preccurlyeq t' \wedge t' \preccurlyeq q \Rightarrow t \preccurlyeq q)$,
- $\forall S \in IMS.\ \forall t \in Tree.\ \forall q \in Twig.\ (G_S^{\forall/\exists} \preccurlyeq t \wedge t \preccurlyeq q \Rightarrow G_S^{\forall/\exists} \preccurlyeq q)$.

We state next two auxiliary lemmas that can be easily proven by structural induction on the dependency graphs (using the order $\leqslant_G$):

Lemma 1.7.4 *A dependency graph G can be simulated in a tree t iff its unfolding u_G can be embedded in t.*

Lemma 1.7.5 *A query q can be embedded in a dependency graph G iff q can be embedded in the unfolding tree of G.*

In Figure 1.6 we present the operations *fuse* and *add*. Given two trees t and t', we say that $t \lhd_0 t'$ if t' is obtained from t by applying one of the operations from Figure 1.6. The *fuse* operation takes two siblings with the same label and creates only one node having below it the subtrees corresponding to each of the siblings. The *add* operation consists simply in adding a subtree at some place in the tree. By $\trianglelefteq$ we denote the transitive and reflexive closure of $\lhd_0$.

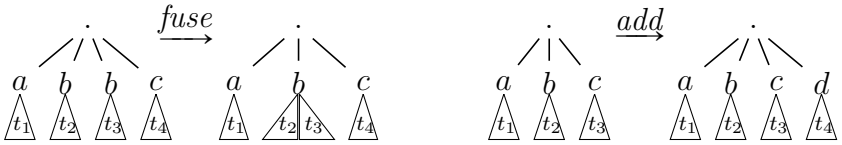


Figure 1.6: Operations *fuse* and *add*.

Note that the fuse and add operations preserve the embedding i.e., given a twig query q and two trees t and t', if $t \preccurlyeq q$ and $t \trianglelefteq t'$, then $t' \preccurlyeq q$. Furthermore, if we can embed a query q in a tree t which can be embedded in the existential dependency graph of an IMS S, we can perform a sequence of operations such that t is transformed into another tree t' satisfying S and q at the same time. Formally, we have the following.

Lemma 1.7.6 *Given an IMS S, a query q and a tree t, if $G_S^{\exists} \preccurlyeq t$ and $t \preccurlyeq q$, then there exists a tree $t' \in L(S) \cap L(q)$. The tree t' can be constructed after a sequence of fuse and add operations (consistently with the schema S) from the tree t and we denote $t \trianglelefteq_S t'$.*

1.7.3 Family of characteristic graphs

Given a schema S and a query q, we can capture all trees satisfying both S and q with the characteristic graphs that we introduce next.

More formally, a *characteristic graph* G is a tuple $(V_G, root_G, lab_G, E_G)$, where V_G is a finite set of vertices, $root_G \in V_G$ is the root of the graph, $lab_G : V_G \to \Sigma$ is a labeling function (with $lab_G(root_G) = root_S$), and $E_G \subseteq V_G \times V_G$ is the set of edges. Let us assume that $G_S^{\exists} \preccurlyeq q$ and take such an embedding $\lambda : N_q \to \Sigma$. By $\Lambda(q, S, \lambda)$ we denote the set of all *characteristic*

graphs for q and S w.r.t. λ. To construct such a graph, let us start with $G = (V_G, root_G, lab_G, E_G)$ where V_G and E_G are empty, and perform the four steps described below.

1. For every n in N_q, add a node n' to V_G such that $lab_G(n') = \lambda(n)$. Let $root_G$ be the node such that $lab_G(root_G) = root_S$.

2. For every (n_1, n_2) in $child_q$, add (n'_1, n'_2) to E_G, where n'_1 and n'_2 are the nodes corresponding to n_1 and n_2, respectively, as constructed at step 1.

3. For every (n_1, n_2) in $desc_q$, *choose an acyclic path* $a_0, \ldots, a_k$ in $G_S^\exists$ where $\lambda(n_1) = a_0$ and $\lambda(n_2) = a_k$. Notice that, since n_1 and n_2 belong to N_q, we have already added in V_G two nodes n'_1 and n'_2, respectively, corresponding to them at step 1. Then, for every a_i (with $1 \leqslant i \leqslant k-1$), we add in V_G a node n''_i such that $lab_G(n''_i) = a_i$. Also, add in E_G the edges $(n'_1, n''_1), (n''_1, n''_2), \ldots, (n''_{k-1}, n'_2)$.

4. For every n in V_G, take from $G_S^\forall$ the subgraph $(V', lab_G(n), E')$ rooted at $lab_G(n)$. Then, for every $a \neq lab_G(n)$ in V' add a node n' in V' such that $lab_G(n') = a$. Also, for every $(a_1, a_2) \in E'$, add in E_G an edge (n_1, n_2) where n_1 and n_2 are the nodes corresponding to a_1 and a_2, respectively.

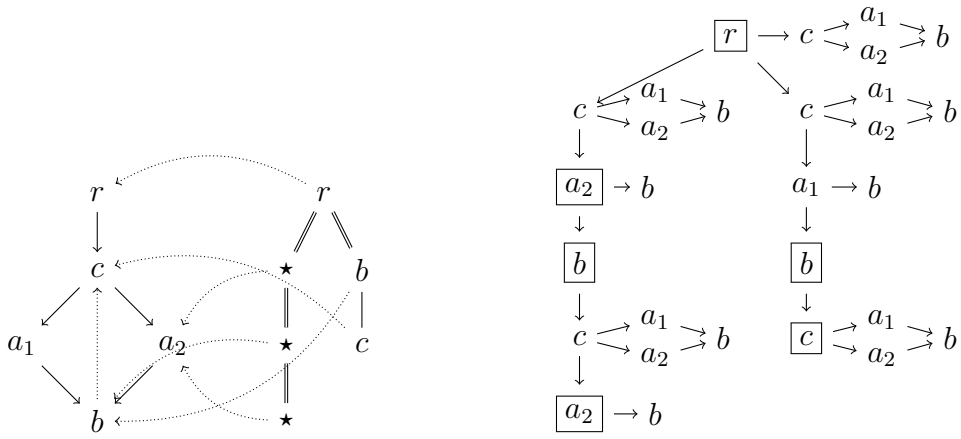
The following example illustrates the construction of such a graph.

Example 1.7.7 Take in Figure 1.7(a) an existential dependency graph $G_S^\exists$, a twig query q, and an embedding $\lambda : N_q \to G_S^\exists$. Notice that in $G_S^\exists$ we have drawn the universal edges with a full line and those that are existential without being universal with a dotted line. Then, in Figure 1.7(b) we present an example of a graph G from $\Lambda(q, S, \lambda)$. Notice that in G we have represented in boxes the nodes corresponding to the images $\lambda(n)$ for the nodes of the query $n \in N_q$. ▫

Next, we define the set of all characteristic graphs for q and S w.r.t. the all embeddings λ of q in $G_S^\exists$:

$$\mathcal{G}(q, S) = \{G \in \Lambda(q, S, \lambda) \mid \lambda \text{ is an embedding of } q \text{ in } G_S^\exists\}.$$

Note that $G \preccurlyeq q$ and the size of G is polynomially bounded by $|q| \times |\Sigma|^2$ for every G in $\mathcal{G}(q, S)$. Indeed, after step 1 of the construction, a characteristic graph G has $|q|$ nodes. Then, after steps 2 and 3, since at step 3 we allow only acyclic paths of $G_S^\exists$, we add at most $|\Sigma|$ nodes for each already existing



(a) Embedding $\lambda : N_q \to G_S^{\exists}$.

(b) Graph $G \in \Lambda(q, S, \lambda)$.

Figure 1.7: An embedding from a query q to an existential dependency graph $G_S^{\exists}$ and a graph $G \in \mathcal{G}(q, S)$. In $G_S^{\exists}$, the universal edges are drawn with a full line and those that are existential without being universal with a dotted line.

node, hence G has at most $|q| \times |\Sigma|$ nodes. Finally, after 4, since we add at most $|\Sigma|$ nodes for each already existing node, G has at most $|q| \times |\Sigma|^2$ nodes.

Furthermore, let $\Lambda^*(q, S, \lambda)$ and $\mathcal{G}^*(q, S)$ be sets of characteristic graphs constructed similarly to $\Lambda(q, S, \lambda)$ and $\mathcal{G}(q, S)$, respectively, the only difference being that we allow cyclic paths at step 3 of the aforementioned construction. While the size of the graphs in $\mathcal{G}(q, S)$ is polynomial , notice that the size of the graphs in $\mathcal{G}^*(q, S)$ is not necessary polynomial since the possible cyclic paths chosen at step 3 can be arbitrarily long. Additionally, note that $|\mathcal{G}(q, S)|$ is finite and may be exponential while $|\mathcal{G}^*(q, S)|$ may be infinite if the existential dependency graph $G_S^{\exists}$ contains cycles reachable from the root.

Next, we extend the previous definition of the unfolding to the characteristic graphs. Given an IME E and a symbol a, by $min\_nb(E, a)$ we denote the minimum number of occurrences of the symbol a in every unordered word defined by E. Next, we define the *unfolding of a characteristic graph*. Given a query q, an IMS S, and a characteristic graph $G \in \mathcal{G}^*(q, S)$, we construct its unfolding as follows:

- Let u_G be the unfolding of G obtained as defined in Section 1.7.2.
- Update u_G such that for every $n \in N_{u_G}$, for every $a \in \Sigma$, let t_a the subtree having as root the child of n labeled by a. Next, add copies of

t_a as children of n until n has $min\_nb(R_S(lab_{u_G}(n)), a)$ children labeled by a.

Notice that every graph G in $\mathcal{G}^*(q, S)$ is acyclic. Indeed, when constructing such a graph G, after steps 1, 2 and 3, G is basically shaped as a tree. Then, the subgraphs that we fuse at step 4 are all acyclic since they are subgraphs of the universal dependency graph $G_S^\forall$ that we assume trimmed (cf. Section 1.7.1). Since every graph G in $\mathcal{G}^*(q, S)$ is acyclic, it has a finite unfolding, which naturally belongs to the language of S.

1.7.4 Complexity results

In this section, we use the above defined tools to show the complexity results for IMSs. First, the dependency graphs and embeddings capture satisfiability and implication of queries by IMSs.

Lemma 1.7.8 *Given a twig query q and an IMS S:*

1. *q is satisfiable by S iff $G_S^\exists \preccurlyeq q$,*
2. *q is implied by S iff $G_S^\forall \preccurlyeq q$.*

Proof *1)* For the *if* part, we know that $G_S^\exists \preccurlyeq q$, thus the family of graphs $\mathcal{G}(q, S)$ is not empty. The unfolding of every graph from $\mathcal{G}(q, S)$ satisfies S and q at the same time, hence q is satisfiable by S. For the *only if* part, we know that there exists a tree $t \in L(S) \cap L(q)$, and we assume w.l.o.g. that it is the unfolding of a graph G from $\mathcal{G}^*(q, S)$. Since $t \preccurlyeq q$, we obtain $u_G \preccurlyeq q$, hence $G \preccurlyeq q$ (by Lemma 1.7.5), which, from the construction of G, implies that $G_S^\exists \preccurlyeq q$.

2) For the *if* part, we know that $G_S^\forall \preccurlyeq q$, which implies by Lemma 1.7.5 that $u_{G_S^\forall} \preccurlyeq q$. On the other hand, take a tree $t \in L(S)$. By Lemma 1.7.3 we have $t \preccurlyeq G_S^\forall$, which implies by Lemma 1.7.4 that $t \preccurlyeq u_{G_S^\forall}$. From the last embedding and $u_{G_S^\forall} \preccurlyeq q$ we infer that $t \preccurlyeq q$. Since t can be every tree in the language of S, we conclude that q is implied by S. For the *only if* part, we know that for every $t \in L(S)$, $t \preccurlyeq q$. Consider the tree t obtained as follows: we take $u_{G_S^\forall}$ and we duplicate some subtrees in order to have, for each node $n \in N_t$, $min\_nb(R_S(lab_t(n)), a)$ children labeled by a. Naturally, t is in the language of S, hence $t \preccurlyeq q$ from the hypothesis. From the definition of the unfolding, we infer that $G_S^\forall \preccurlyeq t$, which implies that $G_S^\forall \preccurlyeq q$. □

For instance, the twig query $q = r[a]/b//d$ can be embedded in the existential dependency graph of the IMS S from Example 1.7.2, thus q is satisfiable by

S. In Figure 1.8 we present embeddings of q in $G_S^{\exists}$ and in a tree t satisfying both S and q. Additionally, notice that the twig query $q = r[a]/b//d$ cannot be embedded in $G_S^{\forall}$ from Example 1.7.2, and therefore, q is not implied by S. On the other hand, the twig query $q' = r/b//d$ can be embedded in $G_S^{\forall}$, thus q' is implied by S.

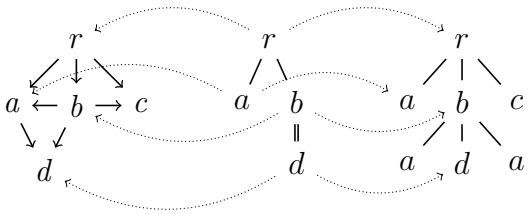


Figure 1.8: Embeddings of q in $G_S^{\exists}$ and in a tree t which satisfies S and q at the same time.

Moreover, we point out that testing the embedding of a query in a dependency graph can be done in polynomial time with a simple bottom-up algorithm. From this observation and Lemma 1.7.8 we obtain the following.

Theorem 1.7.9 $\mathrm{SAT}_{IMS,Twig}$ *and* $\mathrm{IMPL}_{IMS,Twig}$ *are in PTIME.*

Next, we present the complexity of query containment in the presence of IMSs. The coNP-completeness of the containment of twig queries [MS04] implies the coNP-hardness of the containment of twig queries in the presence of IMSs. Proving the membership of the problem to coNP is, however, not trivial. Given an instance (p, q, S), the set of all trees satisfying p and S can be characterized with a set $\mathcal{G}(p, S)$ containing an exponential number of polynomially-sized graphs and p is contained in q in the presence of S iff the query q can be embedded into all graphs in $\mathcal{G}(p, S)$. This condition is easily checked by a non-deterministic Turing machine.

Theorem 1.7.10 $\mathrm{CNT}_{IMS,Twig}$ *is coNP-complete.*

Proof The coNP-completeness of the containment of twig queries (Theorem 4 in [MS04]) implies that $\mathrm{CNT}_{IMS,Twig}$ is coNP-hard. Next, we prove the membership of the problem to coNP. Given an instance (p, q, S), a witness is a function $\lambda : N_p \to \Sigma$. Testing whether λ is an embedding from p to $G_S^{\exists}$ requires polynomial time. If λ is an embedding, a non-deterministic polynomial algorithm chooses a graph G from $\Lambda(p, S, \lambda)$ and checks whether q can be embedded in G. We claim that $p \not\subseteq_S q$ iff there exists a graph G in $\mathcal{G}(p, S)$ such that $G \not\models q$.

For the *if* case, we assume that there exists a graph $G \in \mathcal{G}(p, S)$ such that $G \not\preccurlyeq q$. We know that $G \preccurlyeq p$, thus $u_G \preccurlyeq p$ (by Lemma 1.7.5), hence there exists a tree $t \in L(S)$ such that $t \preccurlyeq p$ and $u_G \trianglelefteq_S t$ (by Lemma 1.7.6). If we assume by absurd that $t \preccurlyeq q$, we have $u_G \preccurlyeq q$, thus $G \preccurlyeq q$, which is a contradiction. We infer thus that there exists a tree $t \in L(S) \cap L(p)$, such that $t \notin L(q)$, and consequently, $p \not\subseteq_S q$.

For the *only if* case, we assume that $p \not\subseteq_S q$, hence there exists a tree $t \in L(S) \cap L(p)$ such that $t \notin L(q)$. Because $t \in L(S) \cap L(p)$, we know that there exists a graph $G \in \mathcal{G}^*(p, S)$, such that $u_G \trianglelefteq_S t$. We know that $t \not\preccurlyeq q$, thus $u_G \not\preccurlyeq q$ (by Lemma 1.7.5), that yields $G \not\preccurlyeq q$. Furthermore, by using a simple pumping argument, we have $\forall q \in \mathit{Twig}.\ \forall G \in \mathcal{G}^*(q, S).\ (G \not\preccurlyeq q \Rightarrow \exists G' \in \mathcal{G}(q, S).\ G' \not\preccurlyeq q)$, which implies that there exists a graph $G' \in \mathcal{G}(p, S)$ such that $G' \not\preccurlyeq q$. □

1.7.5 Extentions to disjunction-free DTDs

We also point out that the complexity results for implication and containment of twig queries in the presence of IMSs can be adapted to disjunction-free DTDs. This allows us to state results which, to the best of our knowledge, are novel. Similarly to the IMSs, we represent a *disjunction-free DTD* as a tuple $S = (\mathit{root}_S, R_S)$, where root_S is a designated root label and R_S maps symbols to regular expressions using no disjunction, basically regular expressions of the grammar:

$$E ::= \epsilon \mid a \mid E^* \mid E^? \mid E^+ \mid (E \cdot E),$$

where $a \in \Sigma$ and "$\cdot$" stands for the standard concatenation operator. Given such an expression E, let $\mathit{symbols}^\forall(E)$ be the set of symbols present in all words from $L(E)$, and $\mathit{symbols}^\exists(E)$ the set of symbols present in at least one word from $L(E)$:

$$\mathit{symbols}^\forall(E) = \{a \in \Sigma \mid \forall w \in L(E).\ \exists w_1, w_2.\ w = w_1 \cdot a \cdot w_2\},$$
$$\mathit{symbols}^\exists(E) = \{a \in \Sigma \mid \exists w \in L(E).\ \exists w_1, w_2.\ w = w_1 \cdot a \cdot w_2\}.$$

As pointed out for the IMEs, note that the sets $\mathit{symbols}^\forall(E)$ and $\mathit{symbols}^\exists(E)$ can be easily constructed from E. Next, we adapt the notions of dependency graph and universal dependency graph for disjunction-free DTDs. The *existential dependency graph* of a disjunction-free DTD S is a directed rooted graph $G_S^\exists = (\Sigma, \mathit{root}_S, E_S^\exists)$, where

$$E_S^\exists = \{(a, a') \mid a' \in \mathit{symbols}^\exists(R_S(a))\}.$$

Similarly, the *universal dependency graph* of a disjunction-free DTD S is a directed rooted graph $G_S^{\forall} = (\Sigma, root_S, E_S^{\forall})$, where

$$E_S^{\forall} = \{(a, a') \mid a' \in symbols^{\forall}(R_S(a))\}.$$

Analogously to the IMSs, we assume w.l.o.g. that we manipulate only disjunction-free DTDs having no cycle reachable from the root in the universal dependency graph. Otherwise, if there is a cycle in the universal dependency graph, this means that there is no tree consistent with the schema and containing at least one of the symbols implied in that cycle. Moreover, similarly to IMSs, for a symbol $a \in \Sigma$ and a disjunction-free regular expression E, by $min\_nb(E, a)$ we denote the minimum number of occurrences of the symbol a in every word defined by E.

Next, we state our complexity results for disjunction-free DTDs.

Theorem 1.7.11 $\mathrm{IMPL}_{\textit{disj-free-DTD},\textit{Twig}}$ *is in PTIME and* $\mathrm{CNT}_{\textit{disj-free-DTD},\textit{Twig}}$ *is coNP-complete.*

Proof We claim that a query q is implied by a disjunction-free DTD S iff $G_S^{\forall} \preccurlyeq q$ and since the embedding of a query in a graph can be computed in polynomial time, this implies that $\mathrm{IMPL}_{\textit{disj-free-DTD},\textit{Twig}}$ is in PTIME. The proof follows from the proof of Lemma 1.7.8.2. The coNP-completeness of the containment of twig queries (Theorem 4 in [MS04]) implies that $\mathrm{CNT}_{\textit{disj-free-DTD},\textit{Twig}}$ is coNP-hard. Theorem 1.7.10 states the coNP-completeness of the query containment in the presence of IMSs and an easy adaptation of its proof technique yields the membership of $\mathrm{CNT}_{\textit{disj-free-DTD},\textit{Twig}}$ to coNP. The mentioned proofs can be adapted because given a disjunction-free regular expression E and a word $u \in L(E)$, u can in fact be obtained as an ordering of the unordered word $w = \biguplus_{a \in \Sigma} a^{min\_nb(E,a)}$. Moreover, the order imposed by the DTD on the siblings is not important because the twig queries are order-oblivious. □

1.8 Expressiveness of interval multiplicity schemas

First, we compare the expressive power of DIMSs with yardstick languages of unordered trees. We begin with FO logic that uses only the binary *child* predicate and the unary label predicates P_a with $a \in \Sigma$. It is easy to show that DIMSs are not comparable with FO. With a simple rule $a \rightarrow (b \parallel c)^*$ a DIMS can express the language of trees where every node labeled by a has as children only nodes labeled by b and c such that the number of b's is equal to the number of c's. Such language cannot be captured with FO for

reasons similar to those for which it cannot be expressed in FO whether the cardinality of the universe is even. There are languages of unordered trees expressible by FO, but not expressible by DIMSs e.g., the language of trees that contain exactly two nodes labeled b. Such languages are not expressible by DIMSs for reasons similar to those for which they cannot be expressed by DTDs, more precisely they are not closed under substitution of subtrees with the same root type (cf. Lemma 2.10 in [PV00]). By using exactly the same examples, note that DIMSs and MSO are also incomparable. MSO with Presburger constraints [SSM03, SSM08, BT05, BTT05] is essentially an extension of MSO that additionally allows elements of arithmetic (numerical variables and value comparisons) and unary functions $\#a$ that return the number of children of a node having a given label $a \in \Sigma$. This extension is very powerful and can express Parikh images of arbitrary regular languages. DIMSs are strictly less expressive than Presburger MSO as they use a strict restriction of unordered regular expressions.

Next, we compare the expressive power of DIMSs and DTDs. For this purpose, we introduce a simple tool for comparing regular expressions with DIMEs. Given a regular expression R, the language $L(R)$ of unordered words is obtained by removing the relative order of symbols from every ordered word defined by R. A DIME E *captures* R if $L(E) = L(R)$. This tool is equivalent to considering DTDs under *commutative closure* [BM99, NS]. We believe that this simple comparison is adequate because if a DTD is to be used in a data-centric application, then supposedly the order between siblings is not important. Therefore, a DIME that captures a regular expression defines basically the same admissible content model of a node, without imposing an order among the children.

Naturally, by using the above notion to compare the expressive powers of DTDs and DIMSs, DTDs are strictly more expressive than DIMSs. For example, the commutative closure of the regular expression $(a \cdot (b \mid c))^*$ cannot be expressed by a DIME. Various classes of regular expressions have been reported in widespread use in real-world schemas and have been studied in the literature: *simple regular expressions* [BNVdB04, MNS04], *single occurrence regular expressions* (SOREs) [BNSV10], *chain regular expressions* (CHAREs) [BNSV10]. DIMEs are strictly more expressive than CHAREs and incomparable to the other mentioned classes of regular expressions.

Finally, we investigate how many real-life DTDs can be captured with DIMSs and use the comparison on the XMark benchmark [SWK+02] and the University of Amsterdam XML Web Collection [GM13]. All 77 regular expressions of the XMark benchmark are captured by DIMEs, and among them 76 by IMEs. As for the DTDs from the University of Amsterdam XML Web Collection, 92% of regular expressions are captured by DIMEs

and among them 78% by IMEs. We also point out that CHAREs, captured by DIMEs, are reported to represent up to 90% of regular expressions used in real-life DTDs [BNSV10]. These numbers give a generally positive coverage, but should be interpreted with caution, as we do not know which of the considered DTDs were indeed intended for data-centric applications.

1.9 Related work

Languages of unordered trees can be expressed by *logic formalisms* or by *tree automata*. Boneva et al. [BT05, BTT05] make a survey on such formalisms and compare their expressiveness. The fundamental difference resides in the kind of constraints that can be expressed for the allowed collections of children for some node. We mention here only formalisms introduced in the context of XML. *Presburger automata* [SSM03], *sheaves automata* [DZL03], and the *TQL logic* [CG04] allow to express *Presburger constraints* on the numbers of occurrences of the different symbols among the children of some node. Suitable restrictions allow to obtain the same expressiveness as the *Presburger MSO logic* on unordered trees [BT05, BTT05], strictly more expressive than DIMSs. Additionally, we believe that DIMSs are more appropriate to be used as schema languages, as they were designed as such, in particular regarding the more user-friendly DTD-like syntax.

Languages of unordered trees can be also expressed by considering DTDs under *commutative closure* [BM99, NS]. We assume DTDs using arbitrary regular expressions, not necessarily one-unambiguous [BKW98] as required by the W3C. We also point out that it has been recently shown that it is PSPACE-complete to decide whether a given regular expression can be rewritten as an equivalent one-unambiguous one [CDLM13]. Given a DTD using arbitrary regular expressions under commutative closure, we say that an (ordered) tree matches such a DTD iff every tree obtained by reordering of sibling nodes also matches the DTD. However, it is PSPACE-complete to test whether a DTD defines a commutatively-closed set of trees [NS] and, moreover, such a DTD may be of exponential size w.r.t. the size of the alphabet, which makes such DTDs unfeasible. Another consequence of the high expressive power of DTDs under commutative closure is that the membership problem is NP-complete [KT10]. Therefore, these formalisms were not extensively used in practice. From a different point of view, Martens et al. [MN05, MNG08] investigate DTDs equipped with formulas from the $\mathcal{SL}$ logic that specifies unordered languages and obtain complexity improvements for typechecking XML transformations.

The unordered concatenation operator "$\|$" should not be confused with

the *shuffle* (*interleaving*) operator "&" used in a restricted form in XML Schema and RELAX NG to define order-oblivious, yet still ordered, content. On the one hand, $a^*\&b$ defines all ordered words with an arbitrary number of a's and exactly one occurrence of b, and analogously, $a^* \parallel b$ defines all unordered words with exactly the same characteristic. On the other hand, $(a\&b)^*$ defines ordered words of the form $w_1 \cdot \ldots \cdot w_n$, where the factors $w_1, \ldots, w_n$ are either ab or ba, while $(a \parallel b)^*$ defines unordered words having the same number of a's and b's. For instance, $(a\&b)^*$ does not accept the ordered word $aabb$ while it has the same number of a's and b's. Adding the shuffle and interval multiplicities to the regular expressions increases the computational complexity of fundamental decision problems such as: membership [BBH11, Hov12], inclusion, equivalence, and intersection [GMN09]. Colazzo et al. [CGPS13, CGS09, GCS08] propose efficient algorithms for membership and inclusion of *conflict-free types*, a class of regular expressions with shuffle and numerical constraints using intervals. Their approach is based on capturing a language with a set of constraints, similar to our characterizing tuples for DIMEs. While conflict-free types and DIMEs both forbid repetitions of symbols, they differ on the restrictions imposed on the use of the operators and the interval multiplicities. Consequently, they are incomparable.

We finally point out that the static analysis problems involving twig queries i.e., twig query satisfiability [BFG08], implication [HKIF11, BMS13], and containment [NS06] in the presence of schema have been extensively studied in the context of DTDs. However, to the best of our knowledge, these problems have not been previously studied neither for the mentioned unordered schema languages, nor for DTDs using classes of regular expressions extended with counting and interleaving.

Chapter 2

Relational-to-graph data exchange

Data exchange is the problem of translating data structured under a source schema according to a target schema and a set of source-to-target constraints known as schema mappings. In this chapter, we investigate the problem of data exchange in a heterogeneous setting, where the source is a relational database, the target is a graph database, and the schema mappings are defined across them. We study the classical problems considered in data exchange, namely the existence of solutions and query answering. We show that both problems are intractable in the presence of target constraints, already under significant restrictions.

2.1 Context

Data exchange is the problem of translating data structured under a source schema according to a target schema and a set of source-to-target constraints [FKMP05]. Such a problem has been studied in settings where both the source and target schemas belong to the same data model, in particular relational and nested relational [PVM+02, FKMP05], XML [AL08], or graph [BPR13]. Settings in which the source and the target schema are of heterogeneous data models have not been considered so far, apart from combinations of relational and nested relational schemas in schema mapping tools [PVM+02, KPSS14].

In this chapter, we focus on the problem of exchanging data between relational sources and graph-shaped target databases, which might occur in several interoperability scenarios in the Semantic Web, such as ontology-based data access [PLC+08] and direct mappings [SAM12]. Motivations to map relational data to graphs abound, due to the far majority of data residing in relational databases and the need of integrating large amounts of linked data.

We express the relationships between the source and the target via *schema mappings* [PVM+02, FKMP05, BBR11] i.e., logical assertions between two conjunctive queries, one on the source and the other on the target. Schema

mappings between graph databases have already been introduced in [BPR13] and we adopt their syntax for expressing the consequents of relational-to-graph schema mapping assertions. We point out that the setting without target constraints directly follows from the results in [BPR13] on graph-to-graph data exchange. Furthermore, motivated by the fact that target constraints have been largely investigated within relational data exchange but so far disregarded for graph data exchange [BPR13], we add them to our setting.

In particular, we focus on two fundamental problems of interest: *existence of solutions* (i.e., given a source schema and an instance of it, a target schema, a set of source-to-target constraints, and a set of target constraints, decide whether there exists an instance of the target schema satisfying all given constraints) and *query answering* (i.e., computing the answers that hold for all solutions).

Our *main contributions* are the following:

- We show that in the presence of *target equality-generating dependencies* [BV84], both existence of solutions and query answering are intractable (NP-hard and coNP-hard, respectively). This holds even under significant restrictions.
- We relax the notion of target constraints by introducing *sameAs*[1] target constraints, inspired by RDF[2]. We show that the existence of solutions becomes tractable while query answering is intractable (coNP-hard) for the same restrictions as for the previous case.
- We show that the notion of graph patterns [BLR11], employed for graph data exchange [BPR13] as *universal representatives* of all solutions, cannot be used as such when adding target constraints.

We point out that our hardness results stand in terms of *query complexity* since in the proofs we have used a fixed source schema and instance, while the target schema and the mappings are part of the input.

We also point out that none of our results is specific to the relational-to-graph setting, and hold in any setting where the target is a graph and the source is an arbitrary data model projecting on relational tuples. The source data can then be either XML, graph-shaped, or any other complex format as long as the left-hand sides of mappings extract relational tuples from it. Indeed, we shall pinpoint that the constraints on the target graph are solely responsible for the intractability results. Nevertheless, in the remainder of

[1] http://www.w3.org/wiki/WebSchemas/sameAs
[2] http://www.w3.org/RDF/

the chapter, we focus on a relational data source for ease of presentation. It is also important to point out that this chapter has been published as [BBC15].

Organization. In Section 2.2, we define our problem setting. In Section 2.3, we illustrate particular cases that can be solved using techniques from relational and graph data exchange. In Section 2.4, we characterize the complexity of the problems of interest. In Section 2.5, we present the challenges of defining and querying universal solutions.

2.2 Problem setting

Let us assume a countably infinite set of *constants* $\mathcal{V}$ that we use both as *domain* of relational databases and as *node identifiers* (or simply *node ids*) of graph databases.

Source schemas and queries. A *source schema* $\mathcal{R}$ is a finite collection of relational symbols. Each relational symbol has an *arity* that is a positive integer. An *instance* I of $\mathcal{R}$ is a function associating to each relational symbol R from $\mathcal{R}$ a set of tuples over $\mathcal{V}$ having the same arity as R. We abuse notation and use R to denote both relational symbol and its instance. A *source query* is a conjunction of atoms over $\mathcal{R}$ that uses only variables.

Target schemas and queries. A *target schema* Σ is a finite alphabet. An *instance* over Σ is a *directed, edge-labeled graph* $G = (V, E)$, where $V \subseteq \mathcal{V}$ is a finite set of node ids and $E \subseteq V \times \Sigma \times V$ is a finite set of edges. A *nested regular expression* (*NRE*) is an expression of the following grammar:

$$r := \varepsilon \mid a \ (a \in \Sigma) \mid a^- \ (a \in \Sigma) \mid r + r \mid r \cdot r \mid r^* \mid [r],$$

where "$+$" stands for disjunction, "$\cdot$" for concatenation, "$*$" for Kleene star, "$^-$" for traversing edges backwards, and "$[\,]$" for nesting. An NRE r defines a binary relation over graph nodes: $[\![r]\!]_G$ is the set of pairs of nodes in G s.t. there exists a path defined by r between the two nodes [BPR13]. A *target query* is a *conjunction of nested regular expressions* (*CNRE*) using variables only. We illustrate CNREs in Example 2.2.2.

Schema mappings. A schema mapping is a set of *source-to-target tuple-generating dependencies* [BV84] (or simply *s-t tgds*) i.e., a set of formulas of the form

$$\forall \overline{x}.\ (\phi_{\mathcal{R}}(\overline{x}) \rightarrow \exists \overline{y}.\ \psi_{\Sigma}(\overline{x}, \overline{y})),$$

where $\phi_{\mathcal{R}}(\overline{x})$ is query over $\mathcal{R}$ and $\psi_{\Sigma}(\overline{x}, \overline{y})$ is a query over Σ. By $\overline{x}$ and $\overline{y}$ we denote vectors of variables. Moreover, all variables in $\overline{x}$ appear free in $\phi_{\mathcal{R}}(\overline{x})$, all variables in $\overline{y}$ appear free in $\psi_{\Sigma}(\overline{x}, \overline{y})$, and the variables in $\overline{x}$ that appear in $\psi_{\Sigma}(\overline{x}, \overline{y})$ are free.

Target constraints. We consider two well-known types of target constraints:

- *target equality-generating dependencies* [BV84] (or simply *egds*) i.e., $\forall \overline{x}.\ (\psi_{\Sigma}(\overline{x}) \rightarrow (x_1 = x_2))$,
- *target tuple-generating dependencies* [BV84] (or simply *target tgds*) i.e., $\forall \overline{x}.\ (\phi_{\Sigma}(\overline{x}) \rightarrow \exists \overline{y}.\ \psi_{\Sigma}(\overline{x}, \overline{y}))$.

In the aforementioned definitions, $\phi_{\Sigma}(\overline{x})$ and $\psi_{\Sigma}(\overline{x})$ are CNREs over Σ, and x_1 and x_2 are among the variables in $\overline{x}$. Moreover, we introduce *sameAs target constraints* that are a special case of target tgds i.e.,

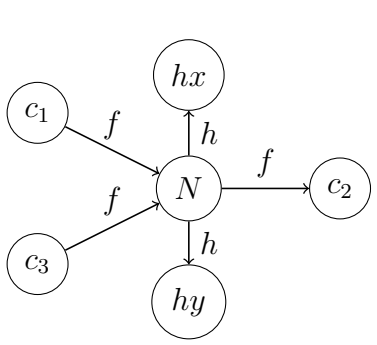
$$\forall \overline{x}.\ (\psi_{\Sigma}(\overline{x}) \rightarrow (x_1, sameAs, x_2)).$$

In the sequel, we omit w.l.o.g. the universal quantifiers in front of a formula.

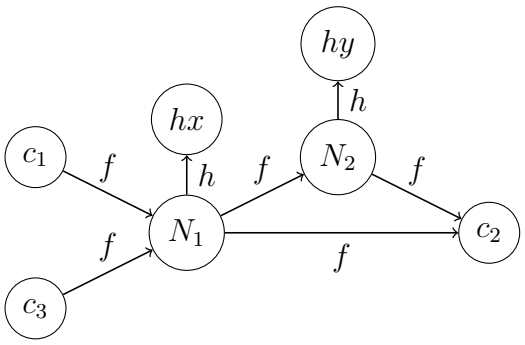
Definition 2.2.1 *A* (relational-to-graph) data exchange setting $\Omega = (\mathcal{R}, \Sigma, \mathcal{M}_{st}, \mathcal{M}_t)$ *consists of a relational source schema* $\mathcal{R}$*, a graph target schema* Σ*, a set* $\mathcal{M}_{st}$ *of s-t tgds, and a set* $\mathcal{M}_t$ *of target constraints.*

Solutions. Given a setting $\Omega = (\mathcal{R}, \Sigma, \mathcal{M}_{st}, \mathcal{M}_t)$, an instance I of $\mathcal{R}$ and a graph database G over Σ, we say that G is a *solution* for I under Ω if (I, G) satisfies $\mathcal{M}_{st}$ and G satisfies $\mathcal{M}_t$. We denote the set of all solutions by $Sol_{\Omega}(I)$. Usually, in relational data exchange, one aims at finding the universal solutions, from which there exist homomorphisms to all solutions [FKMP05]. This notion has been redefined for graph data exchange as universal representatives captured with graph patterns [BPR13] that we discuss more in detail in Section 2.3.2.

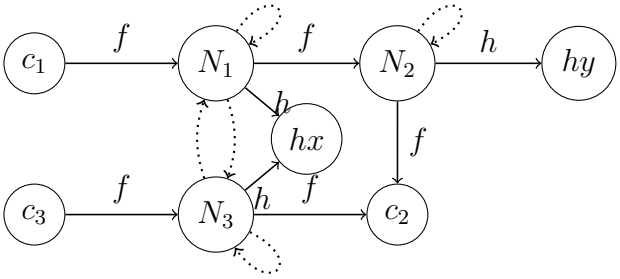
Example 2.2.2 Take a source schema $\mathcal{R}$ consisting of two relations: *Flight* storing information about flights that may have intermediate stops between the source and destination cities, and *Hotel* storing information about the hotels in which the passengers of such flights have stopped. Moreover, take the following instance I:



(a) G_1.



(b) G_2.



(c) G_3 (the dotted edges are labeled by *sameAs*).

Figure 2.1: Solutions from Example 2.2.2.

Flight		
flight_id	*src*	*dest*
01	c_1	c_2
02	c_3	c_2

Hotel	
flight_id	*hotel_id*
01	hx
01	hy
02	hx

Take a target schema consisting of the alphabet $\Sigma = \{f, h\}$. The edges labeled by f indicate a direct flight between two cities while the edges labeled by h indicate that a city has a hotel. Moreover, consider the following s-t tgd that basically requires that for each hotel where the passengers of a flight have stopped, there exists a city where the respective hotel is situated, and there exist flights from *src* to such city and from such city to *dest*:

$$\mathcal{M}_{st}: \ \mathit{Flight}(x_1, x_2, x_3) \wedge \mathit{Hotel}(x_1, x_4) \rightarrow \\ \exists y.\ (x_2, (f \cdot f^*), y) \wedge (y, h, x_4) \wedge (y, (f \cdot f^*), x_3).$$

Notice that $\mathcal{M}_{st}$ uses a CNRE on its right hand side. Then, a natural

constraint is that a hotel is situated in exactly one city, which can be captured either by the egd $\mathcal{M}_t$ or by the *sameAs* constraint $\mathcal{M}'_t$:

$$\mathcal{M}_t : \ (x_1, h, x_3) \wedge (x_2, h, x_3) \rightarrow (x_1 = x_2),$$
$$\mathcal{M}'_t : \ (x_1, h, x_3) \wedge (x_2, h, x_3) \rightarrow (x_1, sameAs, x_2).$$

The two ways of expressing the aforementioned target constraint yield two different settings $\Omega = (\mathcal{R}, \Sigma, \mathcal{M}_{st}, \mathcal{M}_t)$ and $\Omega' = (\mathcal{R}, \Sigma, \mathcal{M}_{st}, \mathcal{M}'_t)$, respectively. We illustrate in Figure 2.1 solutions for I under these two settings: the graphs G_1 and G_2 are solutions under Ω, while G_3 is a solution under Ω'. □

Problems of interest. We are interested in studying the following two problems:

1. *Existence of solutions.* Given a setting $\Omega = (\mathcal{R}, \Sigma, \mathcal{M}_{st}, \mathcal{M}_t)$ and an instance I of $\mathcal{R}$, decide whether there exists a solution for I under Ω. Additionally, we are interested in finding in our heterogeneous setting a mechanism similar to universal solutions [FKMP05] or universal representatives [BPR13].

2. *Query answering.* Given a setting $\Omega = (\mathcal{R}, \Sigma, \mathcal{M}_{st}, \mathcal{M}_t)$, an instance I of $\mathcal{R}$, and a query Q over Σ, we are interested in the *certain answers of Q w.r.t. I under Ω*, denoted $cert_\Omega(Q, I)$, which are the answers that hold for all solutions i.e., the set $\bigcap\{[\![Q]\!]_G \mid G \in Sol_\Omega(I)\}$ (where by $[\![Q]\!]_G$ we denote the set of tuples of nodes of G selected by the query Q). The query answering problem consists of deciding whether a given tuple of constants belongs to $cert_\Omega(Q, I)$ or not.

Example 2.2.2 (continued). Take the above instance I of the relations *Flight* and *Hotel*, and the above setting Ω. Then, take the query

$$Q = (x_1, f \cdot f^*[h] \cdot f^- \cdot (f^-)^*, x_2).$$

Intuitively, this query selects the pairs of nodes (x_1, x_2) from which the same hotel can be reached, or in other words, one can fly (possibly with connections) from the city x_1 to another city that has a hotel and an ingoing flight (possibly with connections) whose origin x_2 we want to select. Recall that the graphs G_1 and G_2 are both solutions for I under Ω. On these two graphs, the query Q selects as follows:

$$[\![Q]\!]_{G_1} = \{(c_1, c_1), (c_1, c_3), (c_3, c_1), (c_3, c_3)\},$$
$$[\![Q]\!]_{G_2} = \{(c_1, c_1), (c_1, c_3), (c_3, c_1), (c_3, c_3),$$
$$(c_1, N_1), (c_3, N_1), (N_1, c_1), (N_1, c_3), (N_1, N_1)\}.$$

Notice that only four pairs of nodes are common to these answer sets for the two considered graphs. Also notice that these four pairs of nodes are in fact the certain answers of Q w.r.t. I under Ω:

$$cert_{\Omega}(Q, I) = \{(c_1, c_1), (c_1, c_3), (c_3, c_1), (c_3, c_3)\}.$$

On the other hand, notice that if we want to pose the same query Q under the other aforementioned example of setting (i.e., Ω'), we obtain a different set of certain answers: $cert_{\Omega'}(Q, I) = \{(c_1, c_1), (c_3, c_3)\}$. Intuitively, this happens because the egds from the setting Ω ensure that in all of its possible solutions the nodes having the same hotel have been merged. In the second setting, this natural requirement has been encoded using a *sameAs* constraint, which is not exploited by the query Q, hence some of the certain answers of Q under Ω are no longer certain under Ω'. □

2.3 Background

In this section, we show that in two particular cases of our problem setting existing techniques from relational and graph data exchange can be applied (Section 2.3.1 and Section 2.3.2, respectively). This does not happen in the general case, as we show in the next section.

2.3.1 Relational data exchange

If we consider s-t tgds having on the right hand side conjunctions of NREs of the form a (with $a \in \Sigma$), our problem setting reduces to a particular case of relational data exchange and the techniques from relational data exchange can be naturally applied. In particular, we can see the target schema as a set of binary relational symbols (one for each symbol of the target alphabet) and the chase [FKMP05] returns a universal solution that can be essentially seen as a graph since it consists of a set of binary relations.

Example 2.3.1 Take the schemas $\mathcal{R}$ and Σ, the instance I, and the egds $\mathcal{M}_t$ from Example 2.2.2. Since we consider only NREs of the form a (with $a \in \Sigma$), we cannot express the same $\mathcal{M}_{st}$ as in Example 2.2.2. However, we can express the following:

$$\mathcal{M}'_{st}: \ \mathit{Flight}(x_1, x_2, x_3) \wedge \mathit{Hotel}(x_1, x_4) \rightarrow \exists y.\ (x_2, f, y) \wedge (y, h, x_4) \wedge (y, f, x_3).$$

We illustrate in Figure 2.2 the chased solution for I under $(\mathcal{R}, \Sigma, \mathcal{M}'_{st}, \mathcal{M}_t)$. Notice that there is no solution that has N_1 and N_2 on the same path from c_1

to c_2. Such a condition is desirable for a flight from c_1 to c_2 whose passengers have stopped in both hotels hy and hx, situated in the cities N_1 and N_2, respectively. We finally point out that we cannot capture solutions satisfying this kind of constraints for flights with an arbitrary number of stops without using the Kleene star (as in Example 2.2.2). ▫

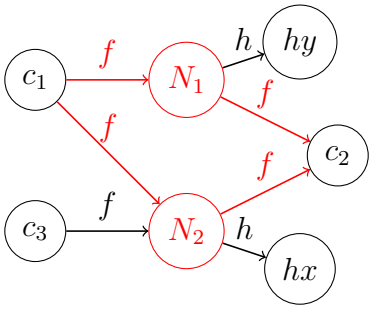


Figure 2.2: Solution from Example 2.3.1.

2.3.2 Graph data exchange

If we consider s-t tgds only, the existence of solutions and query answering can be solved using techniques from graph-to-graph data exchange [BPR13]. In particular, solutions always exist and all solutions are captured by universal representatives defined as graph patterns.

Graph patterns. Let $\mathcal{N}$ be a countably infinite set of labeled null values. A *graph pattern* π over a finite alphabet Σ is a pair (N, D), where $N \subseteq \mathcal{V} \cup \mathcal{N}$ is a finite set of node ids or null values, and $D \subseteq N \times NRE(\Sigma) \times N$, where $NRE(\Sigma)$ denotes the set of all NREs over Σ. The semantics of graph patterns are defined in terms of homomorphisms [BLR11]. Given a graph pattern $\pi = (N, D)$ and a graph database $G = (V, E)$, a *homomorphism* from π into G is a total function $h : N \to V$ s.t.:

1. h is the identity over $N \cap \mathcal{V}$ (i.e., over the node ids from N),

2. for all edges $(u, r, v) \in D$ $(u, v \in N, r \in NRE(\Sigma))$, it holds that $(h(u), h(v)) \in [\![r]\!]_G$.

We write $\pi \to G$ if there exists a homomorphism from π to G. The set of all graphs represented by π over Σ, denoted $Rep_\Sigma(\pi)$ is the set of all graphs G over Σ such that $\pi \to G$.

Universal representatives. Given a setting $\Omega = (\mathcal{R}, \Sigma, \mathcal{M}_{st}, \varnothing)$ and an instance I of $\mathcal{R}$, a graph pattern π is a *universal representative* of I under Ω if $Sol_\Omega(I) = Rep_\Sigma(\pi)$ [BPR13]. In graph data exchange, universal representatives are computed using an adaptation of the standard *chase* procedure from relational data exchange [FKMP05]. Moreover, the variant of chase that is tailored for graph data exchange [BPR13] can be easily adapted to construct a universal representative in our relational-to-graph heterogeneous setting. We illustrate a result of this procedure in Example 2.3.2. Then, query answering reduces to querying the graph pattern [BLR11] chased as universal representative.

Example 2.3.2 Take the schemas $\mathcal{R}$ and Σ, the instance I, and the s-t tgds $\mathcal{M}_{st}$ from Example 2.2.2. The graph pattern π in Figure 2.3 is a universal representative of all solutions for I under $(\mathcal{R}, \Sigma, \mathcal{M}_{st}, \varnothing)$ i.e., all graphs to which there exists a homomorphism from π are solutions. □

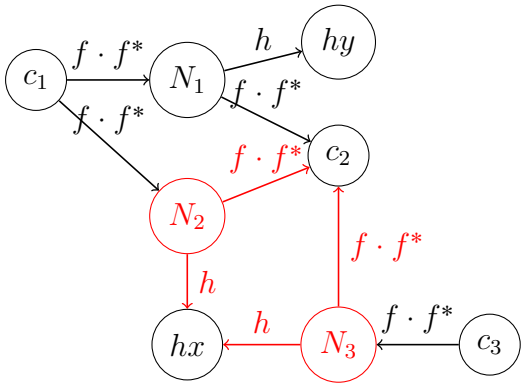


Figure 2.3: Graph pattern from Example 2.3.2.

However, notice that the sole s-t tgds might not capture interesting mapping scenarios involving graphs. As an example, the target constraint "a hotel is situated in exactly one city" cannot be expressed in settings such as the one presented in Example 2.3.2.

2.4 Complexity results

In this section, we present the main contributions of this chapter. More precisely, we study the complexity of the two problems of interest, namely existence of solutions and query answering, for settings that exhibit target egds (Section 2.4.1) or target tgds (Section 2.4.2), respectively.

2.4.1 Complexity of target egds

First, let us show the intractability of the existence of solutions when we allow egds to our setting.

Theorem 2.4.1 *Given a setting $\Omega = (\mathcal{R}, \Sigma, \mathcal{M}_{st}, \mathcal{M}_t)$ where $\mathcal{M}_t$ consists of egds, and an instance I of $\mathcal{R}$, deciding whether there exists a solution for I under Ω is NP-hard.*

Proof We prove by reduction from 3SAT, known as an NP-complete problem. The reduction works as follows. Given a formula $\rho = C_1 \wedge \ldots \wedge C_k$ in 3CNF over the set of variables $\{x_1, \ldots, x_n\}$, we construct

– The setting $\Omega_\rho = (\mathcal{R}_\rho, \Sigma_\rho, \mathcal{M}_{\rho_{st}}, \mathcal{M}_{\rho_t})$ s.t.

- $\mathcal{R}_\rho = \{R_1, R_2\}$, both unary relations,
- $\Sigma_\rho = \{a, t_1, f_1, \ldots, t_n, f_n\}$.
- $\mathcal{M}_{\rho_{st}}$ contains a unique s-t tgd
  $R_1(x) \wedge R_2(y) \rightarrow (x, a, y) \wedge (x, t_1 + f_1, x) \wedge \ldots \wedge (x, t_n + f_n, x)$.
- $\mathcal{M}_{\rho_t}$ contains two types of egds:

  (*) $(x, (t_j \cdot f_j \cdot a), y) \rightarrow (x = y)$, for $1 \leqslant j \leqslant n$,

  (**) $(x, (b_{i_1} \cdot b_{i_2} \cdot b_{i_3} \cdot a), y) \rightarrow (x = y)$, for $1 \leqslant i \leqslant k$, for $1 \leqslant i_1, i_2, i_3 \leqslant n$, $x_{i_1}, x_{i_2}, x_{i_3}$ are the variables used in clause C_i, and for $1 \leqslant l \leqslant 3$, b_{i_l} is t_{i_l} if x_{i_l} appears in a negative literal in C_i, and f_{i_l}, otherwise.

– The instance $I_\rho = \{R_1(c_1), R_2(c_2)\}$.

Intuitively, the egds are defined such that a graph collapses if each variable has more than one valuation (*) or the valuation of the variables makes the formula false (**).

We illustrate the construction on the formula $\rho_0 = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$. We have the s-t tgd $R_1(x) \wedge R_2(y) \rightarrow (x, a, y) \wedge (x, (t_1 + f_1), x) \wedge \ldots \wedge (x, (t_4 + f_4), x)$, the egds of type (*) $(x, (t_i \cdot f_i \cdot a), y) \rightarrow (x = y)$ (with $1 \leqslant i \leqslant 4$), and the egds of type (**) $(x, (f_1 \cdot t_2 \cdot f_3 \cdot a), y) \rightarrow (x = y)$ and $(x, (t_1 \cdot f_3 \cdot t_4 \cdot a), y) \rightarrow (x = y)$. Then, the graph in Figure 2.4 is a solution that encodes the valuation v s.t. $v(x_1) = v(x_2)$=true and $v(x_3) = v(x_4)$=false that makes ρ_0 true.

We claim that there exists a solution for I_ρ under Ω_ρ iff $\rho \in$3SAT.

For the *if* part, we show that the existence of a valuation making ρ true implies the existence of a solution. Take a valuation $v : \{x_1, \ldots, x_n\} \rightarrow \{\text{true, false}\}$ making ρ true. Then, construct the graph $G = (\{c_1, c_2\}, E)$ s.t.

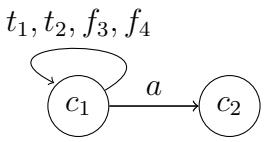


Figure 2.4: Solution for ρ_0.

$E = \{(c_1, a, c_2)\} \cup \{(c_1, t_i, c_1) \mid 1 \leqslant i \leqslant n \text{ and } v(x_i) = \text{true}\} \cup \{(c_1, f_i, c_1) \mid 1 \leqslant i \leqslant n \text{ and } v(x_i) = \text{false}\}$. Note that G and I_ρ satisfy the s-t tgd. Since there is exactly one edge labeled $b_i \in \{t_i, f_i\}$ from c_1 to c_2, the egds of type (*) are satisfied. Moreover, since the b_i's correspond to a valuation making ρ true, there is at least one satisfied literal in every clause of ρ, hence the egds of type (**) are also satisfied. Thus, G is a solution.

For the *only if* part, take a solution G. Since G satisfies the s-t tgd, we infer that G encodes at least one valuation of every variable. Since G satisfies the egds of type (*), we infer that G encodes at most one valuation of every variable. Thus, G encodes exactly one valuation of every variable. Since G satisfies the egds of type (**), we conclude that G encodes a valuation making ρ true. □

We point out that Theorem 2.4.1 holds even under significantly restricted assumptions that have been used in the proof: (i) *fixed source schema* consisting of two unary relations only, (ii) *fixed source instance*, (iii) s-t tgds using only conjunctions of NREs of the form a or $a + b$ (with $a, b \in \Sigma$) that is a slight relaxation of the restriction from Section 2.3.1, and (iv) egds that use only NREs of the form $a_1 \cdot \ldots \cdot a_n$, with pairwise distinct $a_1, \ldots, a_n \in \Sigma$ (NREs referred to as "SORE(·)" [ANS13]). Next, we prove that query answering is intractable under the same assumptions and for queries consisting of NREs that use disjunction and concatenation only.

Corollary 2.4.2 *Given a setting $\Omega = (\mathcal{R}, \Sigma, \mathcal{M}_{st}, \mathcal{M}_t)$ where $\mathcal{M}_t$ consists of egds, an instance I of $\mathcal{R}$, a NRE r, and a tuple of constants (c_1, c_2), deciding whether $(c_1, c_2) \in cert_\Omega(r, I)$ is coNP-hard.*

Proof We take the proof of Theorem 2.4.1, and we additionally consider the NRE $r_\rho = a \cdot a$. We claim that $(c_1, c_2) \in cert_{\Omega_\rho}(r_\rho, I_\rho)$ iff $\rho \notin$3SAT. For the *if* part, notice that $\rho \notin$3SAT implies that there is no solution hence (c_1, c_2) is a certain answer. For the *only if* part, since (c_1, c_2) is a certain answer, we infer that either (i) there is no solution or (ii) there is at least a solution and (c_1, c_2) is an answer for all solutions. But (ii) is false since there exist solutions for which (c_1, c_2) is not an answer for r_ρ (e.g., in Figure 2.4). Both parts follow directly from the proof of Theorem 2.4.1. □

Finally, we point out that in our reduction the source schema and instance are fixed while the target schema and the mappings are part of the input. Hence, our hardness results stand in terms of *query complexity*. Similar intractability results in the presence of target constraints (particularly in combined complexity) have been shown for relational and XML data exchange [KPT06, AL08, ADLM14, CGK13]. However, our contribution is novel since to the best of our knowledge target constraints on a graph target schema have not been previously considered in the literature, and moreover, we use a fixed source schema and instance in the proof. We also point out that our results are not specific to the relational-to-graph setting and hold in any setting where the target is a graph.

2.4.2 Complexity of target tgds

In this section, we use *sameAs* constraints instead of egds. First, let us show that the existence of solutions becomes trivial. More precisely, a solution can be computed as follows: (i) chase a graph pattern π using the s-t tgds only, (ii) take a graph G s.t. $\pi \rightarrow G$, and (iii) add in G the necessary *sameAs* edges to satisfy the *sameAs* constraints. Recall that the difficulty of deciding the existence of solutions in the case of egds was that we cannot merge two constants. Notice that this difficulty is overcome since we can add *sameAs* edges between any two nodes, even between two constants.

Next, we prove that in the presence of *sameAs* constraints the problem of certain answers is intractable under the same assumptions as in Section 2.4.1.

Proposition 2.4.3 *Given a setting $\Omega = (\mathcal{R}, \Sigma, \mathcal{M}_{st}, \mathcal{M}_t)$ where $\mathcal{M}_t$ consists of sameAs constraints, an instance I of $\mathcal{R}$, a NRE r, and a tuple of constants (c_1, c_2), deciding whether $(c_1, c_2) \in cert_\Omega(r, I)$ is coNP-hard.*

Proof We take from the proof of Theorem 2.4.1 the same $\mathcal{R}_\rho$, I_ρ, Σ_ρ, $\mathcal{M}_{\rho_{st}}$, and we replace each $(x = y)$ from $\mathcal{M}_{\rho_t}$ by $(x, sameAs, y)$ to obtain the set of *sameAs* constraints $\mathcal{M}'_{\rho_t}$ and $\Omega'_\rho = (\mathcal{R}_\rho, \Sigma_\rho \cup \{sameAs\}, \mathcal{M}_{\rho_{st}}, \mathcal{M}'_{\rho_t})$. Then, take $r'_\rho = sameAs$. We claim that $(c_1, c_2) \in cert_{\Omega'_\rho}(r'_\rho, I_\rho)$ iff $\rho \notin$3SAT, which follows similarly to Theorem 2.4.1. □

Moreover, we observe that *sameAs* constraints are a particular case of target tgds, and therefore, query answering is intractable in the presence of target tgds.

Corollary 2.4.4 *Given a setting $\Omega = (\mathcal{R}, \Sigma, \mathcal{M}_{st}, \mathcal{M}_t)$ where $\mathcal{M}_t$ consists of target tgds, an instance I of $\mathcal{R}$, a NRE r, and a tuple of constants (c_1, c_2), deciding whether $(c_1, c_2) \in cert_\Omega(r, I)$ is coNP-hard.*

2.5 Towards universal solutions

Next, we study a natural adaptation of the standard *chase* procedure [FKMP05] to take into account egds. The result of our adapted chase is a graph pattern π. To this purpose, we consider two types of chase steps: (1) for s-t tgds we do similarly to [BPR13] when computing universal representatives in graph data exchange without target constraints, and (2) for egds, for each $\psi_\Sigma(\overline{x}) \rightarrow (x_1 = x_2)$, (i) if the images in π of both x_1 and x_2 are constants, then the chase fails, (ii) if one has as image in π a constant and the other a labeled null, then the chase replaces in π the labeled null by the constant, and (iii) if both have labeled nulls as images in π, the chase chooses one of them and replaces it in π with the other.

Example 2.5.1 For the setting $(\mathcal{R}, \Sigma, \mathcal{M}_{st}, \mathcal{M}_t)$ and the instance I from Example 2.2.2, by applying the aforementioned adapted chase we obtain the graph pattern in Figure 2.5. □

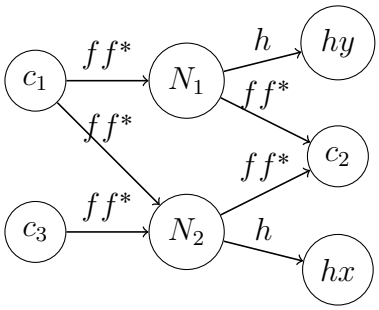


Figure 2.5: Graph pattern from Example 2.5.1.

As for relational data exchange, if the chase fails, then there is no solution. As opposed to relational data exchange, we observe that a successful chase does not guarantee the existence of a solution. Intuitively, the difficulty comes from the fact that the chase result is a graph pattern with NREs on the edges (unlike a graph with symbols on the edges). Consequently, there might not exist any graph G s.t. $\pi \rightarrow G$ and G satisfies the target constraints because it may be the case that there is no path satisfying the NREs and the egds at the same time. The following example shows such a situation.

Example 2.5.2 Take the source schema $\{R, P\}$, an instance $R(c_1)$ and $P(c_2)$, the target schema $\{a, b, c\}$, the s-t tgd $R(x) \wedge P(y) \rightarrow (x, a \cdot (b^* + c^*) \cdot a, y)$, and the egd $(x, a + b + c, y) \rightarrow (x = y)$. The aforementioned adapted chase succeeds and returns the graph pattern π in Figure 2.6(a). Although the chase

has not failed, no solution exists because there is no graph G s.t. $\pi \to G$ and G satisfies the egds. In particular, the graph G (s.t. $\pi \to G$) from Figure 2.6(b) satisfies the s-t tgd but if we try to transform it in a solution we fail because we attempt to merge two constants. □

(a) Graph pattern π. (b) Graph G.

Figure 2.6: Result of a successful chase.

We next show that, even when solutions exist, graph patterns as such cannot be used as universal representatives in the presence of egds.

Proposition 2.5.3 *Given a setting $\Omega = (\mathcal{R}, \Sigma, \mathcal{M}_{st}, \mathcal{M}_t)$ where $\mathcal{M}_t$ consists of a non-empty set of egds, and an instance I of $\mathcal{R}$, there does not exist a graph pattern π s.t. $Sol_\Omega(I) = Rep_\Sigma(\pi)$.*

Assume that there exists a graph pattern π s.t. $Sol_\Omega(I) = Rep_\Sigma(\pi)$. Then, if we take a graph $G \in Sol_\Omega$ and a homomorphism $h : \pi \to G$, we can construct the graph G' by adding nodes and edges to G s.t. some egd is no longer satisfied, thus G' is not a solution for I under Ω, but $h : \pi \to G'$ is still a homomorphism. The next example clarifies when such a situation can occur.

Example 2.5.4 The graph in Figure 2.7 is not a solution for the mappings and instance from Example 2.2.2 although there exists a homomorphism from the chased graph pattern from Figure 2.5. □

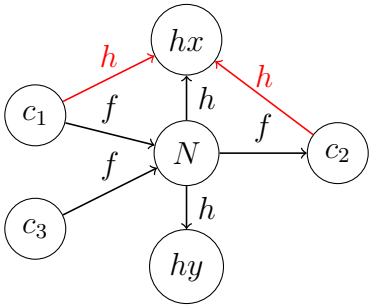


Figure 2.7: Graph from Example 2.5.4.

To address the problem of universal representatives in settings involving egds, a natural approach is to define the universal representative as a pair (graph pattern, set of egds). In this case, the solutions are the graphs satisfying the egds and s.t. there exists a homomorphism from the pattern. For example, the universal representatives for Example 2.2.2 would be the pattern in Figure 2.5 together with the egd in $\mathcal{M}_t$ from Example 2.2.2. We also point out that the above discussion can be easily generalized for *sameAs* constraints or arbitrary target tgds.

Part II

Learning schemas and queries

Chapter 3

Learning schemas for unordered XML

Similarly to Chapter 1, we consider unordered XML, where the relative order among siblings is ignored. In this chapter, we investigate the problem of learning schemas from examples given by the user and we focus on the unordered schema formalisms introduced in Chapter 1. A learning algorithm takes as input a set of XML documents that must satisfy the schema (i.e., *positive examples*) and a set of XML documents that must not satisfy the schema (i.e., *negative examples*), and returns a schema consistent with the examples. We aim for a learning algorithm that should be *sound* i.e., always return a schema consistent with the examples given by the user, and *complete* i.e., able to produce every schema with a sufficiently rich set of examples. Additionally, the algorithm should be *efficient* i.e., polynomial in the size of the input. We prove that the schemas defined by DIMEs are not learnable in the general case, where both positive and negative examples are allowed. Consequently, we identify two practical restrictions that are learnable: one from positive examples only, and another one from both positive and negative examples. Furthermore, for both learnable cases, the proposed learning algorithms return minimal schemas consistent with the examples.

3.1 Context

Similarly Chapter 1, we consider *data-centric* XML applications and unordered XML, where the relative order among siblings is ignored. As an example, take in Figure 3.1 three XML documents storing information about books. While the order of the elements title, year, author, and editor may differ from one book to another, it has no impact on the semantics of the data stored in this semi-structured database.

In this chapter, we investigate the problem of learning schemas from examples given by the user and we focus on the unordered schema formalisms introduced in Chapter 1. For instance, consider the three XML documents from Figure 3.1 and assume that the user wants to obtain a schema that is satisfied by all the three documents. A desirable solution is a schema which

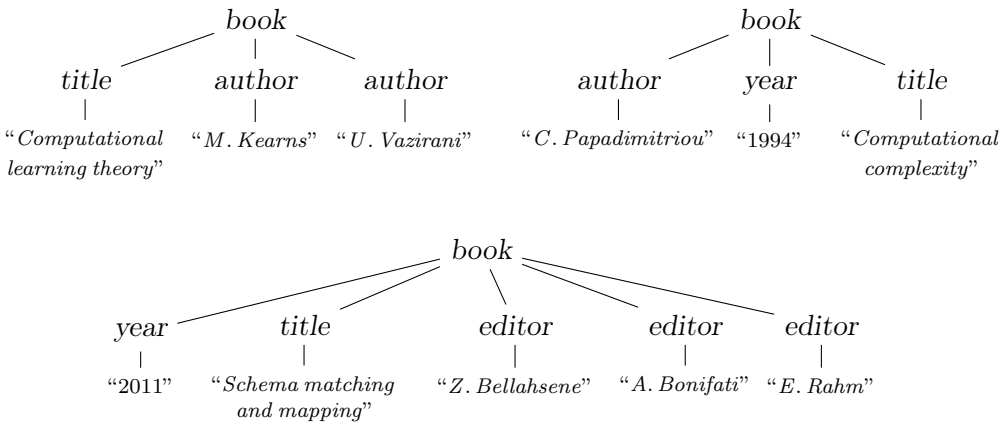


Figure 3.1: Three XML documents storing information about books.

allows a book to have, in any order, exactly one title, optionally one year, and either at least one author or at least one editor. Using the formalisms from Chapter 1, this unordered content can be expressed using the following rule that associates a DIME to the label *book*:

$$book \rightarrow title \parallel year^{?} \parallel (author^{+} \mid editor^{+}).$$

This schema allows a book to have, in any order, exactly one title, optionally one year, and either at least one author or at least one editor. Moreover, this is a *minimal* schema satisfied by the documents from Figure 3.1 because it captures the most specific schema satisfied by them (assuming that we disregard the arbitrary interval multiplicities and we use simple multiplicities only i.e., +, *, 1, ?). On the other hand, the following schema is also satisfied by the documents from Figure 3.1, but it is more general:

$$book \rightarrow title \parallel year^{?} \parallel author^{*} \parallel editor^{*}.$$

This schema allows a book to have, in any order, exactly one title, optionally one year, and any number of authors and editors. It is not minimal because it accepts a book having at the same time authors and editors, unlike the first example of schema. Moreover, the second schema associates to the label *book* an IME because it does not use the disjunction operator.

Studying the theoretical foundations of learning unordered schemas has several practical motivations. A schema serves as a reference for users who do not know yet the structure of the XML document, and attempt to query or modify its contents. If the schema is not given explicitly, it can be learned from document examples and then read by the users. From another point

of view, Florescu [Flo05] pointed out the need to automatically infer good-quality schemas and to apply them in the process of *data integration*. This is clearly a data-centric application, therefore unordered schemas might be more appropriate. Another motivation of learning the unordered schema of an XML collection is *query minimization* [AYCLS02] i.e., given a query and a schema, find a smaller yet equivalent query in the presence of the schema. Furthermore, we want to use inferred unordered schemas and optimization techniques to boost the learning algorithms for twig queries [SW12], which are order-oblivious.

Previously, schema learning has been studied from *positive examples* only i.e., documents which must satisfy the schema. For instance, we have already shown a schema learned from the three documents from Figure 3.1 given as positive examples. However, it is conceivable to find applications where *negative examples* (i.e., documents that must not satisfy the schema) might be useful. For instance, assume a scenario where the schema of a data-centric XML collection evolves over time and some documents may become obsolete w.r.t. the new schema. A user can employ these documents as negative examples to extract the new schema of the collection. Thus, the *schema maintenance* [Flo05] can be done incrementally, with little feedback needed from the user. This kind of application motivates us to investigate the problem of learning unordered schemas when we also allow negative examples.

In this chapter, we address the problem of learning unordered schemas for XML from examples given by the user. We propose a definition of the learnability influenced by computational learning theory [KV94], in particular by the inference of languages [Gol78, dlH97]. A learning algorithm takes as input a set of XML documents that must satisfy the schema (i.e., *positive examples*), and a set of XML documents that must not satisfy the schema (i.e., *negative examples*). Essentially, a class of schemas is *learnable* if there exists an algorithm which takes as input a set of examples given by the user and returns a schema which is consistent with the examples. Moreover, the learning algorithm should be *sound* i.e., always return a schema consistent with the examples given by the user, *complete* i.e., able to produce every schema with a sufficiently rich set of examples, and *efficient* i.e., polynomial in the size of the input. Our approach is novel in two directions:

- Previous research on schema learning has been done in the context of ordered XML, typically on learning automata representing restricted classes of regular expressions as content models of the DTDs. Being based on automata techniques, such algorithms take ordered input, therefore an additional input that we do not have i.e., the order among the labels. For this reason, we cannot reduce learning unordered schema

formalisms to existing learning algorithms and we have investigated new techniques to solve the problem of learning unordered schemas.

- The learning frameworks investigated before in the literature typically infer a schema using a collection of documents serving as positive examples, whereas we also study the impact of negative examples in the process of schema learning. In this case, the learning algorithm should return a schema satisfied by all the positive examples and by none of the negative ones.

Our contributions are essentially threefold. First, we prove that in the general case when we allow positive and negative examples, the DIMEs (and the schemas that they define) are not learnable. Consequently, we identify two practical restrictions that are learnable: one from positive examples only, and another one from both positive and negative examples. Furthermore, for both learnable cases, the proposed learning algorithms return minimal schemas consistent with the examples.

It is important to point out that this chapter has been published as [CS13]. Moreover, the two learnable restrictions correspond precisely to schema formalisms for unordered XML that we have previously introduced in [BCS13].

Organization. In Section 3.2 we formally define the learning framework and in Section 3.3 we prove that it is intractable in general. In Section 3.4 we present a practical class that is learnable from positive examples only, while in Section 3.5 we identify a more restricted class that is learnable from both positive and negative examples. We discuss related work in Section 3.6.

3.2 Learning framework

In this section, we define a variant of the standard language inference framework [Gol78, dlH97] adapted to learning unordered schemas for XML. A similar definition has been recently employed in the context of learning XML twig queries [SW12].

We consider the same definitions for alphabet, trees, unordered words, and for DIMEs and the schemas they define as in Chapter 1. Additionally, by W_Σ we denote the set of all unordered words over the alphabet Σ. A *learning setting* is a tuple containing the set of *concepts* that are to be learned, the set of *instances* of the concepts that are to serve as examples in learning, and the *semantics* mapping every concept to its set of instances.

Definition 3.2.1 *A* learning setting *is a tuple* $(\mathcal{E}, \mathcal{C}, \mathcal{L})$*, where* $\mathcal{E}$ *is a set of examples,* $\mathcal{C}$ *is a class of concepts, and* $\mathcal{L}$ *is a function that maps every concept in* $\mathcal{C}$ *to the set of all its examples (a subset of* $\mathcal{E}$*).*

For example, the setting for learning DIMEs from positive examples is the tuple $(W_\Sigma, \mathit{DIME}, L)$ and the setting for learning DIMSs from positive examples is $(\mathit{Tree}, \mathit{DIMS}, L)$.

The general formulation of the definition allows us to easily define settings for learning from both positive and negative examples. More precisely, we use two symbols + and − to mark whether an example is positive or negative, respectively, and we define:

- $W_\Sigma^\pm = W_\Sigma \times \{+, -\}$,
- $L^\pm(E) = \{(w, +) \mid w \in L(E)\} \cup \{(w, -) \mid w \in W_\Sigma \setminus L(E)\}$, where E is a DIME,
- $\mathit{Tree}^\pm = \mathit{Tree} \times \{+, -\}$,
- $L^\pm(S) = \{(t, +) \mid t \in L(S)\} \cup \{(t, -) \mid t \in \mathit{Tree} \setminus L(S)\}$, where S is a DIMS.

Formally, the setting for learning DIMEs from positive and negative examples is $(W_\Sigma^\pm, \mathit{DIME}, L^\pm)$, while for learning DIMS from positive and negative examples we have $(\mathit{Tree}^\pm, \mathit{DIMS}, L^\pm)$. The settings for learning restrictions of DIMEs and DIMSs are obtained analogously.

To define a learnable concept, we fix a learning setting $\mathcal{K} = (\mathcal{E}, \mathcal{C}, \mathcal{L})$ and we introduce some auxiliary notions. A *sample* is a finite nonempty subset D of $\mathcal{E}$ i.e., a set of examples. A sample D is *consistent* with a concept $c \in \mathcal{C}$ if $D \subseteq \mathcal{L}(c)$. A *learning algorithm* is an algorithm that takes a sample and returns a concept in $\mathcal{C}$ or a special value *null*.

Definition 3.2.2 *A class of concepts* $\mathcal{C}$ *is* learnable in polynomial time and data *in the setting* $\mathcal{K} = (\mathcal{E}, \mathcal{C}, \mathcal{L})$ *if there exists a polynomial learning algorithm learner satisfying the following two conditions:*

1. **Soundness.** *For every sample* D*, the algorithm* $\mathit{learner}(D)$ *returns a concept consistent with* D *or a special* null *value if no such concept exists.*
2. **Completeness.** *For every concept* $c \in \mathcal{C}$ *there exists a sample* CS_c *such that for every sample* D *that extends* CS_c *consistently with* c *i.e.,* $CS_c \subseteq D \subseteq \mathcal{L}(c)$*, the algorithm* $\mathit{learner}(D)$ *returns a concept equivalent to* c*. Furthermore, the cardinality of* CS_c *is polynomially bounded by the size of the concept.*

The sample CS_c is called the *characteristic sample* for c w.r.t. *learner* and $\mathcal{K}$. For a learning algorithm there may exist many such samples. The definition requires that one characteristic sample exists. The soundness condition is a natural requirement, but alone it is not sufficient to eliminate trivial learning algorithms. For instance, if we want to learn DIMEs from positive examples over the alphabet $\{a_1, \ldots, a_n\}$, an algorithm always returning $a_1^* \parallel \ldots \parallel a_n^*$ is sound. Consequently, we require the algorithm to be complete analogously to how it is done for grammatical language inference [Gol78, dlH97].

Typically, in the case of polynomial grammatical inference, the *size* of the characteristic sample is required to be polynomial in the size of the concept to be learned [dlH97], where the size of a sample is the sum of the sizes of the examples that it contains. From the definition of the DIMS, since repetitions of symbols are discarded among the disjunctions, the size of a schema is polynomial in the size of the alphabet. Thus, a natural requirement would be that the size of the characteristic sample is polynomially bounded by the size of the alphabet. However, there exist DIMSs such that the smallest tree in their language is exponential in the size of the alphabet, as we observe in the following example.

Example 3.2.3 We consider for $n > 1$ the alphabet $\Sigma = \{r, a_1, b_1, \ldots, a_n, b_n\}$ and the DIMS S having the root label r and the following rules:

$$
\begin{aligned}
& r \to a_1 \parallel b_1, \\
& a_i \to a_{i+1} \parallel b_{i+1} \quad (\text{for } 1 \leqslant i < n), \\
& b_i \to a_{i+1} \parallel b_{i+1} \quad (\text{for } 1 \leqslant i < n), \\
& a_n \to \epsilon, \\
& b_n \to \epsilon.
\end{aligned}
$$

We present in Figure 3.2 the unique tree satisfying this schema and we observe that its size is exponential in the size of the alphabet. ▫

Consequently, we have imposed in the definition of learnability that the *cardinality* (and not the size) of the characteristic sample is polynomially bounded by the size of the concept, hence by the size of the alphabet.

Additionally to the conditions imposed by the definition of learnability, we are interested in the existence of learning algorithms which return *minimal* concepts for a given set of examples. It is important to emphasize that we mean minimality in terms on language inclusion. In particular, when only positive examples are allowed, a DIMS S is a *minimal* DIMS consistent with a set of trees D iff $D \subseteq L(S)$, and, for every $S' \neq S$, if $D \subseteq L(S')$, then $L(S') \not\subset L(S)$. We similarly obtain the definition of minimality for learning

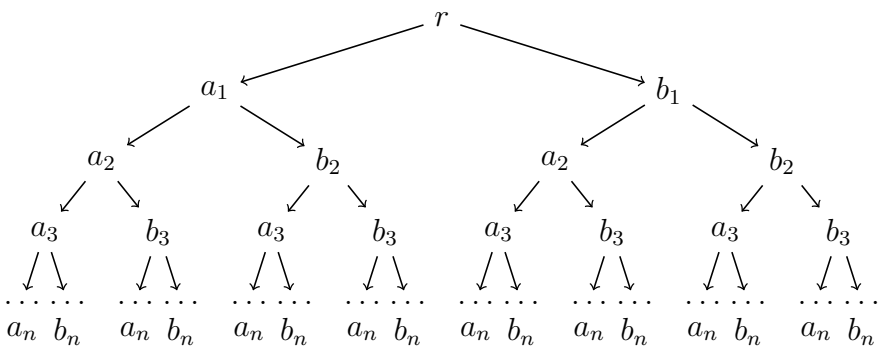


Figure 3.2: The unique tree satisfying the schema from Example 3.2.3.

DIMEs. Intuitively, a minimal schema consistent with a set of examples is the most specific schema consistent with them. For example, recall the three XML documents storing information about books from Figure 3.1. Assume that the user provides the three documents as positive examples to a learning algorithm. The most specific schema using simple multiplicities only (i.e., only $+$, *, 1, ?) and consistent with the examples is:

$$book \rightarrow title \parallel year^{?} \parallel (author^{+} \mid editor^{+}).$$

Another possible solution is the schema:

$$book \rightarrow title \parallel year^{?} \parallel author^{*} \parallel editor^{*}.$$

It is less likely that a user wants to obtain such a schema which allows a book to have at the same time author and editor. In this case, the most specific schema also corresponds to the natural requirements that one might want to impose on an XML collection storing information about books, in particular a book has either at least one author or at least one editor. Minimality is often perceived as a better fitted learning solution [Ang80, Ang82, BGNV10, GV90], and this motivates our requirement for the learning algorithms to return minimal concepts consistent with the examples.

3.3 Intractability of the general case

In this section, we prove that DIMSs are not learnable in the presence of both positive and negative examples, or more formally, the concept class *DIMS* is not learnable in polynomial time and data in the setting $DIMS^{\pm} = (Tree^{\pm}, DIMS, L^{\pm})$. For this purpose, we prove that the consistency checking is intractable for DIMEs, i.e., in the setting $DIME^{\pm} = (W_{\Sigma}^{\pm}, DIME, L^{\pm})$.

As already pointed out in Definition 3.2.2, we aim at a polynomial learning algorithm that returns a concept (e.g., DIME or DIMS) that selects all positive examples and none of the negative ones. On the other hand, the learning algorithm should return *null* if and only if there is no such concept. Consequently, we first study the *consistency checking* problem (i.e., deciding whether a consistent concept exists), which is a fundamental problem underlying learning. Since consistency checking is an "easier" problem than learning, its intractability precludes learnability. Formally, given a learning setting $\mathcal{K} = (\mathcal{E}, \mathcal{C}, \mathcal{L})$, the $\mathcal{K}$-*consistency* is the following decision problem:

$$CONS_{\mathcal{K}} = \{D \subseteq \mathcal{E} \mid \exists c \in \mathcal{C}.\ D \subseteq \mathcal{L}(c)\}.$$

Note that the consistency checking is trivial when only positive examples are allowed. For instance, if we want to learn DIMEs from positive examples over the alphabet $\{a_1, \ldots, a_n\}$, the DIME $a_1^* \parallel \ldots \parallel a_n^*$ is always consistent with the examples. When we also allow negative examples, the problem becomes more complex and it is intractable in general.

First, let us prove the intractability of $CONS_{DIME^{\pm}}$. Intuitively, this follows from the fact that, given a set of unordered words, there may exist an exponential number of consistent DIMEs, and we may need to check all of them to decide whether there exist negative examples satisfying them. Formally, we have the following result:

Lemma 3.3.1 *$CONS_{DIME^{\pm}}$ is NP-complete.*

Proof To prove the membership of $CONS_{DIME^{\pm}}$ to NP, we point out that a Turing machine guesses a DIME E, whose size is linear in $|\Sigma|$ since repetitions are discarded among the disjunctions of E. Moreover, checking whether E is consistent with the sample can be easily done in polynomial time.

We prove the NP-hardness by reduction from 3SAT which is known as being NP-complete. We take a formula φ in 3CNF containing the clauses $C_1, \ldots, C_k$ over the variables $x_1, \ldots, x_n$. We generate a sample D_φ over the alphabet $\Sigma = \{t_1, f_1, \ldots, t_n, f_n\}$ such that:

- $(t_1 f_1 \ldots t_n f_n, +) \in D_\varphi$,
- $(\varepsilon, -) \in D_\varphi$,
- $(t_i f_i, +), (t_i t_i f_i f_i, -) \in D_\varphi$, for $1 \leqslant i \leqslant n$,
- $(w_j, -) \in D_\varphi$, where $w_j = v_{j1} v_{j1} v_{j2} v_{j2} v_{j3} v_{j3}$, for every j such that $1 \leqslant j \leqslant k$, where x_{j1}, x_{j2}, x_{j3} are the literals used in the clause C_j and for every l such that $1 \leqslant l \leqslant 3$, v_{jl} is t_{jl} if x_{jl} is a negative literal in C_j, and f_{jl} otherwise.

For example, for the formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$, we generate the sample:

$$\begin{array}{cc} (t_1f_1t_2f_2t_3f_3t_4f_4, +), & (\varepsilon, -), \\ (t_1f_1, +), & (t_1t_1f_1f_1, -), \\ (t_2f_2, +), & (t_2t_2f_2f_2, -), \\ (t_3f_3, +), & (t_3t_3f_3f_3, -), \\ (t_4f_4, +), & (t_4t_4f_4f_4, -), \\ & (f_1f_1t_2t_2f_3f_3, -), \\ & (t_1t_1f_3f_3t_4t_4, -). \end{array}$$

For a given φ, a valuation is a function $V : \{x_1, \ldots, x_n\} \to \{true, false\}$. Each of the 2^n possible valuations encodes a DIME E_V consistent with the positive examples from D_φ, constructed as follows:

$$E_V = (v_1 \mid \ldots \mid v_n)^+ \parallel \overline{v_1}^? \parallel \ldots \parallel \overline{v_n}^?,$$

where, for $1 \leqslant i \leqslant n$, if $V(x_i) = true$ then $v_i = t_i$ and $\overline{v_i} = f_i$. Otherwise, $v_i = f_i$ and $\overline{v_i} = t_i$. Next, we show that, for every valuation V, $V \models \varphi$ iff E_V is consistent with D_φ.

For the *only if* case, consider a valuation V such that $V \models \varphi$ and we take the corresponding expression $E_V = (v_1 \mid \ldots \mid v_n)^+ \parallel \overline{v_1}^? \parallel \ldots \parallel \overline{v_n}^?$. Note that $t_1f_1 \ldots t_nf_n$ and all t_if_i's (with $1 \leqslant i \leqslant n$) satisfy E_V, while ε does not satisfy E_V. Also note that for $1 \leqslant i \leqslant n$, one symbol between t_i and f_i occurs at least once, while the other occurs at most once, so all $t_it_if_if_i$'s do not satisfy E_V. Assume that there is a w_j (with $1 \leqslant j \leqslant k$) such that w_j satisfies E_V, which by construction implies that the clause C_j is not satisfied by the valuation V, which implies a contradiction. Hence, w_j does not satisfy E_V for every $1 \leqslant j \leqslant k$. Therefore, E_V is consistent with D_φ.

For the *if* case, we assume that E_V is consistent with the sample D_φ. Since the w_j's (with $1 \leqslant j \leqslant k$) encode the valuations making the clauses C_j's false and none of the w_j's satisfies E_V, then the valuation V encoded in E_V makes the formula φ satisfiable.

The construction of D_φ also ensures that if there exists a DIME consistent with D_φ, it has the form of E_V. Therefore, $\varphi \in$3SAT iff $D_\varphi \in CONS_{DIME^\pm}$. □

We extend the above result to $CONS_{DIMS^\pm}$:

Corollary 3.3.2 *$CONS_{DIMS^\pm}$ is NP-complete.*

Proof The NP-hardness of $CONS_{DIME^\pm}$ implies that $CONS_{DIMS^\pm}$ is also NP-hard: it is sufficient to consider flat trees having all the same root label. Moreover, to prove the membership of $CONS_{DIMS^\pm}$ to NP, a Turing machine guesses a DIMS S, whose size is polynomial in $|\Sigma|$, and checks whether S is consistent with the sample (which can be done in polynomial time). □

Since consistency checking in the presence of positive and negative examples is intractable for DIMS, we conclude that there does not exist an algorithm able to answer *null* in polynomial time if an inconsistent sample is provided. Consequently, we have the following result.

Theorem 3.3.3 *The concept class DIMS is not learnable in polynomial time and data from positive and negative examples i.e., in the setting* ($Tree^{\pm}$, *DIMS*, $L^{\pm}$).

Next, our goal is to find restrictions that allow learnability and we observe that the negative examples and the disjunction operator play central roles in the proof of the intractability of the consistency checking (cf. Lemma 3.3.1). Hence, there are two natural ideas to address this intractability: (i) to forbid the negative examples and (ii) to forbid disjunction. We exploit these two ideas to find learnable restrictions in Section 3.4 and Section 3.5, respectively.

Additionally, we observe that if we require the learning algorithm to be minimal and we still want to learn languages where a certain label is presented an arbitrary number of times, it is necessary to drop arbitrary intervals. For instance, if we have as positive examples a, a^3, a^{10} and no negative example, we consider that a^+ is a more desirable solution than $a^{[1,10]}$, which is the minimal consistent DIME. Nonetheless, the two learnable restrictions that we identify and that do not use arbitrary intervals are still of practical interest, as we point out in Section 3.4 and Section 3.5, respectively.

3.4 Learning from positive examples only

In this section, we identify a restriction that is learnable from positive examples only. More precisely, we first introduce in Section 3.4.1 the studied restriction of DIMEs (that we denote DMEs). Then, we prove that the DMEs and the schema they define (that we denote DMSs) are learnable from positive examples only, in Section 3.4.2 and Section 3.4.3, respectively.

3.4.1 Simple disjunctive multiplicity expressions (DMEs)

As already mentioned, we have decided to drop the arbitrary interval multiplicities and to consider in the remainder only the following simple multiplicities: $*, +, ?, 0$, and 1, which are macros for $[0, \infty]$, $[1, \infty]$, $[0, 1]$, $[0, 0]$, and $[1, 1]$, respectively. Then, a *simple disjunctive multiplicity expression* (*DME*) is essentially a DIME using simple multiplicities only and more limited nesting of disjunction and unordered concatenation. Formally, a DME

E is:

$$E := D_1^{M_1} \parallel \ldots \parallel D_n^{M_n},$$

where for every $1 \leqslant i \leqslant n$, M_i is a multiplicity and D_i is:

$$D_i := a_1^{M'_1} \mid \ldots \mid a_k^{M'_k},$$

where for every $1 \leqslant j \leqslant k$, M'_j is a multiplicity and $a_j \in \Sigma$. Since DMEs are restricted DIMEs hence restricted UREs, the semantics are defined as in Section 1.2. Then, a *simple disjunctive multiplicity schema* (*DMS*) is a DIMS that employs only DMEs. In particular, the examples of schemas that we have shown in Section 1.1 and Section 3.1 are DMSs. It is important to notice that in the proof of the intractability of the consistency checking (cf. Lemma 3.3.1), all considered DIMEs are in fact DMEs, which makes DMEs and DMSs not learnable when negative examples are also allowed.

Excepting the use of simple multiplicities instead of arbitrary interval multiplicities, the other difference between DMEs and DIMEs is that in DMEs we can use the disjunction only near symbols with multiplicities a^M and not near unordered concatenations of symbols with multiplicities. For instance, $(a \parallel b) \mid c$ is a DIME but not a DME. However, it seems that these restrictions do not preclude the practical interest of DMEs. In particular, by using the same approach as in Section 1.8 for analyzing the expressiveness of DIMEs, we observe that all 77 regular expressions of the XMark benchmark [SWK+02] are captured by DMEs and that 84% of the regular expressions from the University of Amsterdam XML Web Collection [GM13] are captured by DMEs [BCS13]. These numbers suggest that studying the learnability of DMEs is relevant from a practical point of view.

Recall that every DIME can be alternatively expressed by its characterizing tuple (cf. Section 1.4.1). We observe that the construction of characterizing tuples is simplified for DMEs, as we illustrate next on $E_0 = a^+ \parallel (b \mid c) \parallel d^?$:

- Recall that C_E contains pairs of symbols such that E defines no unordered word using both symbols simultaneously: $C_{E_0} = \{(b, c), (c, b)\}$.

  Since in the DMEs we use the disjunction operator only near symbols with multiplicities a^M, we can compactly represent all conflicts using $\hat{C}_E$ that consists of sets of symbols present in E such that any pairwise two of them are conflicting. On our example, we obtain:

  $$\hat{C}_{E_0} = \{\{b, c\}\}.$$

- Similarly to general DIMEs, we use $\hat{N}_E$ that is a function mapping symbols to multiplicities such that for every unordered word $w \in L(E)$,

and for every symbol $a \in \Sigma$, $w(a) \in \hat{N}_E(a)$:

$$\hat{N}_{E_0}(a) = +, \quad \hat{N}_{E_0}(b) = \hat{N}_{E_0}(c) = \hat{N}_{E_0}(d) = ?.$$

- Recall that P_E consists of sets of required symbols: $P_{E_0} = \{\{a\}, \{b, c\}, \ldots\}$ (we have listed only its $\subseteq$-minimal elements). This can be compactly represented as
$$\hat{P}_{E_0} = \{\{a\}, \{b, c\}\},$$
which is in fact the unique non-redundant and covering subset of $P_E^{\subseteq\min}$ (cf. Section 1.4.3) because in the restricted grammar of DMEs we use the disjunction operator only near symbols with multiplicities a^M. Using the same argument, we observe that the set of counting dependencies K_E is always empty and hence we ignore it in the remainder.

Additionally, note that one can easily construct a DME from its compact characterizing tuple: a simple algorithm has to loop over the sets from $\hat{C}_E$ and $\hat{P}_E$ to compute for each label with which other labels it is linked by the disjunction operator. Then, using $\hat{N}_E$, the algorithm associates to each label and each disjunction the correct multiplicity. For instance, take the following compact tuple:

$$\hat{C}_{E_1} = \{\{a, e\}, \{c, d\}\}, \qquad \hat{P}_{E_1} = \{\{a, e\}, \{b\}\},$$
$$\hat{N}_{E_1}(a) = *, \quad \hat{N}_{E_1}(b) = 1, \quad \hat{N}_{E_1}(c) = \hat{N}_{E_1}(d) = \hat{N}_{E_1}(e) = ?.$$

Note that they characterize the expression:

$$E_1 = (a^+ \mid e) \parallel b \parallel (c^? \mid d^?).$$

We have recalled the alternative definition with characterizing tuples and shown how it adapts for DMEs because we next propose an algorithm which learns characterizing tuples from unordered word examples (Algorithm 2 from Section 3.4.2). Then, from this information, the corresponding DME can be constructed in a straightforward manner.

3.4.2 Learning DMEs from positive examples

In this section, we study the problem of learning a DME from positive examples (i.e., in the setting (W_Σ, DME, L)) and we present a learning algorithm that constructs a minimal DME consistent with the input collection of unordered words. Given a set of unordered words, there may exist many consistent minimal DMEs. In fact, for some sets of positive examples there may be an exponential number of such expressions (cf. the proof of Lemma 3.3.1). Take in Example 3.4.1 a sample and two consistent minimal DMEs.

Example 3.4.1 Consider the alphabet $\Sigma = \{a, b, c, d, e\}$ and the set of unordered words $D = \{aabc, abd, be\}$. Take the following two DMEs:

$$E_1 = (a^+ \mid e) \parallel b \parallel (c^? \mid d^?),$$
$$E_2 = a^* \parallel b \parallel (c \mid d \mid e).$$

Note that $D \subseteq L(E_1)$ and $D \subseteq L(E_2)$. Also note that $L(E_1) \not\subseteq L(E_2)$ (because of bce) and $L(E_2) \not\subseteq L(E_1)$ (because of abe). On the other hand, we easily observe that both E_1 and E_2 are minimal DMEs with languages including D. □

Before we present the learning algorithms, we have to introduce additional notions. First, we define the function *min_fit_multiplicity* which, given a set of unordered words D and a label $a \in \Sigma$, computes the multiplicity M such that $\forall w \in D.\ w(a) \in M$ and there does not exist another multiplicity M' such that $M' \subset M$ and $\forall w \in D.\ w(a) \in M'$. For example, given the set of unordered words $D = \{aabc, abd, be\}$, we have:

$$min\_fit\_multiplicity(D, a) = *,$$
$$min\_fit\_multiplicity(D, b) = 1,$$
$$min\_fit\_multiplicity(D, c) = ?.$$

Next, we introduce the notion of *maximal-clique partition of a graph*. Given a graph $G = (V, E)$, a maximal-clique partition of G is a graph partition $(V_1, \ldots, V_k)$ such that:

- The subgraph induced in G by any V_i is a clique (with $1 \leqslant i \leqslant k$),
- The subgraph induced in G by the union of any V_i and V_j is not a clique (with $1 \leqslant i \neq j \leqslant k$).

In Figure 3.3 we present a graph and a maximal-clique partition of it i.e., $\{\{a, e\}, \{b\}, \{c, d\}\}$. Note that the graph from Figure 3.3 allows one other maximal-clique partition i.e., $\{\{a\}, \{b\}, \{c, d, e\}\}$. On the other hand, $\{\{a\}, \{b\}, \{c, d\}, \{e\}\}$ is not a maximal-clique partition because it contains two sets such that their union induces a clique i.e., $\{a\}$ and $\{e\}$.

Unlike the *clique* problem, which is known to be NP-complete [Pap94], we can partition in polynomial time a graph in maximal cliques with a greedy algorithm. In the sequel, we assume that the vertices of the graph are labels from Σ. For a given graph there may exist many maximal-clique partitions and we use the total order $<_\Sigma$ to propose a deterministic algorithm constructing a maximal-clique partition. The algorithm works as follows: we

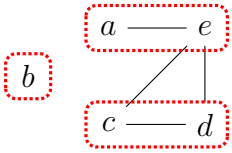


Figure 3.3: A graph and a maximal-clique partition of it. Vertices from the same rectangle belong to the same set.

take the smallest label from Σ w.r.t. $<_\Sigma$ and not yet used in a clique, and we iteratively extend it to a maximal clique by adding connected labels. Every time when we have a choice to add a new label to the current clique, we take the smallest label w.r.t. $<_\Sigma$. We repeat this until all the labels are used. This algorithm yields a unique maximal-clique partition. For example, for the graph from Figure 3.3, we compute the maximal-clique partition marked on the figure i.e., $\{\{a, e\}, \{b\}, \{c, d\}\}$. We additionally define the function *max_clique_partition* which takes as input a graph, computes a maximal-clique partition using the greedy algorithm described above and, at the end, for technical reasons, the algorithm discards the singletons. For example, for the graph from Figure 3.3, the function *max_clique_partition* returns $\{\{a, e\}, \{c, d\}\}$. Clearly, the function *max_clique_partition* works in polynomial time.

Next, we present Algorithm 2 and we claim that, given a set of unordered words D, it computes in polynomial time a DME E consistent with D.

Algorithm 2 Learning DMEs from positive examples.

algorithm $learner^{+}_{DME}(D)$
Input: A set of unordered words $D = \{w_1, \ldots, w_n\}$
Output: A minimal DME E consistent with D

1: **for** $a \in \Sigma$ **do**
2: **let** $\hat{N}_E(a) = min\_fit\_multiplicity(D, a)$
3: **let** $\Sigma' = \{a \in \Sigma \mid \hat{N}_E(a) \in \{?, 1, *, +\}\}$
4: **let** $G = (\Sigma',\ \{(a, b) \in \Sigma' \times \Sigma' \mid \forall w \in D.\ a \notin w \vee b \notin w\})$
5: **let** $\hat{C}_E = max\_clique\_partition(G)$
6: **let** $\hat{P}_E = \{\{a\} \mid \hat{N}_E(a) \in \{1, +\}\} \cup \{X \in \hat{C}_E \mid \forall w \in D.\ \exists a \in X.\ a \in w\}$
7: **return** E characterized by $(\hat{C}_E, \hat{N}_E, \hat{P}_E)$

Algorithm 2 works in three steps and we illustrate each of them on the sample $D = \{aabc, abd, be\}$ from Example 3.4.1. The first step (lines 1-2) computes the compact representation of the extended cardinality map for each symbol from Σ, using the function *min_fit_multiplicity*. We ignore in the sequel the symbols never occurring in words from D (line 3). For the

sample from Example 3.4.1, we infer:

$$\hat{N}_E(a) = *, \qquad \hat{N}_E(b) = 1,$$
$$\hat{N}_E(c) = \hat{N}_E(d) = \hat{N}_E(e) = ?.$$

The second step of the algorithm (lines 4-5) computes the compact sets of conflicting siblings. First, we construct the graph G having as set of vertices the labels occurring at least once in unordered words from D. Two labels are linked by an edge in G if there does not exist an unordered word in D where both of them are present at the same time, in other words the two labels are a candidate pair of conflicting siblings. Next, we apply the function *max_clique_partition* on the graph G. For the unordered words from Example 3.4.1 we obtain the graph from Figure 3.3, and we infer $\hat{C}_E = \{\{a, e\}, \{c, d\}\}$. Note that the maximal-clique partition implies the minimality of the DME constructed later using the inferred $\hat{C}_E$.

The third step of the algorithm (line 6) computes the $\subseteq$-minimal sets of required symbols $\hat{P}_E$. Each symbol having associated a multiplicity 1 or + belongs to a required set of symbols containing only itself because it is present in all the unordered words from D and we want to learn a minimal concept. Moreover, we add in $\hat{P}_E$ the sets of conflicting siblings inferred at the previous step with the property that one of them is present in any unordered word from D, to guarantee the minimality of the inferred language. For the sample from Example 3.4.1, $\{b\}$ belongs to $\hat{P}_E$. Since from the previous step we have $\hat{C}_E = \{\{a, e\}, \{c, d\}\}$, at this step we have to add $\{a, e\}$ to $\hat{P}_E$ because all the words in the sample contain either a or e. On the other hand, we do not add $\{c, d\}$ because the sample contains the word be. The inferred $\hat{P}_E$ is $\{\{a, e\}, \{b\}\}$.

Finally, the algorithm returns the DME characterized by the inferred tuple (line 7). For the sample D, it returns $E = (a^+ \mid e) \parallel b \parallel (c^? \mid d^?)$. Note that if at step 2 we take a partition which is not a maximal-clique one, for example $\{\{a\}, \{b\}, \{c, d\}, \{e\}\}$, and we later construct a DME using it, we get $a^* \parallel b \parallel (c^? \mid d^?) \parallel e^?$, which includes both E_1 and E_2 from Example 3.4.1, therefore is not minimal. Also note that at step 3, without $\{a, e\}$ added to $\hat{P}_E$, the resulting schema would accept an unordered word without any a and e, so the learned language would not be minimal.

Algorithm 2 is sound and each of its three steps requires polynomial time. Next, we prove the completeness of the algorithm. Given a DME E, we construct in three steps its characteristic sample CS_E. At the same time, we illustrate the construction on the DME $E_1 = (a^+ \mid e) \parallel b \parallel (c^? \mid d^?)$:

1. We take the pairs of symbols which can be found together in an unordered word in $L(E)$. For each of them, we add in CS_E an unordered

word containing only the two symbols. Next, for each symbol occurring in the disjunctions from E, we add in CS_E an unordered word containing only one occurrence of that symbol. We also add in CS_E the empty word. For E_1 we obtain: $\{ab, ac, ad, bc, bd, be, ce, de, a, b, c, d, e, \varepsilon\}$.

2. We replace each unordered word w obtained at the previous step with $w \uplus w'$, where w' is a minimal unordered word such that $w \uplus w' \in L(E)$. The newly obtained CS_E contains unordered words from $L(E)$. For E_1 we obtain: $\{ab, abc, abd, be, bce, bde\}$.

3. For each symbol a from the alphabet such that $\hat{N}_E(a)$ is $*$ or $+$, we randomly take an unordered word w from CS_E and containing a and we add to CS_E the unordered word $w \uplus a$. In the worst case, at this step the number of words in the characteristic sample is doubled, but it remains polynomial in the size of the alphabet. For E_1 we obtain: $\{ab, aab, abc, abd, be, bce, bde\}$.

Note that there may exist many equivalent characteristic samples. The first step of the construction implies that the only potential conflicts to be considered in Algorithm 2 are the conflicts implied by the expression. In other words, all the connected components of the graph of potential conflicts from Algorithm 2 are cliques. Thus, there is only one possible maximal-clique partition to be done in the algorithm. Moreover, the second and third steps of the construction ensure that, for every sample consistently extending the characteristic sample, Algorithm 2 infers the correct sets of required symbols and the extended cardinality map, respectively.

We have proposed Algorithm 2, which is a polynomial, sound and complete algorithm for learning minimal DMEs from unordered words positive examples. Thus, we can state the following result:

Lemma 3.4.2 *The concept class DME is learnable in polynomial time and data from positive examples i.e., in the setting* (W_Σ, DME, L).

3.4.3 Learning DMSs from positive examples

In this section, we extend the result from the previous section and present Algorithm 3, which learns a minimal DMS consistent with the input set of trees. We assume w.l.o.g. that all the trees from the sample have as root label the same label r. If this assumption is not satisfied, the sample is not consistent. The algorithm infers, for each label a from the alphabet, the minimal DME consistent with the children of all the nodes labeled a from the trees from the sample.

Algorithm 3 Learning DMSs from positive examples.

algorithm: $learner^{+}_{DMS}(D)$
Input: A set of trees $D = \{t_1, \ldots, t_n\}$ s.t. $lab_{t_i}(root_{t_i}) = r$ (with $1 \leqslant i \leqslant n$)
Output: A minimal DMS S consistent with D

1: **for** $a \in \Sigma$ **do**
2: **let** $D' = \{ch^n_t \mid t \in D.\ n \in N_t.\ lab_t(n) = a\}$
3: **let** $R_S(a) = learner^{+}_{DME}(D')$
4: **return** $S = (r, R_S)$

Algorithm 3 returns a minimal DMS consistent with the sample because the inferred rule for each label represents a minimal DME obtained using Algorithm 2. Next, we show that Algorithm 3 is also complete by providing a construction of a characteristic sample of cardinality polynomial in the size of the alphabet. For this purpose, we have to define first two additional notions. Given a DMS $S = (root_S, R_S)$ and a label $a \in \Sigma$, we define the following two trees:

- $\min_{t\uparrow(S,a)}$ is a minimal tree satisfying S and containing a node labeled a,
- $\min_{t\downarrow(S,a)}$ is a minimal tree satisfying $S' = (a, R_S)$. It is equivalent to $\min_{t\uparrow(S',a)}$.

We illustrate the two notions defined above in the following example:

Example 3.4.3 Consider the DMS S having the root label r and the rules:

$$r \to a^* \parallel (b \mid c) \qquad a \to d^?$$
$$b, c \to e^+ \qquad d, e \to \epsilon$$

We present in Figure 3.4 some trees and we precise for each of them how it can be used. □

Next, we present the construction of the characteristic sample for learning a DMS from positive examples. We take a DMS $S = (root_S, R_S)$ over an alphabet Σ and we assume w.l.o.g. that any symbol of the alphabet can be present in at least one tree from $L(S)$. For each $a \in \Sigma$, for each $w \in CS_{R_S(a)}$, we compute a tree t as follows: we generate a tree $\min_{t\uparrow(S,a)}$, we take the node labeled by a (let it n_a), and for every $b \in \Sigma$, while $ch^{n_a}_t(b) < w(b)$ we fuse in n_a a copy of $\min_{t\downarrow(S,b)}$. We obtain a sample of cardinality polynomially bounded by the size of the alphabet. Given a DMS S, there may exist many characteristic samples CS_S. Each of them has the property

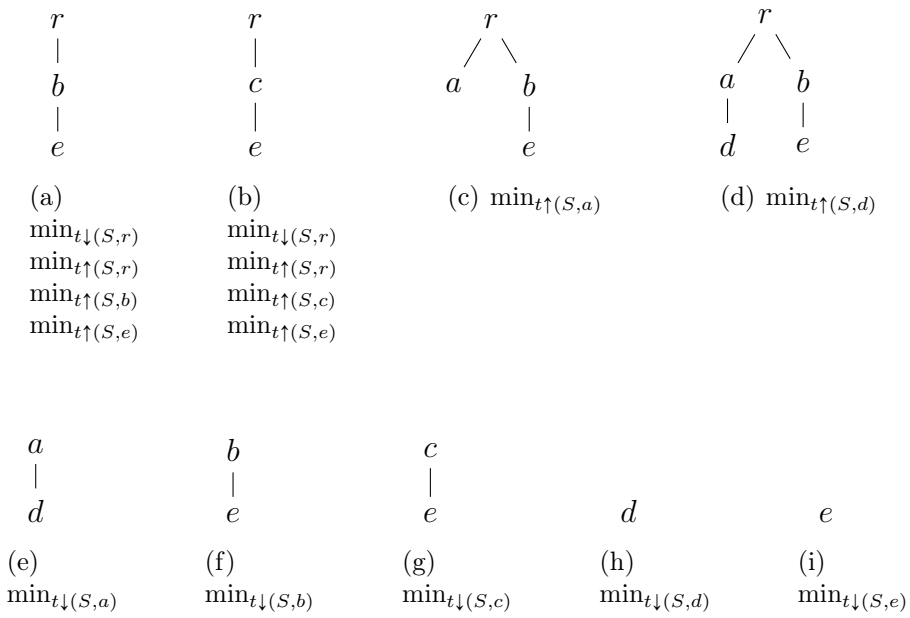


Figure 3.4: Trees used for Example 3.4.3.

that, if we construct a sample D which extends CS_S consistently with S, then $learner^{+}_{DMS}(D)$ returns S. This proves the completeness of Algorithm 3.

We illustrate the construction of the characteristic sample on the schema S from Example 3.4.3. Recall that we have already presented the trees $\min_{t\uparrow(S,a)}$ and $\min_{t\downarrow(S,a)}$ for each a from the alphabet. We also construct the characteristic samples for the DMEs from the rules of S:

- $CS_{R_S(r)} = \{aab, ab, ac, b, c\}$,
- $CS_{R_S(a)} = \{\varepsilon, d\}$,
- $CS_{R_S(b)} = CS_{R_S(c)} = \{e, ee\}$,
- $CS_{R_S(d)} = CS_{R_S(e)} = \{\varepsilon\}$.

In Figure 3.5 we present a characteristic sample CS_S for the DMS S and we explain the purpose of each tree:

- (a), (b), (c), (d), and (e) ensure that there is inferred the correct rule for the root i.e., $R_S(r)$,
- (b) and (f) ensure that there is inferred the correct $R_S(a)$,
- (d) and (g) ensure that there is inferred the correct $R_S(b)$,

- (e) and (h) ensure that there is inferred the correct $R_S(c)$,
- The nodes labeled by d and e never have children in the trees from CS_S, hence there are inferred the correct rules for $R_S(d)$ and $R_S(e)$.

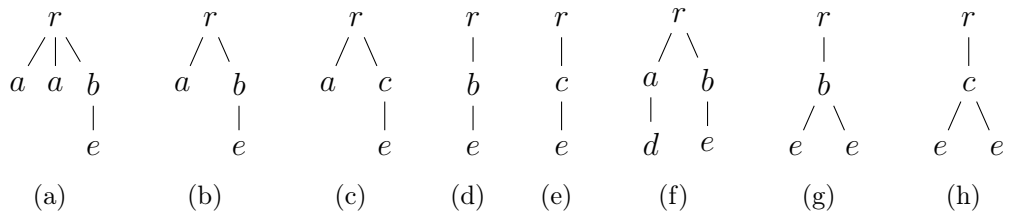


Figure 3.5: Characteristic sample for the schema S from Example 3.4.3.

We have proposed Algorithm 3, which is a polynomial, sound, and complete algorithm for learning DMSs from trees positive examples. Thus, we can state the main result of this section:

Theorem 3.4.4 *The concept class DMS is learnable in polynomial time and data from positive examples i.e., in the setting* $(Tree, DMS, L)$.

3.5 Learning from positive and negative examples

In this section, we present a restriction that is learnable from both positive and negative examples.

A *simple disjunction-free multiplicity expression* (*ME*) is a DME that does not use disjunction i.e., is of the form $a_1^{M_1} \parallel \ldots \parallel a_n^{M_n}$. Then, a *simple disjunction-free multiplicity schema* (*MS*) is a DMS that uses MEs only. Due to this very particular form, we can *capture* a MS $S = (root_S, R_S)$ using a function $\mu : \Sigma \times \Sigma \to \{0, 1, ?, +, *\}$ obtained directly from the rules of S:

$$a \to a_1^{\mu(a,a_1)} \parallel \ldots \parallel a_n^{\mu(a,a_n)}.$$

For example, given the schema S having the root r and the rules:

$$r \to a^+ \parallel b, \qquad a \to b^*, \qquad b \to a^? \parallel b^?,$$

we have :

$$\begin{array}{lll} \mu(r,a) = +, & \mu(r,b) = 1, & \mu(r,r) = 0, \\ \mu(a,a) = 0, & \mu(a,b) = *, & \mu(a,r) = 0, \\ \mu(b,a) = ?, & \mu(b,b) = ?, & \mu(b,r) = 0. \end{array}$$

Note that given the function μ we can easily construct the initial S. Even though MEs may seem very restrictive, by using the same approach as in Section 1.8 for analyzing the expressiveness of DIMEs, we observe that 76 of the 77 regular expressions of the XMark benchmark [SWK+02] are captured by MEs and that 74.6% of the regular expressions from the University of Amsterdam XML Web Collection [GM13] are captured by MEs [BCS13]. These numbers suggest that studying the learnability of MEs is still relevant from a practical point of view. We show next that we are able to learn MEs and MSs in the presence of both positive and negative examples.

The main insight is that given a set of unordered words, there exists a *unique minimal ME* consistent with them, which can be computed immediately using the function *min_fit_multiplicity* (cf. Section 3.4.2). Hence, consistency checking for MEs becomes tractable: given a sample containing positive and negative unordered word examples, there exists a ME consistent with them iff no unordered word used as negative example satisfies the minimal ME selecting all positive unordered words.

Similarly, take a set of trees labeled as positive and negative examples, all of them having the same root label r (if this assumption is not satisfied, the sample is not consistent). Then, we observe that there exists a *unique minimal MS* consistent with the positive tree examples. This makes the consistency checking tractable for MS too: given a sample containing positive and negative tree examples, there exists a MS consistent with them iff no tree used as negative example satisfies the minimal MS selecting all positive trees.

We use these simple ideas to propose Algorithm 4, which is polynomial and learns a minimal MS from a set of trees. In the algorithm, we first take all trees labeled as positive examples (line 1). Then, we compute the minimal MS selecting all positive examples using the function *min_fit_multiplicity* for all pairs of symbols (lines 2-6). If this MS selects a negative example, this means that the sample is inconsistent and *null* is returned (line 7-8); otherwise, we return the minimal consistent MS (line 9). The minimality of Algorithm 4 follows from the minimality of the inferred multiplicity for each pair of labels (a, b), using the function *min_fit_multiplicity* (cf. Section 3.4.2). Then, to show the completeness of Algorithm 4, we can easily construct a characteristic sample of cardinality polynomial in the size of the alphabet by using the same techniques as in Section 3.4.3, for unordered words and for trees. We have proposed a polynomial, sound, and complete algorithm which learns minimal MS from positive and negative examples, thus we can state the following result.

Theorem 3.5.1 *The concept class MS is learnable in polynomial time and data from positive and negative examples i.e., in the setting* $(Tree^{\pm}, MS, L^{\pm})$.

Algorithm 4 Learning MS from positive and negative examples.

algorithm $learner^{\pm}_{MS}(D)$
Input A sample $D = \{(t, \alpha) \mid t \in Tree, lab_t(root_t) = r, \alpha \in \{+, -\}\}$
Output A minimal MS S such that $D \subseteq L^{\pm}(S)$, or *null* if no such MS exists.

```
1: let D_+ = {t ∈ Tree | (t, +) ∈ D}
2: for a ∈ Σ do
3:     let D' = {ch_t^n | t ∈ D_+. n ∈ N_t. lab_t(n) = a}
4:     for b ∈ Σ do
5:         let μ(a, b) = min_fit_multiplicity(D', b)
6: let S be the MS having the root label r and captured by μ
7: if ∃t ∈ Tree. (t, −) ∈ D ∧ t ∈ L(S) then
8:     return null
9: return S
```

3.6 Related work

The *Document Type Definition (DTD)*, the most widespread XML schema formalism [GM13, BNVdB04], is essentially a set of rules associating with each label a regular expression that defines the admissible sequences of children. Therefore, learning DTDs reduces to learning regular expressions, which are not identifiable in the limit [Gol78]. Consequently, research has been done on restricted classes of regular expressions that can be efficiently learnable [MAC03, GGR+03, BNST06, BNSV10, BGNV10, FK13].

For instance, Bex et al. [BNST06, BNSV10] proposed learning algorithms for two classes of regular expressions that capture many practical DTDs and are succinct by definition: *single occurrence regular expressions* (SOREs) and its subclass consisting of *chain regular expressions* (CHAREs). Bex et al. [BGNV10] also studied learning algorithms for the subclass of deterministic regular expressions where each alphabet symbol occurs at most k times (k-OREs). More recently, Freydenberger and Kötzing [FK13] proposed new learning algorithms for the aforementioned restricted classes of regular expressions.

Since the DIMEs disallow repetitions of symbols, they can be seen as restricted SOREs interpreted under commutative closure i.e., an unordered collection of children matches a regular expression if there exists an ordering that matches the regular expression in the standard way. The algorithms proposed for the inference of SOREs [BNSV10, FK13] are typically based on constructing an automaton and then transforming it into an equivalent SORE. Being based on automata techniques, the algorithms for learning SOREs take ordered input, therefore an additional input that the DIMEs

do not have i.e., the order among the labels. For this reason, we cannot reduce learning DIMEs to learning SOREs. Consequently, we investigated new techniques to solve the problem of learning unordered schemas. Moreover, all existing learning algorithms take into account only positive examples.

We also mention some work done on learning schema formalisms more expressive than DTDs. *XML Schema*, the second most widespread schema formalism [GM13, BNVdB04], allow the content model of an element to depend on the context in which it is used, therefore it is more difficult to learn. Bex et al. [BNV07] proposed efficient algorithms to automatically infer a concise XML Schema describing a given set of XML documents.

Chapter 4

Learning relational join queries

In this chapter, we investigate the problem of learning relational join queries from examples given by the user. The user is presented with a set of candidate tuples and is asked to label them as *positive* or *negative* examples, depending on whether or not she would like the tuples as part of the join result. The goal is to quickly infer an arbitrary n-ary join predicate across an arbitrary number m of relations while keeping the number of user interactions as minimal as possible. We assume no prior knowledge of the integrity constraints across the involved relations. Inferring the join predicate across multiple relations when the referential constraints are unknown may occur in several applications such as data integration, reverse engineering of database queries, and schema inference. In such scenarios, the number of tuples involved in the join is typically large. We introduce a set of strategies that let us inspect the search space and aggressively prune what we call "uninformative" tuples, and directly present to the user the *informative* ones i.e., those that allow to quickly find the goal query that the user has in mind. We focus on the inference of joins with equality predicates, and we also allow disjunctive join predicates and projection in the queries. We precisely characterize the frontier between tractability and intractability for the following problems of interest in these settings: consistency checking, learnability, and deciding the informativeness of a tuple. Next, we propose several strategies for presenting tuples to the user in a given order that lets minimize the number of interactions. We show the efficiency and scalability of our approach through an experimental study on both benchmark and synthetic datasets.

4.1 Context

Join specification may become feasible for non-expert users whenever they can easily access data and metadata altogether. This happens in traditional query specification paradigms, such as query-by-example [Zlo75], which are typically centered around a single database. When it comes to consider raw data coming from different data sources, such paradigms are not applicable any longer. The reason is twofold: (i) such data may not carry pertinent

metadata to be able to specify a join predicate and (ii) value-based matching of tuples is unfeasible in most cases, due to a massive number of tuples.

In this chapter, we consider very simple user input via Boolean membership queries ("Yes/No") to assist unfamiliar users to write their queries upon integrated data. In particular, we focus on two fundamental operators of any data integration or querying tool: *equijoins* – combining data from multiple sources, and *semijoins* – filtering data from one source based on the data from another source. Besides data integration, such operators are sensible in many other applications, such as reverse engineering of database queries and constraint inference in case of limited knowledge of the database schemas. In particular, the queries that we investigate are of practical use in the context of denormalized databases having a small number of relations with large numbers of attributes.

Inference algorithms for schema mappings have been recently studied in [AtCKT11a, AtCKT11b] by leveraging data examples. However, such examples are expected to be provided by an expert user, namely the mapping designer, who is also responsible of selecting the mappings that best fit them. Query learning for relational queries with quantifiers has recently been addressed in [AAP+13, AHS12]. There, the system starts from an initial formulation of the query and refines it based on primary-foreign key relationships and the input from the user. We discuss in detail the differences with our work in the related work section (i.e., Section 4.7). To the best of our knowledge, ours is the first work that considers inference of joins via simple tuple labeling and with no knowledge of integrity constraints.

Consider a scenario where a user working for a travel agency wants to build a list of flight&hotel packages. The user is not acquainted with querying languages and can access the information on flights and hotels in a denormalized table, result of some data integration scenario, as in Figure 4.1.

The airline operating every flight is known and some hotels offer a discount when paired with a flight of a selected airline. Two queries can be envisioned: one that selects travel packages consisting of a flight and a stay in a hotel and another one that additionally ensures that the package is combined in a way allowing a discount. These two queries correspond to the following equijoin predicates:

$$To = City, \qquad (Q_1)$$
$$To = City \wedge Airline = Discount. \qquad (Q_2)$$

Note that since we assume no knowledge of the schema and of the integrity constraints, a number of other queries can possibly be formulated but we

	From	*To*	*Airline*	*City*	*Discount*	
	Paris	Lille	AF	NYC	AA	(1)
	Paris	Lille	AF	Paris	None	(2)
+	Paris	Lille	AF	Lille	AF	(3)
+	Lille	NYC	AA	NYC	AA	(4)
	Lille	NYC	AA	Paris	None	(5)
	Lille	NYC	AA	Lille	AF	(6)
	NYC	Paris	AA	NYC	AA	(7)
−	NYC	Paris	AA	Paris	None	(8)
	NYC	Paris	AA	Lille	AF	(9)
	Paris	NYC	AF	NYC	AA	(10)
	Paris	NYC	AF	Paris	None	(11)
	Paris	NYC	AF	Lille	AF	(12)

Figure 4.1: Integrated table.

remove them from consideration for the sake of simplicity and clarity of the example.

While the user may be unable to formulate her query, it is reasonable to assume that she can indicate whether or not a given pair of flight and hotel is of interest to her. We view this as labeling with + and − the tuples from the integrated table (Figure 4.1). For instance, suppose the user chooses the flight from Paris to Lille operated by Air France (AF) and the hotel in Lille. This corresponds to labeling by + the tuple (3).

Observe that both queries Q_1 and Q_2 are consistent with this labeling i.e., both queries select the tuple (3). Naturally, the objective is to use the labeling of further tuples to identify the goal query i.e., the query that the user has in mind. Not every tuple can however serve this purpose. For instance, if the user labels next the tuple (4) with +, both queries remain consistent. Intuitively, the labeling of the tuple (4) does not contribute any new information about the goal query and is therefore *uninformative*, an important concept that we formalize in this chapter. Since the input table may contain a large number of tuples, it may be unfeasible for the user to label every tuple.

For such a reason, we aim at limiting the number of tuples that the user needs to label in order to infer the goal query. More precisely, in this chapter we propose solutions that analyze and measure the potential information about the goal query that labeling a tuple can contribute and present to the user tuples that maximize this measure. In particular, since uninformative tuples do not contribute any additional information, they would not be presented to the user. In the example of the flight&hotel packages, a tuple

whose labeling can distinguish between Q_1 and Q_2 is, for instance, the tuple (8) because Q_1 selects it and Q_2 does not. If the user labels the tuple (8) with $-$, then the query Q_2 is returned; otherwise Q_1 is returned. We also point out that the use of only *positive* examples, tuples labeled with $+$, is not sufficient to identify all possible queries. As an example, query Q_2 is contained in Q_1, and therefore, satisfies all positive examples that Q_1 does. Consequently, the use of *negative* examples, tuples with label $-$, is necessary to distinguish between these two.

Since our goal is to minimize the number of interactions with the user, our research is of interest for novel database applications e.g., query processing using the crowd [FKK+11], where minimizing the number of interactions entails lower financial costs. In particular, crowdsourced joins have been mainly defined in terms of entity resolution, where joining two datasets means finding all pairs of tuples that refer to the same entity [MWK+11, WLK+13]. Conversely, our goal is to handle arbitrary n-ary join predicates, thus targeting a quite different and more intricate goal for the crowd i.e., inferring such join predicates from a set of positive and negative labels.

Moreover, our research also applies to schema mapping inference, assuming a less expert user than in [AtCKT11a, AtCKT11b]. Indeed, in our case the annotations correspond to simple membership queries [Ang88] to be answered even by a user who is not familiar with schema mappings.

Summarizing, the *main contributions* of this chapter are the following:

- We characterize the *learnability* of join queries using a definition based on the standard framework of *language identification in the limit with polynomial time and data* [Gol78]. Essentially, we show that the equijoins are learnable (with or without disjunction), while the semijoins are learnable only when disjunction is allowed. In particular, to prove that the semijoins without disjunction are not learnable, we have used the intractability of the consistency checking, a fundamental problem underlying learning i.e., to decide whether there exists a query consistent with a given set of examples. Thus, we precisely characterize the frontier between the learnable and the non learnable cases depending on whether or not we allow disjunction and/or projection.

- We focus on an interactive scenario inspired by the well-known framework of *learning with membership queries* [Ang88], we characterize the potential information that labeling a given tuple may contribute to the join inference process, and identify uninformative tuples. More precisely, we propose two notions of uninformativeness, one based on the knowledge of the goal query and one not based on this knowledge, and we show that the two notions are equivalent. Then, we prove that for

all aforementioned learnable cases, deciding whether a tuple is informative can be tested in polynomial time. Additionally, we show that this problem remains intractable for semijoins without disjunction.

- We propose a set of strategies for interactively inferring a goal join query, and we show their efficiency and scalability within an experimental study on both benchmark and synthetic data. More precisely, we have considered all queries of the TPC-H benchmark. Then, to cope with the absence of disjunction in the TPC-H queries, we have defined a set of synthetic queries using such operator and implemented a synthetic dataset inspired by our motivating example.

It is important to point out that this chapter is a substantially extended version of a conference paper [BCS14b], which is currently under journal submission [BCS14d]. More precisely, the problem setting considered in [BCS14b] is restricted to two relations on which only conjunctions of equality predicates can be learned. This chapter and [BCS14d] substantially extend the results of [BCS14b], by (i) allowing an arbitrary number of relations in the join inference and (ii) adding disjunction to the join predicates.

First, notice that the extension to an arbitrary number of relations is rather natural. For instance, in our motivating example one can consider multiple flights and multiple hotels e.g., a user may be interested in a round trip with a stay in a hotel in an intermediate city. We have considered such queries in a synthetic scenario in the experimental evaluation.

Second, to illustrate a case when the disjunction is useful, assume for instance that the user is interested in travel packages consisting of a flight and a stay in a hotel (either in the source or destination city), combined in a way allowing a discount. This query corresponds to the following disjunction of conjunctions of equijoin predicates:

$$(\mathit{From} = \mathit{City} \wedge \mathit{Airline} = \mathit{Discount}) \vee (\mathit{To} = \mathit{City} \wedge \mathit{Airline} = \mathit{Discount}).$$

To infer such a query, the user has to label on the instance from Figure 4.1 the tuples (3) and (7) as positive examples, and the tuples (8) and (11) as negative examples.

Organization. In Section 4.2, we introduce some preliminary notions. In Section 4.3, we define a framework for learning join queries from a given set of examples and analyze the complexity of two fundamental problems of interest: consistency checking and learnability. In Section 4.4, we describe the studied interactive scenario and investigate the problem of deciding the informativeness of a tuple. In Section 4.5, we propose practical strategies of

presenting tuples to the user, while in Section 4.6, we experimentally evaluate their performance. We discuss related work in Section 4.7.

4.2 Join queries

In this section, we define the basic concepts that we manipulate throughout the chapter.

Relations. A *schema* $\mathscr{S}$ is a finite set of *relations* $\mathscr{S} = \{R_1, \ldots, R_m\}$ with implicitly given sets of *attributes* $attrs(R_i)$ (for $1 \leqslant i \leqslant m$). We assume that these sets of attributes are pairwise disjoint. We assume no other knowledge of the database schema, in particular no knowledge of the integrity constraints between the relations. Given a schema $\mathscr{S}$, a *signature* is a subset of relations $\mathcal{R} \subseteq \mathscr{S}$. We naturally extend *attrs* to signatures i.e., given a signature $\mathcal{R}$, we have $attrs(\mathcal{R}) = \bigcup_{R \in \mathcal{R}} attrs(R)$.

Instances. We assume an infinite *domain* $\mathcal{U}$ that is a set of numerical constants with equality $=$ and inequality $\neq$ defined in the natural way. Then, a *tuple* t over a set of attributes $\{A_1, \ldots, A_k\}$ is a function $t : \{A_1, \ldots, A_k\} \to \mathcal{U}$ that associates a value of the domain to each attribute. We say that the set $\{A_1, \ldots, A_k\}$ is the signature of t, denoted $sig(t)$. Additionally, we say that a tuple t is *compatible* with a signature $\mathcal{R}$ if $sig(t) = attrs(\mathcal{R})$. Given two tuples t_1 and t_2 over disjoint signatures $\mathcal{R}_1$ and $\mathcal{R}_2$, respectively, by $t_1 \cdot t_2$ we denote the tuple t over the signature $\mathcal{R}_1 \cup \mathcal{R}_2$ such that $t[A] = t_1[A]$ for every attribute A from $attrs(\mathcal{R}_1)$ and $t[A] = t_2[A]$ for every attribute A from $attrs(\mathcal{R}_2)$. Furthermore, an *instance* of a schema $\mathscr{S}$ is a function that associates to each relation $R \in \mathscr{S}$ a finite set of tuples $\{t_1, \ldots, t_p\}$ such that $sig(t_i) = attrs(R)$ (for $1 \leqslant i \leqslant p$). For each relation $R \in \mathscr{S}$, we denote its corresponding set of tuples by $I(R)$. Moreover, given a signature $\mathcal{R} \subseteq \mathscr{S}$, by $I(\mathcal{R}) = \bigcup_{R \in \mathcal{R}} I(R)$ we denote the set of tuples from I corresponding to all relations from $\mathcal{R}$ and we refer to it as the instance of $\mathcal{R}$.

Example 4.2.1 In this example, for simplicity reasons, we use a schema of two relations $\mathscr{S}_0 = \{R_1, R_2\}$ with $attrs(R_1) = \{A_1, A_2\}$ and $attrs(R_2) = \{B_1, B_2, B_3\}$. Moreover, take the following instance I that contains 4 tuples for R_1 and 3 tuples for R_2.

$I(R_1) =$

	A_1	A_2
t_1	0	1
t_2	0	2
t_3	2	2
t_4	1	0

$I(R_2) =$

	B_1	B_2	B_3
t'_1	1	1	0
t'_2	0	1	2
t'_3	2	0	0

□

Queries. A *query* is basically a function that takes an instance and returns a set of tuples. More formally, a query q has an *input signature* $inSig(q)$ that is a non-empty set of input attributes and an *output signature* $outSig(q)$ that is a non-empty set of output attributes that we assume being a subset of the input signature i.e., $outSig(q) \subseteq inSig(q)$. We say that a query q is over a schema $\mathscr{S}$, or alternatively, is *compatible* with $\mathscr{S}$ if $inSig(q) \subseteq attrs(\mathscr{S})$. Then, given a query q over a schema $\mathscr{S}$ and an instance I of $\mathscr{S}$, the *answers* to q over I, denoted $q(I)$, is a finite set of tuples of signature $outSig(q)$.

In this chapter, we focus on four classes of *join queries* that we define in the remainder of this section. To this purpose, let us first define, for a schema $\mathscr{S}$, the set Ω such that

$$\Omega = \bigcup_{R,R' \in \mathscr{S}, R \neq R'} attrs(R) \times attrs(R').$$

Then, a *join predicate* is a subset $\theta \subseteq \Omega$. Moreover, a *disjunctive join predicate* is a union of join predicates i.e., a subset $\Theta \subseteq 2^{\Omega}$.

Classes of join queries. Given a schema $\mathscr{S}$, a *join query* q is essentially a triple that consists of a non-empty input signature $inSig(q) \subseteq \mathscr{S}$, a non-empty output signature $outSig(q) \subseteq inSig(q)$, and a (disjunctive) join predicate i.e., it can be of the form $(\mathcal{R}, \mathcal{R}_o, \theta)$ or $(\mathcal{R}, \mathcal{R}_o, \Theta)$, where $\mathcal{R} \subseteq \mathscr{S}$, $\mathcal{R}_o \subseteq \mathcal{R}$, $\mathcal{R} \neq \emptyset$, $\mathcal{R}_o \neq \emptyset$, and $\theta \subseteq \Omega$ or $\Theta \subseteq 2^{\Omega}$. Given a class of join queries $\mathcal{Q}$, we refer to $\mathcal{R}$ and $\mathcal{R}_o$ as the input signature $inSig(\mathcal{Q})$ and the output signature $outSig(\mathcal{Q})$ of the class of queries $\mathcal{Q}$, respectively.

Since we consider two possibilities for the relationship between $\mathcal{R}$ and $\mathcal{R}_o$ (they can differ or not) and two possibilities for the join predicates (they can be disjunctive or not), we obtain four classes of join queries that we define below, together with their semantics. To be able to formalize their semantics, we introduce first the *Cartesian product* of an instance I of a signature $\mathcal{R} \subseteq \mathscr{S}$ that is $D(\mathcal{R}, I) = \times_{R \in \mathcal{R}} I(R)$. In the rest of the chapter, we model the notion of integrated table (cf. Introduction) with Cartesian product. Moreover, in the remainder we use the terms Cartesian product and integrated table interchangeably.

The considered classes of queries are the following:

1. *Join*($\mathcal{R}$) (*equijoins*) – queries of the form $(\mathcal{R}, \mathcal{R}, \theta)$, where $\mathcal{R} \subseteq \mathscr{S}$, $\mathcal{R} \neq \varnothing$, and $\theta \subseteq \Omega$. We always present such queries as $(\bowtie_\theta \mathcal{R})$. Given an instance I of $\mathscr{S}$, we have:

$$(\bowtie_\theta \mathcal{R})(I) = \{t \in D(\mathcal{R}, I) \mid \forall (A, A') \in \theta.\ t[A] = t[A']\}.$$

2. *Join*$^{\ltimes}(\mathcal{R}, \mathcal{R}_o)$ (*semijoins*) – queries of the form $(\mathcal{R}, \mathcal{R}_o, \theta)$, where $\mathcal{R} \subseteq \mathscr{S}$, $\mathcal{R}_o \subset \mathcal{R}$, $\mathcal{R}_o \neq \varnothing$, and $\theta \subseteq \Omega$. We always present such queries as $(\mathcal{R}_o \ltimes_\theta (\mathcal{R} \backslash \mathcal{R}_o))$. Given an instance I of $\mathscr{S}$, we have:

$$(\mathcal{R}_o \ltimes_\theta (\mathcal{R} \backslash \mathcal{R}_o))(I) = \{t \in D(\mathcal{R}_o, I) \mid \exists t' \in D(\mathcal{R} \backslash \mathcal{R}_o, I).\ \forall (A, A') \in \theta.\ (t \cdot t')[A] = (t \cdot t')[A']\}.$$

3. *UJoin*($\mathcal{R}$) (*disjunctive equijoins*) – queries of the form $(\mathcal{R}, \mathcal{R}, \Theta)$, where $\mathcal{R} \subseteq \mathscr{S}$, $\mathcal{R} \neq \varnothing$, and $\Theta \subseteq 2^\Omega$. We always present such queries as $(\bowtie_\Theta \mathcal{R})$. Given an instance I of $\mathscr{S}$, we have:

$$(\bowtie_\Theta \mathcal{R})(I) = \{t \in D(\mathcal{R}, I) \mid \exists \theta \in \Theta.\ \forall (A, A') \in \theta.\ t[A] = t[A']\}.$$

4. *UJoin*$^{\ltimes}(\mathcal{R}, \mathcal{R}_o)$ (*disjunctive semijoins*) – queries of the form $(\mathcal{R}, \mathcal{R}_o, \Theta)$, where $\mathcal{R} \subseteq \mathscr{S}$, $\mathcal{R}_o \subset \mathcal{R}$, $\mathcal{R}_o \neq \varnothing$, and $\Theta \subseteq 2^\Omega$. We always present such queries as $(\mathcal{R}_o \ltimes_\Theta (\mathcal{R} \backslash \mathcal{R}_o))$. Given an instance I of $\mathscr{S}$, we have:

$$(\mathcal{R}_o \ltimes_\Theta (\mathcal{R} \backslash \mathcal{R}_o))(I) = \{t \in D(\mathcal{R}_o, I) \mid \exists t' \in D(\mathcal{R} \backslash \mathcal{R}_o, I).\ \exists \theta \in \Theta.\ \forall (A, A') \in \theta.\ (t \cdot t')[A] = (t \cdot t')[A']\}.$$

Notice that the four aforementioned classes of join queries correspond to four classes of relational algebra expressions [AHV95]:

1. equijoins: $(\bowtie_\theta \mathcal{R}) = \bowtie_{\bigwedge_{(A,A') \in \theta, A \in attrs(R), A' \in attrs(R')} R[A] = R'[A']} (\mathcal{R})$,

2. semijoins: $(\mathcal{R}_o \ltimes_\theta (\mathcal{R} \backslash \mathcal{R}_o)) = \Pi_{attrs(\mathcal{R}_o)}(\bowtie_\theta \mathcal{R})$,

3. disjunctive equijoins: $(\bowtie_\Theta \mathcal{R}) = \bigcup_{\theta \in \Theta} (\bowtie_\theta \mathcal{R})$,

4. disjunctive semijoins: $(\mathcal{R}_o \ltimes_\Theta (\mathcal{R} \backslash \mathcal{R}_o)) = \Pi_{attrs(\mathcal{R}_o)}(\bowtie_\Theta \mathcal{R})$.

When $\mathcal{R}$ and $\mathcal{R}_o$ differ, we say that the join result is *projected* on the relations of $\mathcal{R}_o$. Moreover, when the two signatures are known from the context, we often identify a join query by its (disjunctive) join predicate.

Example 4.2.1 (continued). We illustrate the four classes of join queries. Take the signatures $\mathcal{R} = \{R_1, R_2\}$, $\mathcal{R}_1 = \{R_1\}$, $\mathcal{R}_2 = \{R_2\}$, and the following join predicates:

$$\theta_1 = \{(A_1, B_1), (A_2, B_3)\},$$
$$\theta_2 = \{(A_2, B_2)\},$$
$$\theta_3 = \{(A_2, B_1), (A_2, B_2), (A_2, B_3)\}.$$

The answers over I to the queries defined by the aforementioned signatures and join predicates are:

$$(\bowtie_{\theta_1} \mathcal{R})(I) = \{t_2 \cdot t'_2, t_4 \cdot t'_1\}, \qquad (\mathcal{R}_1 \ltimes_{\theta_1} \mathcal{R}_2)(I) = \{t_2, t_4\},$$
$$(\bowtie_{\theta_2} \mathcal{R})(I) = \{t_1 \cdot t'_1, t_1 \cdot t'_2, t_4 \cdot t'_3\}, \qquad (\mathcal{R}_1 \ltimes_{\theta_2} \mathcal{R}_2)(I) = \{t_1, t_4\},$$
$$(\bowtie_{\theta_3} \mathcal{R})(I) = \varnothing, \qquad (\mathcal{R}_1 \ltimes_{\theta_3} \mathcal{R}_2)(I) = \varnothing. \qquad \square$$

Next, consider the disjunctive join predicates

$$\Theta_1 = \{\{(A_1, B_1)\}, \{(A_2, B_3)\}\},$$
$$\Theta_2 = \{\{(A_2, B_2)\}\},$$
$$\Theta_3 = \{\{(A_2, B_1), (A_2, B_2)\}, \{(A_2, B_3)\}\}$$

The answers over I to the queries defined by the aforementioned signatures and disjunctive join predicates are:

$$(\bowtie_{\Theta_1} \mathcal{R})(I) = \{t_1 \cdot t'_2, t_2 \cdot t'_2, t_3 \cdot t'_2, t_3 \cdot t'_3, t_4 \cdot t'_1, t_4 \cdot t'_3\}, \quad (\mathcal{R}_1 \ltimes_{\Theta_1} \mathcal{R}_2)(I) = I(R_1),$$
$$(\bowtie_{\Theta_2} \mathcal{R})(I) = \{t_1 \cdot t'_1, t_1 \cdot t'_2, t_4 \cdot t'_3\}, \quad (\mathcal{R}_1 \ltimes_{\Theta_2} \mathcal{R}_2)(I) = \{t_1, t_4\},$$
$$(\bowtie_{\Theta_3} \mathcal{R})(I) = \{t_1 \cdot t'_1, t_2 \cdot t'_2, t_3 \cdot t'_2, t_4 \cdot t'_1, t_4 \cdot t'_3\}, \quad (\mathcal{R}_1 \ltimes_{\Theta_3} \mathcal{R}_2)(I) = I(R_1).$$
$\square$

In the next sections, we study the problem of learning join queries from examples given by the user.

4.3 Learning from a set of examples

In this section, we study the problem of learning join queries from a given set of examples. First, we define a *learning framework* (Section 4.3.1). Then, we investigate the *consistency checking* problem (Section 4.3.2) that is fundamental for establishing our *learnability results* (Section 4.3.3).

4.3.1 Learning framework

Assume a schema $\mathscr{S}$. An *example* is a pair (t, α), where t is a tuple and $\alpha \in \{+, -\}$. We say that an example of the form $(t, +)$ is a *positive example* while an example of the form $(t, -)$ is a *negative example*. Moreover, we say that an example (t, α) is compatible with a signature $\mathcal{R} \subseteq \mathscr{S}$ if t is compatible with $\mathcal{R}$. Recall that by $D(\mathcal{R}, I)$ we denote the Cartesian product of the instance I of a signature $\mathcal{R} \subseteq \mathscr{S}$.

A *learning setting* is a tuple $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$ where

- $\mathcal{R} \subseteq \mathscr{S}$ is the non-empty input signature of K i.e., $\mathit{inSig}(K) = \mathcal{R}$,
- $\mathcal{R}_o \subseteq \mathcal{R}$ such that $\mathcal{R}_o \neq \varnothing$ is the non-empty output signature of K i.e., $\mathit{outSig}(K) = \mathcal{R}_o$,
- $\mathcal{Q}$ is a class of queries such that $\mathit{inSig}(Q) = \mathcal{R}$ and $\mathit{outSig}(Q) = \mathcal{R}_o$.

For instance, take two non-empty signatures $\mathcal{R} \subseteq \mathscr{S}$ and $\mathcal{R}_o \subset \mathcal{R}$. Then, the following are examples of learning settings.

- $(\mathcal{R}, \mathcal{R}, \mathit{Join}(\mathcal{R}))$ for learning equijoins over $\mathcal{R}$,
- $(\mathcal{R}, \mathcal{R}_o, \mathit{Join}^{\ltimes}(\mathcal{R}, \mathcal{R}_o))$ for learning semijoins over $\mathcal{R}$ and $\mathcal{R}_o$,
- $(\mathcal{R}, \mathcal{R}, \mathit{UJoin}(\mathcal{R}))$ for learning disjunctive equijoins over $\mathcal{R}$,
- $(\mathcal{R}, \mathcal{R}_o, \mathit{UJoin}^{\ltimes}(\mathcal{R}, \mathcal{R}_o))$ for learning disjunctive semijoins over $\mathcal{R}$ and $\mathcal{R}_o$.

Next, we define four classes of learning settings i.e., one for each class of join queries.

- The class of settings for learning equijoins:

$$\mathsf{Join} = \{(\mathcal{R}, \mathcal{R}, \mathit{Join}(\mathcal{R})) \mid \mathcal{R} \subseteq \mathscr{S} \text{ such that } \mathcal{R} \neq \varnothing \text{ for every schema } \mathscr{S}\}.$$

- The class of settings for learning semijoins:

$$\mathsf{Join}^{\ltimes} = \{(\mathcal{R}, \mathcal{R}_o, \mathit{Join}^{\ltimes}(\mathcal{R}, \mathcal{R}_o)) \mid \mathcal{R} \subseteq \mathscr{S} \text{ and } \mathcal{R}_o \subset \mathcal{R} \text{ such that } \mathcal{R}_o \neq \varnothing \text{ for every schema } \mathscr{S}\}.$$

- The class of settings for learning disjunctive equijoins:

$$\mathsf{UJoin} = \{(\mathcal{R}, \mathcal{R}, \mathit{UJoin}(\mathcal{R})) \mid \mathcal{R} \subseteq \mathscr{S} \text{ such that } \mathcal{R} \neq \varnothing \text{ for every schema } \mathscr{S}\}.$$

- The class of settings for learning disjunctive semijoins:

$$\mathsf{UJoin}^{\ltimes} = \{(\mathcal{R}, \mathcal{R}_o, \mathit{UJoin}^{\ltimes}(\mathcal{R}, \mathcal{R}_o)) \mid \mathcal{R} \subseteq \mathscr{S} \text{ and } \mathcal{R}_o \subset \mathcal{R} \text{ such that } \mathcal{R}_o \neq \varnothing \text{ for every schema } \mathscr{S}\}.$$

Additionally, given a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$, we define:

- The set $\mathcal{I}^K = \bigcup_{R \in \mathcal{R}} \mathcal{U}^{attrs(R)}$ of all tuples compatible with relations in $\mathcal{R}$.
- The set $\mathcal{E}^K = \mathcal{U}^{attrs(\mathcal{R}_o)} \times \{+, -\}$ of all examples compatible with $\mathcal{R}_o$.
- The function $\mathcal{L}^K$ that maps every query q from $\mathcal{Q}$ and instance $I \subseteq \mathcal{I}^K$ such that $\mathit{inSig}(q) = \mathit{sig}(D(\mathcal{R}, I))$ to the corresponding set of examples $S \subseteq \mathcal{E}^K$ such that $\mathit{outSig}(q) = \mathit{sig}(D(\mathcal{R}_o, I))$ i.e.,

$$\mathcal{L}^K(q, I) = q(I) \times \{+\} \cup (D(\mathcal{R}_o, I) \backslash q(I)) \times \{-\}.$$

Given a learning setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$ and an instance $I \subseteq \mathcal{I}^K$, a *sample w.r.t. I and K* is a subset $S \subseteq \mathcal{E}^K$ of examples (t, α) such that $\mathit{sig}(t) = \mathit{outSig}(\mathcal{Q})$ and $\alpha \in \{+, -\}$. When it does not lead to confusion, we simply write that S is a *sample* or a *sample over I*. For a sample S, we denote the set of positive examples $\{t \in D(\mathcal{R}_o, I) \mid (t, +) \in S\}$ by S_+ and the set of negative examples $\{t \in D(\mathcal{R}_o, I) \mid (t, -) \in S\}$ by S_-.

Moreover, given a learning setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$, an instance $I \subseteq \mathcal{I}^K$, a sample $S \subseteq \mathcal{E}^K$ over I, and a query $q \in \mathcal{Q}$, we say that q is *consistent with S* if it selects all positive examples and none of the negative ones i.e., $S_+ \subseteq q(I)$ and $S_- \cap q(I) = \varnothing$. Naturally, the goal of learning should be the construction of a consistent join query.

When the class of queries is clear from the context, we may write that a (disjunctive) join predicate is consistent with a sample rather than the query that it defines. For instance, we may simply write that the join predicate θ is consistent with a sample S instead of writing that the query $(\bowtie_\theta \mathcal{R})$ is consistent with S.

Next, we propose a definition of learnability based on the standard framework of *language identification in the limit with polynomial time and data* [Gol78] adapted to learning join queries. A *learning algorithm* is an algorithm that takes an instance and a sample, and returns a query in $\mathcal{Q}$ or a special value *null*.

Definition 4.3.1 *A class of queries $\mathcal{Q}$ is* learnable in polynomial time and data *in its corresponding learning setting $K = (inSig(\mathcal{Q}), outSig(\mathcal{Q}), \mathcal{Q})$ if there exists a* polynomial learning algorithm *learner satisfying the following two conditions:*

1. **Soundness.** *For every instance $I \subseteq \mathcal{I}^K$ and every sample $S \subseteq \mathcal{E}^K$ over I, the algorithm learner(I, S) returns a query $q \in \mathcal{Q}$ that is consistent with S or a special* null *value if no such query exists.*

2. **Completeness.** *For every query $q \in \mathcal{Q}$ there exists an instance $I \subseteq \mathcal{I}^K$ and a sample $CS_q \subseteq \mathcal{E}^K$ over I such that for every sample S that extends CS_q consistently with q i.e., $CS_q \subseteq S \subseteq \mathcal{L}^K(q, I)$, the algorithm learner$(I, S)$ returns a query equivalent to q. Furthermore, the size of CS_q is polynomially bounded by the size of the query.*

The sample CS_q is called the *characteristic sample* for q w.r.t. *learner* and K. For a learning algorithm there may exist many such samples. The definition requires that a characteristic sample exists. The soundness condition is a natural requirement while the completeness condition guarantees that the learning algorithm constructs the goal query from a sufficiently rich (but still polynomial) set of examples.

4.3.2 Consistency checking

As we have already pointed out in Definition 4.3.1, we aim at a polynomial learning algorithm that returns a query that selects all positive examples and none of the negative ones. To this purpose, we first investigate the consistency checking problem i.e., deciding whether such a query exists. This also permits to check whether the user who has provided the examples is honest, has not made any error, and therefore, has labeled the tuples consistently with some goal join query that she has in mind.

More formally, the *consistency checking for a class of learning settings* K is the following decision problem: given a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$ from K, an instance $I \subseteq \mathcal{I}^K$, and a sample $S \subseteq \mathcal{E}^K$ over I, decide whether there exists a query $q \in \mathcal{Q}$ that is consistent with the sample. Consistency checking is parametrized by the learning setting and the corresponding decision problem is:

$$\begin{aligned}\mathrm{CONS}_{\mathsf{K}} = \{(K, I, S) \mid\ & K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q}) \in \mathsf{K}, I \subseteq \mathcal{I}^K, S \subseteq \mathcal{E}^K. \\ & \exists q \in \mathcal{Q}.\ S_+ \subseteq q(I) \wedge S_- \cap q(I) = \varnothing\}.\end{aligned}$$

We summarize in Table 4.1 the complexity of consistency checking for the considered classes of join queries and we present the proofs of these results in the rest of the section.

	Equijoins	*Semijoins*
Without disjunction	PTIME (Theorem 4.3.3)	NP-complete (Theorem 4.3.4)
With disjunction	PTIME (Theorem 4.3.7)	PTIME (Theorem 4.3.7)

Table 4.1: Summary of complexity results for consistency checking.

Consistency checking for equijoins. First, we show that in the case of equijoins the consistency checking has a simple solution that employs an elementary tool that we introduce next. Given a tuple $t \in D(\mathcal{R}, I)$, we define the *most specific join predicate selecting* t as follows:

$$T(t) = \{(A, A') \mid t[A] = t[A'] \wedge A \in attrs(R) \wedge A' \in attrs(R') \wedge R \neq R'\}.$$

Additionally, we extend T to sets of tuples $T(X) = \bigcap_{t \in X} T(t)$. Our interest in T follows from the observation that for a given set of tuples X, if θ is a join predicate selecting X, then $\theta \subseteq T(X)$.

Example 4.2.1 (continued). In Figure 4.2 we present the Cartesian product $R_1 \times R_2$, the value of T for each tuple from it, and the sample S_0 such that $S_{0,+} = \{t_2 \cdot t'_2, t_4 \cdot t'_1\}$ and $S_{0,-} = \{t_3 \cdot t'_2\}$. The sample is consistent and the most specific consistent join predicate is $\theta_0 = \{(A_1, B_1), (A_2, B_3)\}$. Another consistent join predicate (but not the most specific) is $\theta'_0 = \{(A_1, B_1)\}$. On the other hand, the sample S'_0 such that $S'_{0,+} = \{t_1 \cdot t'_2, t_1 \cdot t'_3\}$ and $S'_{0,-} = \{t_3 \cdot t'_1\}$ is not consistent. □

We next show that a sample S is consistent with a join predicate iff $T(S_+)$ selects no negative example.

Lemma 4.3.2 *Given a setting* $K = (\mathcal{R}, \mathcal{R}, Join(\mathcal{R}))$ *in* Join, *an instance* $I \subseteq \mathcal{I}^K$, *and a sample* $S \subseteq \mathcal{E}^K$ *over* I, *it holds that* $(K, I, S) \in \text{CONS}_{\text{Join}}$ *iff* $S_- \cap (\bowtie_{T(S_+)} \mathcal{R})(I) = \varnothing$.

Proof For the *if* part, since $T(S_+)$ selects all positive examples (by definition) and selects no negative tuple (by hypothesis) we infer that $T(S_+)$ is a predicate consistent with the sample.

For the *only if* part, assume that there exists a predicate θ selecting all positive examples and none of the negative ones. Since $T(S_+)$ is the most specific join predicate selecting all positive examples, $\theta \subseteq T(S_+)$, and since θ selects no negative example, neither does $T(S_+)$. Hence, $T(S_+)$ is also a join predicate consistent with the set of examples. □

	A_1	A_2	B_1	B_2	B_3	T
$t_1 \cdot t'_1$	0	1	1	1	0	$\{(A_1, B_3), (A_2, B_1), (A_2, B_2)\}$
$t_1 \cdot t'_2$	0	1	0	1	2	$\{(A_1, B_1), (A_2, B_2)\}$
$t_1 \cdot t'_3$	0	1	2	0	0	$\{(A_1, B_2), (A_1, B_3)\}$
$t_2 \cdot t'_1$	0	2	1	1	0	$\{(A_1, B_3)\}$
$+\ t_2 \cdot t'_2$	0	2	0	1	2	$\{(A_1, B_1), (A_2, B_3)\}$
$t_2 \cdot t'_3$	0	2	2	0	0	$\{(A_1, B_2), (A_1, B_3), (A_2, B_1)\}$
$t_3 \cdot t'_1$	2	2	1	1	0	$\varnothing$
$-\ t_3 \cdot t'_2$	2	2	0	1	2	$\{(A_1, B_3), (A_2, B_3)\}$
$t_3 \cdot t'_3$	2	2	2	0	0	$\{(A_1, B_1), (A_2, B_1)\}$
$+\ t_4 \cdot t'_1$	1	0	1	1	0	$\{(A_1, B_1), (A_1, B_2), (A_2, B_3)\}$
$t_4 \cdot t'_2$	1	0	0	1	2	$\{(A_1, B_2), (A_2, B_1)\}$
$t_4 \cdot t'_3$	1	0	2	0	0	$\{(A_2, B_2), (A_2, B_3)\}$

Figure 4.2: The Cartesian product $R_1 \times R_2$, the value of T for each of its tuples, and sample S_0.

Next, we use the above Lemma to show the tractability of the consistency checking for equijoins.

Theorem 4.3.3 $\text{CONS}_{\mathsf{Join}}$ *is in PTIME.*

Proof Lemma 4.3.2 gives us a necessary and sufficient condition for solving this problem i.e., testing whether the join predicate $T(S_+)$ selects any negative example. First, we point out that for a tuple $t \in D(\mathcal{R}, I)$, a simple algorithm computes $T(t)$ and implicitly $T(S_+)$ in polynomial time. Then, we have to check whether these queries select any negative example, which can be also easily done in polynomial time. □

Consistency checking for semijoins. Next, we show that for semijoins the fundamental decision problem of consistency checking is unfortunately intractable, even when we consider two relations only, as we state below.

Theorem 4.3.4 $\text{CONS}_{\mathsf{Join}^{\ltimes}}$ *is NP-complete. The result holds even when the schema consists of two relations only.*

Proof To prove the membership of the problem to NP, we point out that a Turing machine guesses a join predicate θ, which has polynomial size in the size of the input. Then, we can easily check in polynomial time whether θ selects all positive examples and none of the negative ones.

Next, we prove the NP-hardness of $\text{CONS}_{\mathsf{Join}^{\ltimes}}$ by reduction from 3SAT, known as being NP-complete. Given a formula $\varphi = c_1 \wedge \ldots \wedge c_k$ in 3CNF over the set of variables $\{x_1, \ldots, x_n\}$, we construct:

- A schema $\mathscr{S}_\varphi = \{R_\varphi, P_\varphi\}$, where $attrs(R_\varphi) = \{id_R, A_1, \ldots, A_n\}$ and $attrs(P_\varphi) = \{id_P,\ B_1^t, B_1^f, \ldots, B_n^t, B_n^f\}$, the input signature $\mathcal{R}_\varphi = \mathscr{S}_\varphi$, the output signature $\mathcal{R}_{o\varphi} = \{R_\varphi\}$.

- The learning setting $K\varphi = (\mathcal{R}_\varphi, \mathcal{R}_{o\varphi}, Join^{\ltimes}(\mathcal{R}_\varphi, \mathcal{R}_{o\varphi}))$.

- The instance $I_\varphi(R_\varphi)$ of the relation R_φ that contains:

  - For $1 \leqslant i \leqslant k$, a tuple $t_{R,i}$ with $t_{R,i}[id_R] = c_i^+$ and $t_{R,i}[A_j] = j$ (for $1 \leqslant j \leqslant n$),
  - A tuple $t'_{R,0}$ with $t'_{R,0}[id_R] = X$ and $t'_{R,0}[A_j] = j$ (for $1 \leqslant j \leqslant n$),
  - For $1 \leqslant i \leqslant n$, a tuple $t'_{R,i}$ with $t_{Ri}[id_R] = x_i^-$ and $t'_{R,i}[A_j] = j$ (for $1 \leqslant j \leqslant n$).

- The instance $I_\varphi(P_\varphi)$ of the relation P_φ that contains:

  - For $1 \leqslant i \leqslant k$, let $x_{k_1}, x_{k_2}, x_{k_3}$ the variables used by c_k, with $k_1, k_2, k_3 \in \{1, \ldots, n\}$. Then, we have a tuple for each of them, let it $t_{P,il}$, with $l \in \{1, 2, 3\}$ such that $t_{P,il}[id_P] = c_i^+$, and for $1 \leqslant j \leqslant n$:

$$\begin{cases} t_{P,il}[B_j^t] = t_{P,il}[B_j^f] = j \text{ if } j \neq k_l, \\ t_{P,il}[B_j^t] = j,\ t_{P,il}[B_j^f] = \perp \text{ if } j = k_l \text{ and } x_{k_l} \text{ is a positive literal}, \\ t_{P,il}[B_j^t] = \perp,\ t_{P,il}[B_j^f] = j \text{ otherwise.} \end{cases}$$

  - A tuple $t'_{P,0}$ such that $t'_{P,0}[id_P] = Y$ and $B_i^t = B_i^f = i$ (for $1 \leqslant i \leqslant n$).
  - For $1 \leqslant i \leqslant n$, a tuple $t'_{P,i}$ such that $t'_{P,i}[id_P] = x_i^-$ and $t'_{P,i}[B_j^t] = t'_{P,i}[B_j^f] = j$ if $i \neq j$ or $t'_{P,i}.[B_j^t] = t'_{P,i}[B_j^f] = \perp$ otherwise (for $1 \leqslant j \leqslant n$).

- The sample $S_\varphi \subseteq R_\varphi \times \{+, -\}$ such that $S_{\varphi,+} = \{t_{R,1}, \ldots, t_{R,k}\}$ and $S_{\varphi,-} = \{t'_{R,0}, t'_{R,1}, \ldots t'_{R,n}\}$.

For example, for $\varphi_0 = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$, we construct $I_{\varphi_0}(R_{\varphi_0})$ and $I_{\varphi_0}(P_{\varphi_0})$, respectively as shown below:

$I_{\varphi_0}(R_{\varphi_0}) =$

	id_R	A_1	A_2	A_3	A_4
$t_{R,1}$	c_1^+	1	2	3	4
$t_{R,2}$	c_2^+	1	2	3	4
$t'_{R,0}$	X	1	2	3	4
$t'_{R,1}$	x_1^-	1	2	3	4
$t'_{R,2}$	x_2^-	1	2	3	4
$t'_{R,3}$	x_3^-	1	2	3	4
$t'_{R,4}$	x_4^-	1	2	3	4

$I_{\varphi_0}(P_{\varphi_0}) =$

	id_P	B_1^t	B_1^f	B_2^t	B_2^f	B_3^t	B_3^f	B_4^t	B_4^f
$t_{P,11}$	c_1^+	1	$\perp$	2	2	3	3	4	4
$t_{P,12}$	c_1^+	1	1	$\perp$	2	3	3	4	4
$t_{P,13}$	c_1^+	1	1	2	2	3	$\perp$	4	4
$t_{P,21}$	c_2^+	$\perp$	1	2	2	3	3	4	4
$t_{P,22}$	c_2^+	1	1	2	2	$\perp$	3	4	4
$t_{P,23}$	c_2^+	1	1	2	2	3	3	4	$\perp$
$t'_{P,0}$	Y	1	1	2	2	3	3	4	4
$t'_{P,1}$	x_1^-	$\perp$	$\perp$	2	2	3	3	4	4
$t'_{P,2}$	x_2^-	1	1	$\perp$	$\perp$	3	3	4	4
$t'_{P,3}$	x_3^-	1	1	2	2	$\perp$	$\perp$	4	4
$t'_{P,4}$	x_4^-	1	1	2	2	3	3	$\perp$	$\perp$

and the sample S_{φ_0} such that $S_{\varphi_0,+} = \{t_{R,1}, t_{R,2}\}$ and $S_{\varphi_0,-} = \{t'_{R,0}, t'_{R,1}, t'_{R,2}, t'_{R,3}, t'_{R,4}\}$.

We claim that φ is satisfiable iff $(K_\varphi, I_\varphi, S_\varphi) \in \mathrm{CONS}_{\mathsf{Join}^\ltimes}$.

For the *if* part, let θ_0 the join predicate that selects all positive examples and none of the negatives. We observe from the instances of R_φ and P_φ that $\theta_0 \subseteq \{(id_R, id_P)\} \cup \{(A_i, B_i^t), (A_i, B_i^f) \mid 1 \leqslant i \leqslant n\}$. Because $S_{\varphi,-}$ is not empty, we infer that θ_0 is not empty. Moreover, we show that θ_0 contains (id_R, id_P) and at least one of (A_i, B_i^t) and (A_i, B_i^f) (for $1 \leqslant i \leqslant n$) by eliminating the other cases:

1. If we assume that $(id_R, id_P) \notin \theta_0$, we infer that the negative example $t'_{R,0}$ belongs to $(\mathcal{R}_{o\varphi} \ltimes_{\theta_0} (\mathcal{R}_{o\varphi} \backslash \mathcal{R}_\varphi)(I)$, which contradicts the fact that θ_0 is consistent with S_φ.

2. If we assume that there exists an $1 \leqslant i \leqslant n$ such that neither (A_i, B_i^t) nor (A_i, B_i^f) belongs to θ_0, we infer that the negative example $t'_{R,i}$ belongs to $(\mathcal{R}_{o\varphi} \ltimes_{\theta_0} (\mathcal{R}_{o\varphi} \backslash \mathcal{R}_\varphi)(I)$, which contradicts the fact that θ_0 is consistent with S_φ.

Thus, we know that (id_R, id_P) belongs to θ_0 and at least one of (A_i, B_i^t) and (A_i, B_i^f) also belongs to θ_0 (for $1 \leqslant i \leqslant n$). From the construction of the instance we infer that there exists a join between A_i and B_i^v (with $v \in \{t, f\}$) if the valuation encoded in v for x_i does not make false the clause whose number is encoded in id_P (for $1 \leqslant i \leqslant n$). Moreover, θ_0 is consistent with S_φ implies that for each tuple $t_{R,i}$ from R (with $1 \leqslant i \leqslant k$), there exists a tuple $t_{P,il}$ in P (with $1 \leqslant l \leqslant 3$) such that $t_{R,i}[id_R] = t_{P,il}[id_P]$ and for $1 \leqslant j \leqslant n$ there exists $v \in \{t, f\}$ such that $t_{R,i}[A_j] = t_{P,il}[B_j^v]$. Thus, the valuation encoded in the B_j^v's from θ_0 (for $1 \leqslant j \leqslant n$) satisfies φ.

For the *only if* part, take the valuation $V : \{x_1, \ldots, x_n\} \to \{true, false\}$ that makes φ true. We construct the join predicate θ_0 that contains (id_R, id_P) and $(A_i, B_i^{v_i})$ where $v_i \in \{t, f\}$ corresponds to the valuation $V(x_i)$. From the construction of $I_\varphi(P_\varphi)$, we infer that for $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant k$, there exists $1 \leqslant l \leqslant 3$ such that $t_{P,jl}[id_P] = c_j^+$ and $t_{P,jl}[B_i^{v_i}] = i$. We infer that θ_0 is consistent with S_φ, and therefore, $(K_\varphi, I_\varphi, S_\varphi) \in \mathrm{CONS}_{\mathsf{Join}^\ltimes}$.

Clearly, the described reduction works in polynomial time. □

Consistency checking for disjunctive equijoins and semijoins. We show that consistency checking is tractable when adding the disjunction, both for equijoins and semijoins. We start with the disjunctive semijoins and then we point out that the disjunctive equijoins enjoy the same computational properties since they are in fact a particular case. Let us first present a necessary and sufficient condition for a sample to be consistent.

Lemma 4.3.5 *Given a setting $K = (\mathcal{R}, \mathcal{R}_o, UJoin^\ltimes(\mathcal{R}, \mathcal{R}_o))$ in $\mathsf{UJoin}^\ltimes$, an instance $I \subseteq \mathcal{I}^K$, and a sample $S \subseteq \mathcal{E}^K$ over I, it holds that*

$$(K, I, S) \in \mathrm{CONS}_{\mathsf{UJoin}^\ltimes} \text{ iff } \forall t \in S_+.\ \exists t' \in D(\mathcal{R}\backslash\mathcal{R}_o, I).\ \forall t'' \in S_-.\ t'' \notin (\mathcal{R}_o \ltimes_{T(t\cdot t')} (\mathcal{R}\backslash\mathcal{R}_o))(\mathcal{I}).$$

Proof For the *if* part, we construct a disjunctive join predicate Θ as follows. We start with $\Theta = \varnothing$ and for each $t \in S_+$ we take a $t' \in D(\mathcal{R}\backslash\mathcal{R}_o, I)$ such that $S_- \cap (\mathcal{R}_o \ltimes_{T(t\cdot t')} (\mathcal{R}\backslash\mathcal{R}_o))(\mathcal{I}) = \varnothing$ (we know by hypothesis that such a t' does exist) and we add $T(t \cdot t')$ to Θ. Notice that the constructed Θ selects all positive examples and none of the negative ones, hence we infer that Θ is a disjunctive predicate consistent with the sample.

For the *only if* part, assume that there exists a disjunctive join predicate Θ selecting all positive examples and none of the negative ones. Since Θ selects all positive examples, we infer that for every $t \in S_+$ there exists a predicate $\theta \in \Theta$ such that $t \in (\mathcal{R}_o \ltimes_\theta (\mathcal{R}\backslash\mathcal{R}_o))(\mathcal{I})$ and $S_- \cap (\mathcal{R}_o \ltimes_\theta (\mathcal{R}\backslash\mathcal{R}_o))(\mathcal{I}) = \varnothing$. This implies that for every $t \in S_+$ there exists a predicate $\theta \in \Theta$ and a tuple

$t' \in D(\mathcal{R}\backslash\mathcal{R}_o, I)$ such that $t{\cdot}t' \in (\bowtie_\theta \mathcal{R})(I)$ and $S_- \cap (\mathcal{R}_o \ltimes_\theta (\mathcal{R}\backslash\mathcal{R}_o))(\mathcal{I}) = \varnothing$. Since for all such tuples t and t' the above θ is included in the most specific predicate $T(t{\cdot}t')$ and θ selects no negative, we infer that neither does $T(t{\cdot}t')$, which concludes the proof. □

In the particular case where the input and output signatures coincide, all t' from Lemma 4.3.5 are in fact empty tuples and Lemma 4.3.5 reduces to the following result.

Corollary 4.3.6 *Given a setting $K = (\mathcal{R}, \mathcal{R}, UJoin(\mathcal{R}))$ in* UJoin*, an instance $I \subseteq \mathcal{I}^K$, and a sample $S \subseteq \mathcal{E}^K$ over I, it holds that*

$$(K, I, S) \in \text{CONS}_{\mathsf{UJoin}} \text{ iff } S_- \cap (\bowtie_{\bigcup_{t\in S_+}\{T(t)\}} \mathcal{R})(\mathcal{I}) = \varnothing.$$

Next, we show that consistency checking can be solved in polynomial time for disjunctive equijoins and semijoins.

Theorem 4.3.7 $\text{CONS}_{\mathsf{UJoin}}$ *and* $\text{CONS}_{\mathsf{UJoin}^\ltimes}$ *are in PTIME.*

Proof Lemma 4.3.5 gives us a necessary and sufficient condition for solving the consistency checking for disjunctive semijoins. First, we point out that for two tuples $t \in S_+$ and $t' \in D(\mathcal{R}\backslash\mathcal{R}_o, I)$, a simple algorithm computes $T(t \cdot t')$ in polynomial time and this has to be done for $|S_+| \times |D(\mathcal{R}\backslash\mathcal{R}_o, I)|$ tuples. Then, we have to check whether these queries select any negative example, which can be also easily done in polynomial time. The aforementioned simple procedure can be also applied for disjunctive equijoins i.e., in the particular case where all t' are empty tuples. □

4.3.3 Learnability results

In this section, we use the developments from the previous section to characterize the learnability of different classes of join queries. We summarize in Table 4.2 the learnability results and we present the proofs in the rest of the section. First, we show that the equijoins are learnable.

	Equijoins	*Semijoins*
Without disjunction	Yes (Theorem 4.3.8)	No (Theorem 4.3.9)
With disjunction	Yes (Theorem 4.3.10)	Yes (Theorem 4.3.10)

Table 4.2: Summary of learnability results.

Theorem 4.3.8 *The equijoins are learnable in polynomial time and data i.e., in settings from* Join*.*

Proof Take a setting $K = (\mathcal{R}, \mathcal{R}, \mathit{Join}(\mathcal{R}))$ in Join. Then, take an instance $I \subseteq \mathcal{I}^K$ and a sample $S \subseteq \mathcal{E}^K$ over I. A simple polynomial algorithm outputs $T(S_+)$ if the sample is consistent or *null* otherwise. Recall that this can be decided in polynomial time due to Theorem 4.3.3.

To show the completeness, we point out the construction, for every setting $K = (\mathcal{R}, \mathcal{R}, \mathit{Join}(\mathcal{R}))$ in Join, for every query in $\mathit{Join}(\mathcal{R})$, of an instance and of a polynomial characteristic sample $CS \subseteq \mathcal{E}^K$. Take a query $(\bowtie_\theta \mathcal{R})$ in $\mathit{Join}(\mathcal{R})$. Then, we need an instance $I \subseteq \mathcal{I}^K$ having a tuple t such that $T(t) = \theta$ and CS_+ must contain this tuple. □

Next, we show that the intractability of the consistency checking for semijoins implies that the semijoins are not learnable.

Theorem 4.3.9 *The semijoins are not learnable in polynomial time and data i.e., in settings from* $\mathsf{Join}^{\ltimes}$. *The result holds even when the schema consists of two relations only.*

Proof According to Definition 4.3.1, a learning algorithm should return *null* in polynomial time if a query consistent with the sample does not exist. Since checking the consistency of a sample is NP-complete (cf. Theorem 4.3.4), such an algorithm does not exist, hence the semijoins are not learnable in polynomial time and data. □

Finally, we show that both disjunctive equijoins and disjunctive semijoins are learnable.

Theorem 4.3.10 *The disjunctive equijoins and disjunctive semijoins are learnable in polynomial time and data i.e., in settings from* UJoin *and* $\mathsf{UJoin}^{\ltimes}$, *respectively.*

Proof We show the soundness and completeness for the class of settings $\mathsf{UJoin}^{\ltimes}$ and we point out that the same algorithm is applicable for the class of settings UJoin that is a particular case where the input and output signatures coincide. Take a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathit{UJoin}^{\ltimes}(\mathcal{R}, \mathcal{R}_o))$ in $\mathsf{UJoin}^{\ltimes}$, an instance $I \subseteq \mathcal{I}^K$, and a sample $S \subseteq \mathcal{E}^K$ over I. If the sample is not consistent (decidable in polynomial time due to Theorem 4.3.7), the learning algorithm returns *null*. If the sample is consistent, the learning algorithm returns a consistent disjunctive join predicate constructed as follows: (i) start with an empty Θ, (ii) for every $t \in S_+$, for every $t' \in D(\mathcal{R} \backslash \mathcal{R}_o, I)$, if $T(t \cdot t')$ selects no negative example, add $T(t \cdot t')$ to Θ (we know that such tuples t' exist for every t due to Lemma 4.3.5), (iii) eliminate the redundant predicates from Θ and return $\Theta' = \{\theta \in \Theta \mid \nexists \theta' \in \Theta.\ \theta' \subset \theta\}$.

For example, if after (ii) we have $\Theta_0 = \{\{(A,B)\}, \{(A,B),(C,D)\}, \{(E,F)\}\}$, the predicate $\{(A,B),(C,D)\}$ is redundant because every tuple selected by it is also selected by $\{(A,B)\}$. Thus, by removing the redundant predicate we obtain $\Theta'_0 = \{\{(A,B)\},\{(E,F)\}\}$.

To show the completeness, we point out the construction, for every setting $K = (\mathcal{R}, \mathcal{R}_o, \mathit{UJoin}^{\ltimes}(\mathcal{R},\mathcal{R}_o))$ in $\mathsf{UJoin}^{\ltimes}$, for every query in $\mathit{UJoin}^{\ltimes}(\mathcal{R},\mathcal{R}_o)$, of an instance and of a polynomial characteristic sample. Take a disjunctive join predicate Θ that defines a query $(\mathcal{R}_o \ltimes_\Theta (\mathcal{R}\backslash\mathcal{R}_o))$. First, we eliminate the redundant join predicates in Θ and construct an equivalent disjunctive join predicate $\Theta' = \{\theta \in \Theta \mid \nexists \theta' \in \Theta.\ \theta' \subset \theta\}$. Then, we need an instance $I \subseteq \mathcal{I}^K$ and a characteristic sample $CS \subseteq \mathcal{E}^K$ over I as follows:

- For every $\theta \in \Theta'$, there exists a tuple $t \in D(\mathcal{R}_o, I)$ and another $t' \in D(\mathcal{R}\backslash\mathcal{R}_o, I)$ such that $T(t \cdot t') = \theta$ and $t \in CS_+$.
- For every $\theta \in \Theta$, for every $\theta' \in \{\theta\backslash\{(A,A')\} \mid (A,A') \in \theta\}$, there exists a tuple $t \in D(\mathcal{R}_o, I)$ and another $t' \in D(\mathcal{R}\backslash\mathcal{R}_o, I)$ such that $T(t \cdot t') = \theta'$ and $t \in CS_-$.

The positive examples ensure that the simple algorithm described at the beginning of the proof retrieves from the instance each predicate from Θ' while the negative examples ensure that these predicates are indeed selected by the algorithm.

We end the proof by pointing out that the size of CS_+ is bounded by $|\Theta|$ while the size of CS_- is bounded by $|\Theta| \times \max_{\theta\in\Theta} |\theta|$, which means that the size of the characteristic sample is polynomial in the size of the goal Θ. □

4.4 Learning from user interactions

In this section, we study the problem of learning join queries from a different point of view, where we start with an empty sample that we enrich during the interactions with the user. First, we define our *interactive scenario* (Section 4.4.1). Then, we say a few words about the *instance-equivalent join queries* that may be output by the learning algorithm (Section 4.4.2). Moreover, we characterize the tuples that are *uninformative* or *informative* w.r.t. the learning process (Section 4.4.3).

4.4.1 Interactive scenario

Let us now consider the following *interactive scenario* of join query inference. Take a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$, an instance $I \subseteq \mathcal{I}^K$, and assume that the

user has in mind a query that belongs to the class of queries $\mathcal{Q}$. The user is presented with a tuple from the Cartesian product $D(\mathcal{R}_o, I)$ and indicates whether the tuple is selected or not by the join query that she has in mind by labeling the tuple as a positive or negative example. This process is repeated until a sufficient knowledge of the goal join query has been accumulated (i.e., there exists at most one join query consistent with the user's labels).

This scenario is inspired by the well-known framework of *learning with membership queries* proposed by [Ang88]. Especially with large instances, we would not like that the user has to label all the tuples of the integrated table, but only a small subset of them. Our goal is to minimize the number of interactions with the user while still being computationally efficient. In this context, an interesting question is choosing the right strategy of presenting tuples to the user. To answer this question, our approach leads through the analysis of the potential information that labeling a given tuple may contribute from the point of view of the inference process.

To this purpose, we first need to introduce some auxiliary notions. We assume the existence of some goal join query q^γ from $\mathcal{Q}$ and that the user labels the tuples in a manner consistent with q^γ. Furthermore, given an instance $I \subseteq \mathcal{I}^K$, we identify the sample $S^\gamma \subseteq \mathcal{E}^K$ over I corresponding to fully labeling the database instance:

$$S^\gamma_+ = q^\gamma(I) \quad \text{and} \quad S^\gamma_- = D(\mathcal{R}_o, I) \setminus q^\gamma(I).$$

Moreover, given a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$, an instance $I \subseteq \mathcal{I}^K$, and a sample $S \subseteq \mathcal{E}^K$ over I, we identify the set of all queries that are consistent with S over I in K:

$$\mathcal{C}^K(I, S) = \{q \in \mathcal{Q} \mid S_+ \subseteq q(I) \text{ and } S_- \cap q(I) = \varnothing.\}.$$

When K is clear from the context, we write simply $\mathcal{C}(I, S)$ instead of $\mathcal{C}^K(I, S)$. Initially, $S = \varnothing$, and hence, $\mathcal{C}(I, S) = \mathcal{Q}$. Because S is consistent with q^γ, the set $\mathcal{C}(I, S)$ always contains q^γ. Ideally, we would like to devise a strategy of presenting elements of $D(\mathcal{R}_o, I)$ to the user to get us "quickly" from $\varnothing$ to some S such that $\mathcal{C}(I, S) = \{q^\gamma\}$. Moreover, notice that for a consistent sample S and an unlabeled tuple t, the two possible labels of t split $\mathcal{C}(I, S)$ in two disjoint subsets. Formally, we have the following.

Lemma 4.4.1 *Given a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$, an instance $I \subseteq \mathcal{I}^K$, a consistent sample $S \subseteq \mathcal{E}^K$ over I, and an unlabeled tuple t from $D(\mathcal{R}_o, I)$, it holds that*

$$\mathcal{C}(I, S \cup \{(t, +)\}) \cup \mathcal{C}(I, S \cup \{(t, -)\}) = \mathcal{C}(I, S) \quad \textit{and}$$
$$\mathcal{C}(I, S \cup \{(t, +)\}) \cap \mathcal{C}(I, S \cup \{(t, -)\}) = \varnothing.$$

Proof The subset $\mathcal{C}(I, S \cup \{(t,+)\}) \subseteq \mathcal{C}(I,S)$ is the set of queries in $\mathcal{C}(I,S)$ that select t while the subset $\mathcal{C}(I, S \cup \{(t,-)\}) \subseteq \mathcal{C}(I,S)$ is the set of queries in $\mathcal{C}(I,S)$ that do not select t. Notice that the intersection of the two subsets is empty and their union is $\mathcal{C}(I,S)$. □

4.4.2 Instance-equivalent join predicates

It is important to note that in practice the content of the instance I may not be rich enough to allow the exact identification of the goal query q^γ i.e., when $\mathcal{C}(I, S^\gamma)$ contains elements other than q^γ. In such a case, we want to return to the user a join query that is *equivalent to q^γ w.r.t. the instance I*, and hence, indistinguishable by the user.

For example, assuming that the goal query is a equijoin, we return to the user $T(S_+)$, which is equivalent to q^γ over the instance I i.e., $q^\gamma(I) = (\bowtie_{T(S_+)} \mathcal{R})(I)$, and hence indistinguishable by the user. To clarify when such a situation occurs, take the relations P_1, P_2 with the instance I below:

$$P_1 = \begin{array}{c|cc} & A_1 & A_2 \\ \hline t_1 & 1 & 1 \end{array} \qquad P_2 = \begin{array}{c|c} & B_1 \\ \hline t'_1 & 1 \end{array}$$

and the equijoin goal query q_1^γ defined by the join predicate $\theta_1 = \{(A_1, B_1)\}$. If we present the only tuple of the Cartesian product to the user, she labels it as a positive example, which yields the sample $S_1 = \{(t_1 \cdot t'_1, +)\}$. Then, $\mathcal{C}(I, S_1) = \mathit{Join}(\{P_1, P_2\})$ and all its elements are equivalent to q_1^γ w.r.t. I. In particular, in this case we return to the user the join predicate $T(S_{1,+}) = \{(A_1, B_1), (A_2, B_1)\}$, where $\theta_1 \subsetneq T(S_{1,+})$.

Another situation when we return an instance-equivalent join query is when $q^\gamma(I)$ is empty, and therefore, the user labels all given tuples as negative examples. In such a case, we return to the user the most specific join query of that class. For example, if the goal query is an equijoin, we return the query defined by $T(S_+)$, which in this case equals Ω, which again is equivalent to q^γ over I.

4.4.3 Uninformative and informative tuples

In this section, we identify the tuples that do not yield new information when presented to the user. For this purpose, take a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$ and assume that the user has in mind the goal query q^γ that belongs to the class of queries $\mathcal{Q}$.

Let us assume for a moment that the goal q^γ is known. We say that an *example* (t, α) from S^γ is *uninformative* for the query q^γ w.r.t. an instance

$I \subseteq \mathcal{I}^K$ and a sample $S \subseteq \mathcal{E}^K$ over I if $\mathcal{C}(I,S) = \mathcal{C}(I, S \cup \{(t,\alpha)\})$. In this case, we say that t is an *uninformative tuple* w.r.t. I and S. Formally, we define the set $Uninf^K(I,S)$ of all uninformative examples w.r.t. I and S in a setting K:

$$Uninf^K(I,S) = \{(t,\alpha) \in S^\gamma \mid \mathcal{C}^K(I,S) = \mathcal{C}^K(I, S \cup \{(t,\alpha)\})\}.$$

When K is clear from the context, we write simply $Uninf(I,S)$ instead of $Uninf^K(I,S)$.

To illustrate the notion of uninformativeness, take the instance of the relations R_1 and R_2 from Example 4.2.1, the goal query q_0^γ defined by the join predicate $\{(A_2, B_3)\}$, and a sample S_0 such that $S_{0,+} = \{t_2 \cdot t'_2\}$ and $S_{0,-} = \{t_1 \cdot t'_3\}$. Notice that the examples $(t_4 \cdot t'_1, +)$ and $(t_2 \cdot t'_1, -)$ are uninformative.

Ideally, a smart inference algorithm should avoid presenting uninformative tuples to the user, but it is impossible to identify those tuples using the definition above without the knowledge of q^γ. This motivates us to introduce the notion of *certain tuples* w.r.t. an instance I and a sample S, which is independent of the goal query q^γ. Then, we prove that the notions of uninformative and certain tuples are equivalent and we identify the cases when testing the informativeness of a tuple can be done in polynomial time. We also mention that the notion of certain tuples is inspired by possible world semantics and certain answers [ILJ84] and already employed for XML querying for non-expert users by [CW13]. Formally, we define the set $Cert^K(I,S)$ of all certain examples w.r.t. I and S in a setting K:

$$\begin{aligned}
Cert^K_+(I,S) &= \{t \in D(\mathcal{R}_o, I) \mid \forall q \in \mathcal{C}^K(I,S).\ t \in q(I)\},\\
Cert^K_-(I,S) &= \{t \in D(\mathcal{R}_o, I) \mid \forall q \in \mathcal{C}^K(I,S).\ t \notin q(I)\},\\
Cert^K(I,S) &= Cert^K_+(I,S) \times \{+\} \cup Cert^K_-(I,S) \times \{-\}.
\end{aligned}$$

When K is clear from the context, we write simply $Cert(I,S)$ instead of $Cert^K(I,S)$.

We assume w.l.o.g. that all samples that we manipulate are consistent. In case of an inconsistent sample S, we have $\mathcal{C}(I,S) = \varnothing$, in which case the notion of certain tuples is of no interest. While the inclusion of $Cert(I,S)$ in $Uninf(I,S)$ is rather expected, we next show that the notions of uninformative and certain tuples are in fact equivalent.

Lemma 4.4.2 *Given a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$, an instance $I \subseteq \mathcal{I}^K$, and a consistent sample $S \subseteq \mathcal{E}^K$ over I, it holds that $Uninf(I,S) = Cert(I,S)$.*

Proof First, we show the inclusion $Uninf(I,S) \subseteq Cert(I,S)$. *Case 1.* Take a tuple t such that $(t,+) \in Uninf(I,S)$. From the definition of $\mathcal{C}$ we know that for every query q from $\mathcal{C}(I, S \cup \{(t,+)\})$ it holds that $t \in q(I)$. Because $\mathcal{C}(I,S) = \mathcal{C}(I, S \cup \{(t,+)\})$, we infer that for every query q from $\mathcal{C}(I,S)$ it holds that $t \in q(I)$, and therefore, $t \in Cert_+(I,S)$. *Case 2.* Take a tuple t such that $(t,-) \in Uninf(I,S)$. From the definition of $\mathcal{C}$ we know that for every query q from $\mathcal{C}(I, S \cup \{(t,-)\})$ it holds that $t \notin q(I)$. Because $\mathcal{C}(I,S) = \mathcal{C}(I, S \cup \{(t,-)\})$, we infer that for every query q from $\mathcal{C}(I,S)$ it holds that $t \notin q(I)$, and therefore, $t \in Cert_-(I,S)$.

Next, we prove the inclusion $Cert(I,S) \subseteq Uninf(I,S)$. *Case 1.* Take a tuple t in $Cert_+(I,S)$, which means that for every query q in $\mathcal{C}(I,S)$ it holds that $t \in q(I)$, which implies $\mathcal{C}(I,S) = \mathcal{C}(I, S \cup \{(t,+)\})$, hence $(t,+) \in Uninf(I,S)$. *Case 2.* Take a tuple t in $Cert_-(I,S)$, which means that for every query q in $\mathcal{C}(I,S)$ it holds that $t \notin q(I)$, which implies $\mathcal{C}(I,S) = \mathcal{C}(I, S \cup \{(t,-)\})$, in other words $(t,-) \in Uninf(I,S)$. ▫

Recall that we have defined the uninformative tuples w.r.t. the goal query and we have shown in Lemma 4.4.2 that $Uninf(I,S) = Cert(I,S)$, which means that we are able to characterize the uninformative tuples by using only the given sample, without having the knowledge of the goal query.

Next, let us also characterize the informative tuples i.e., those that contribute to the learning process. Take a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$, an instance $I \subseteq \mathcal{I}^K$, a sample $S \subseteq \mathcal{E}^K$ over I, and an unlabeled tuple t from $D(\mathcal{R}_o, I)$. We say that t is *informative w.r.t. K, I, and S* if there does not exist a label $\alpha \in \{+,-\}$ such that $(t,\alpha) \in Uninf(I,S)$. When K, I, and S are clear from the context, we may write simply that t is *informative*. Next, we give a necessary and sufficient condition for a tuple to be informative.

Lemma 4.4.3 *Given a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$, an instance $I \subseteq \mathcal{I}^K$, a sample $S \subseteq \mathcal{E}^K$ over I, and an unlabeled tuple t from $D(\mathcal{R}_o, I)$, t is informative iff both $S \cup \{(t,+)\}$ and $S \cup \{(t,-)\}$ are consistent.*

Proof From Lemma 4.4.2 and the definition of uninformative tuples, we infer that given a label $\alpha \in \{+,-\}$, $(t,\alpha) \in Uninf(I,S)$ means $\mathcal{C}(I,S) = \mathcal{C}(I, S \cup \{(t,\alpha)\})$. By Lemma 4.4.1, this is equivalent to $\mathcal{C}(I, S \cup \{(t,\neg\alpha)\}) = \varnothing$, which is furthermore equivalent to saying that the sample $S \cup \{(t,\neg\alpha)\}$ is not consistent. In other words t is uninformative iff there is a label $\alpha \in \{+,-\}$ such that $S \cup \{(t,\neg\alpha)\}$ is not consistent, which is equivalent to saying that t is informative iff both $S \cup \{(t,+)\}$ and $S \cup \{(t,-)\}$ are consistent. ▫

Consequently, we use the above characterization to analyze the complexity of deciding whether a tuple is informative or not. We present the summary of complexity results in Table 4.3 and we prove them in the remainder.

	Equijoins	*Semijoins*
Without disjunction	PTIME (Theorem 4.4.4)	NP-complete (Theorem 4.4.5)
With disjunction	PTIME (Theorem 4.4.4)	PTIME (Theorem 4.4.4)

Table 4.3: Summary of complexity results for deciding the informativeness of a tuple.

First, we show that the tractability of the consistency checking implies the tractability of deciding the informativeness of a tuple.

Theorem 4.4.4 *Given a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$ in* Join, UJoin, *or* UJoin$^{\ltimes}$, *an instance $I \subseteq \mathcal{I}^K$, a sample $S \subseteq \mathcal{E}^K$ over I, and an unlabeled tuple t from $D(\mathcal{R}_o, I)$, deciding whether t is informative is in PTIME.*

Proof If K is in Join, the result follows from Lemma 4.4.3 and Theorem 4.3.3. If K is in UJoin or UJoin$^{\ltimes}$, the result follows from Lemma 4.4.3 and Theorem 4.3.7. □

Next, we show that it is intractable to decide the informativeness of a tuple when the goal query is a semijoin.

Theorem 4.4.5 *Given a setting $K = (\mathcal{R}, \mathcal{R}_o, Join^{\ltimes}(\mathcal{R}, \mathcal{R}_o))$ in* Join$^{\ltimes}$, *an instance $I \subseteq \mathcal{I}^K$, a sample $S \subseteq \mathcal{E}^K$ over I, and an unlabeled tuple t from $D(\mathcal{R}_o, I)$, deciding whether t is informative is NP-complete. The result holds even when the schema consists of two relations only.*

Proof To prove the membership of the problem to NP, we point out that a Turing machine guesses a join predicate θ, which has polynomial size in the size of the input. Then, we can easily check in polynomial time whether θ is consistent with both $S \cup \{(t, +)\}$ and $S \cup \{(t, -)\}$.

To prove the NP-hardness, take the same reduction from the proof of Theorem 4.3.4. Then, add in the instance of P_φ, for every $1 \leqslant i \leqslant k$, one tuple t such that $t[id_P] = c_i^+$, $t[B_n^t] = t[B_n^f] = \perp'$, and for every $1 \leqslant j \leqslant n-1$, $t[B_j^t] = t[B_j^f] = j$. For example, for the formula φ_0 from the proof of Theorem 4.3.4, add two tuples $(c_1^+, 1, 1, 2, 2, 3, 3, \perp', \perp')$ and $(c_2^+, 1, 1, 2, 2, 3, 3, \perp', \perp')$. Then, consider S'_φ such that $S'_{\varphi,+} = \{t_{R,1}, \ldots, t_{R,k}\}$ and $S'_{\varphi,-} = \{t'_{R,0}, \ldots, t'_{R,n-1}\}$. We claim that φ is satisfiable iff the tuple $t'_{R,n}$ is informative. By Lemma 4.4.3, this is equivalent to saying that φ is satisfiable iff both $S'_\varphi \cup \{(t'_{R,n}, +)\}$ and $S'_\varphi \cup \{(t'_{R,n}, -)\}$ are consistent. Notice that $S'_\varphi \cup \{(t'_{R,n}, +)\}$ is clearly consistent since the join predicate $\{(id_R, id_P), (A_1, B_1^t), (A_1, B_1^f), \ldots, (A_{n-1}, B_{n-1}^t), (A_{n-1}, B_{n-1}^f)\}$ selects all positive and none of the negative tuples in $S'_\varphi \cup \{(t'_{R,n}, +)\}$. Thus, we have to

prove that φ is satisfiable iff $S'_\varphi \cup \{(t'_{R,n}, -)\}$ is consistent, which follows exactly as in the proof of Theorem 4.3.4. □

We end this section by proposing additional characterizations of certain tuples that hold when the input and output signatures coincide and that are useful in practice (as we show later on in the chapter when we discuss the lattice-based strategies). These additional characterizations are interesting because they are not stated in terms of consistency checking as in the general case (cf. Lemma 4.4.3). First, let us characterize the certain tuples for equijoins.

Lemma 4.4.6 *Given a setting $K = (\mathcal{R}, \mathcal{R}, Join(\mathcal{R}))$ in* Join*, an instance $I \subseteq \mathcal{I}^K$, a consistent sample $S \subseteq \mathcal{E}^K$ over I, and an unlabeled tuple t from $D(\mathcal{R}_o, I)$, it holds that:*

1. *t belongs to $Cert_+(I, S)$ iff $T(S_+) \subseteq T(t)$,*
2. *t belongs to $Cert_-(I, S)$ iff there exists a tuple t' in S_- such that $T(S_+) \cap T(t) \subseteq T(t')$.*

Proof *(1).* For the *if* part, assume $T(S_+) \subseteq T(t)$. From the definitions of $\mathcal{C}$ and T, we infer that for every θ defining a query $(\bowtie_\theta \mathcal{R})$ in $\mathcal{C}(I, S)$, it holds that $\theta \subseteq T(S_+)$, which furthermore, implies that $\theta \subseteq T(t)$, hence $t \in (\bowtie_\theta \mathcal{R})(I)$, in other words $t \in Cert_+(I, S)$.

For the *only if* part, assume $t \in Cert_+(I, S)$, which means that for every q in $\mathcal{C}(I, S)$ it holds that $t \in q(I)$. From the definitions of $\mathcal{C}$ and T, we infer that $(\bowtie_{T(S_+)} \mathcal{R})$ belongs to $\mathcal{C}(I, S)$, and therefore, $t \in (\bowtie_{T(S_+)} \mathcal{R})(I)$, which yields $T(S_+) \subseteq T(t)$.

(2). For the *if* part, take a tuple t' in S_- such that $T(S_+) \cap T(t) \subseteq T(t')$. This implies that for every q in $\mathcal{C}(I, S \cup \{(t, +)\})$ it holds that $t' \in q(I)$, hence $\mathcal{C}(I, S \cup \{(t, +)\}) = \varnothing$. By Lemma 4.4.1, we obtain $\mathcal{C}(I, S \cup \{(t, -)\}) = \mathcal{C}(I, S)$, which means that $(t, -) \in Uninf(I, S)$, and therefore, $t \in Cert_-(I, S)$ (by Lemma 4.4.2).

For the *only if* part, assume by absurd that for every t' in S_- it holds that $T(S_+) \cap T(t) \nsubseteq T(t')$, which implies that the set $\mathcal{C}(I, S \cup \{(t, +)\})$ is non-empty, hence $\mathcal{C}(S) \neq \mathcal{C}(I, S \cup \{(t, -)\})$ (by Lemma 4.4.1). This implies that $(t, -) \notin Uninf(I, S)$ that is equivalent by Lemma 4.4.2 to $(t, -) \notin Cert(I, S)$, which contradicts the hypothesis. We conclude that there exists a tuple t' in S_- such that $T(S_+) \cap T(t) \subseteq T(t')$. □

Next, let us also characterize the certain tuples for disjunctive equijoins.

Lemma 4.4.7 *Given a setting $K = (\mathcal{R}, \mathcal{R}, UJoin(\mathcal{R}))$ in* UJoin*, an instance $I \subseteq \mathcal{I}^K$, a consistent sample $S \subseteq \mathcal{E}^K$ over I, and an unlabeled tuple t from $D(\mathcal{R}_o, I)$, it holds that:*

1. *t belongs to $Cert_+(I, S)$ iff there exists a tuple $t' \in S_+$ such that $T(t') \subseteq T(t)$.*
2. *t belongs to $Cert_-(I, S)$ iff there exists a tuple $t' \in S_-$ such that $T(t) \subseteq T(t')$.*

Proof *(1).* For the *if* part, let t' be a tuple in S_+ such that $T(t') \subseteq T(t)$. From the definitions of $\mathcal{C}$ and T, we infer that for every disjunctive join predicate Θ such that $(\bowtie_\Theta \mathcal{R}) \in \mathcal{C}(I, S)$ there is a join predicate $\theta \in \Theta$ such that $\theta \subseteq T(t')$, hence $\theta \subseteq T(t)$. This implies that for every $q \in \mathcal{C}(I, S)$ it holds that $t \in q(I)$, in other words $t \in Cert_+(I, S)$.

For the *only if* part, assume by absurd that for every $t' \in S_+$ we have $T(t') \nsubseteq T(t)$. Then, take the query $(\bowtie_{\bigcup_{t' \in S_+} \{T(t')\}} \mathcal{R})$ that belongs to $\mathcal{C}(I, S)$ and that does not select t. Consequently, $t \notin Cert_+(I, S)$, which contradicts the hypothesis. Thus, we conclude that there is a tuple $t' \in S_+$ such that $T(t') \subseteq T(t)$.

(2). For the *if* part, let t' be the tuple in S_- such that $T(t) \subseteq T(t')$. Then, from the definitions of $\mathcal{C}$ and T, we infer that for every disjunctive join predicate Θ such that $(\bowtie_\Theta \mathcal{R}) \in \mathcal{C}(I, S)$, for every $\theta \in \Theta$ it holds that $\theta \nsubseteq T(t')$, hence $\theta \nsubseteq T(t)$. This implies that $t \notin q(I)$ for every $q \in \mathcal{C}(I, S)$, in other words $t \in Cert_-(I, S)$.

For the *only if* part, assume by absurd that for every $t' \in S_-$ we have $T(t) \nsubseteq T(t')$. This implies that $(\bowtie_{\bigcup_{t' \in S_+ \cup \{t\}} \{T(t')\}} \mathcal{R})$ belongs to $\mathcal{C}(I, S)$ and selects t at the same time, which contradicts the hypothesis that $t \in Cert_-(I, S)$. Thus, we conclude that there exists a tuple $t' \in S_-$ such that $T(t) \subseteq T(t')$. □

4.5 Strategies

In this section, we use the developments from the previous sections to propose efficient strategies for interactively presenting tuples to the user. First, in Section 4.5.1, we introduce the *general interactive inference algorithm* and we claim that there exists an *optimal strategy* that is however exponential. Consequently, we propose several *efficient strategies* that we essentially classify in two categories: *local* and *lookahead*. More precisely, in Section 4.5.2 we show that when the input and output signatures coincide, we can use the notion of *lattice of join predicates* to efficiently *preprocess* an instance and to

define the *local strategies*. Then, in Section 4.5.3 we propose the *lookahead strategies* that work for all settings, being based on the notion of *entropy* of a tuple that we develop there.

4.5.1 General interactive inference algorithm

Take a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$ and an instance $I \subseteq \mathcal{I}^K$. A *strategy* Υ is a function that takes as input a Cartesian product $D(\mathcal{R}_o, I)$ and a sample $S \subseteq \mathcal{E}^K$ over I, and returns a tuple t in $D(\mathcal{R}_o, I)$. The *general interactive inference algorithm* (Algorithm 5) consists of selecting a tuple w.r.t. a *strategy* Υ and asking the user to label it as a positive or negative example; this process continues until the *halt condition* Γ is satisfied. The algorithm continuously verifies the consistency of the sample, if at any moment the user labels a tuple such that the sample becomes inconsistent, the algorithm raises an exception.

We have chosen to investigate strategies that ask the user to label informative tuples only because we aim to minimize the number of interactions. Therefore, the sample that we incrementally construct is always consistent and our approach does not yield any error in lines 6-7. In our approach, we choose the strongest halt condition i.e., to stop the interactions when there is no informative tuple left:

$$\Gamma := \forall t \in D(\mathcal{R}_o, I).\ \exists \alpha \in \{+, -\}.\ (t, \alpha) \in S \cup \mathit{Uninf}(I, S).$$

At the end of the interactive process, we return the most specific join query consistent with the examples provided by the user (cf. the characterizations from Section 4.3 for different variations of the goal query class). For instance, if our goal is to infer a join query without projection and disjunction (i.e., an equijoin), we return $\theta = T(S_+)$.

However, the *halt condition* Γ may be weaker in practice, as the user might decide to stop the interactive process at an earlier time if, for instance, she finds some intermediate most specific consistent query to be satisfactory.

An optimal strategy exists and can be built by employing the standard construction of a minimax tree [RN10]. While the exact complexity of the optimal strategy remains an open question, a straightforward implementation of minimax requires exponential time (and is in PSPACE), which unfortunately renders it unusable in practice. As a consequence, we propose next a number of time-efficient strategies that attempt to minimize the number of interactions with the user and that we have implemented for our experimental study. For comparison we also introduce the *random strategy* (RND) that at each step chooses randomly an informative tuple.

Algorithm 5 General interactive inference algorithm.

Input: the Cartesian product $D(\mathcal{R}_o, I)$
Output: a join query consistent with the user's labels
Parameters: strategy Υ, halt condition Γ

1: **let** $S = \varnothing$
2: **while** $\neg\Gamma$ **do**
3: **let** $t = \Upsilon(D(\mathcal{R}_o, I), S)$
4: **query** the user about the label α for t
5: $S := S \cup \{(t, \alpha)\}$
6: **if** S is not consistent **then**
7: **error**
8: **return** the most specific query selecting all positive examples

4.5.2 Settings where the signatures coincide

When the input and output signatures coincide i.e., we have settings of the form $(\mathcal{R}, \mathcal{R}, \mathcal{Q})$, the space of all potential join predicates expressible over a given instance is captured by a *lattice of join predicates*. This lattice permits to efficiently *preprocess* the given instance and also to design some simple yet effective strategies that we call *local*. For ease of exposition of our algorithms, in the remainder we denote the Cartesian product $D(\mathcal{R}, I)$ simply by D.

Lattice of join predicates. The *lattice of the join predicates* is $(\mathcal{P}(\Omega), \subseteq)$ with $\varnothing$ as its bottom-most node and Ω as its top-most node. We focus on *non-nullable* join predicates i.e., join predicates that select at least one tuple, because we expect the user to label at least one positive example during the interactive process. We also consider Ω in case the user decides to label all tuples as negative. Naturally, the number of non-nullable join predicates may still be exponential since all join predicates are non-nullable iff there exist two tuples $t \in R$ and $t' \in P$ such that $t[A_1] = \ldots = t[A_n] = t'[B_1] = \ldots = t'[B_m]$.

We present in Figure 4.3 the lattice of join predicates for the instance from Figure 4.1 and in Figure 4.5 the lattice corresponding to the instance from Example 4.2.1. For both of them, notice a set of non-nullable nodes and the Ω. We point out a correspondence between the nodes and the tuples in the Cartesian product D: a tuple $t \in D$ corresponds to a node of the lattice θ if $T(t) = \theta$. Not every node of the lattice has corresponding tuples and in Figure 4.5 only nodes in boxes have corresponding tuples (cf. Figure 4.2).

The main insight behind the lattice-based local strategies is the following: each label given by the user is propagated in the lattice using Lemma 4.4.6 if the goal query class consists of equijoins or using Lemma 4.4.7 if the goal

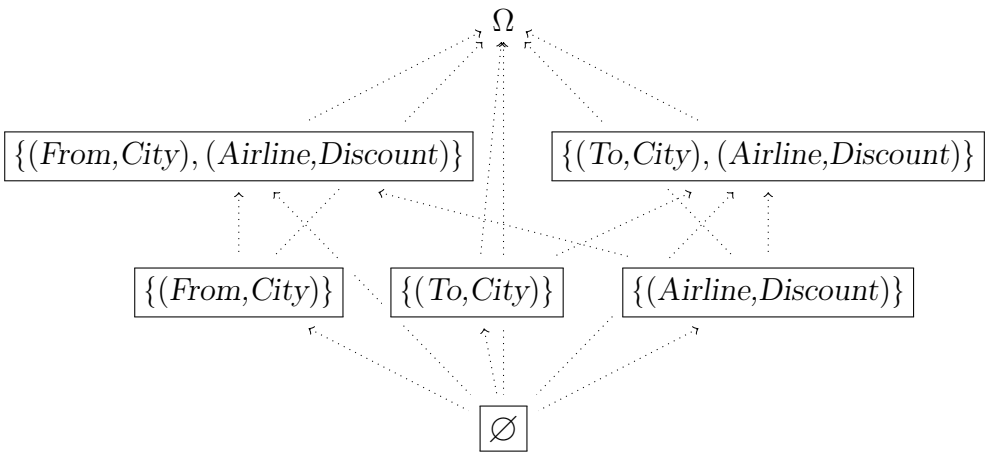


Figure 4.3: Lattice of join predicates for the instance from Figure 4.1.

query class consists of disjunctive equijoins. This allows us to *prune* parts of the lattice corresponding to the tuples that become uninformative. Basically, labeling a tuple t corresponding to a node θ as positive renders tuples corresponding to all nodes above θ uninformative and possibly some other nodes depending on tuples labeled previously (if the query class has no disjunction). Conversely, labeling t as negative prunes (at least) the part of the lattice below θ. For instance, take the lattice from Figure 4.5, assume an empty sample, and take the join predicate $\{(A_1, B_2), (A_1, B_3)\}$ and the corresponding tuple $t^\circ = t_1 \cdot t'_3$. If the user labels t° as a positive example, then the tuple $t_2 \cdot t'_3$ corresponding to $\{(A_1, B_2), (A_1, B_3), (A_2, B_1)\}$ becomes uninformative (cf. Lemma 4.4.6 or Lemma 4.4.7, respectively). On the other hand, if the user labels the tuple t° as a negative example, then the tuples $t_2 \cdot t'_1$ and $t_3 \cdot t'_1$ corresponding to $\{(A_1, B_3)\}$ and $\varnothing$ respectively, become uninformative (cf. the same two results). If we reason at the lattice level, the question "Which is the next tuple to present to the user?" intuitively becomes "Labeling which tuple allows us to prune as much of the lattice as possible?"

Preprocessing. We employ a simple mechanism to preprocess the input instance before asking the user to label any tuple. The idea is to remove redundant tuples based on the lattice of join predicates as follows: for each predicate θ such that there exists a tuple t in the Cartesian product D with $T(t) = \theta$, we take such a tuple t from the instance and we add it to the preprocessed instance.

For example, for the instance from Figure 4.1, we keep only one tuple for

From	*To*	*Airline*	*City*	*Discount*	$T(t)$
Paris	Lille	AF	NYC	AA	$\varnothing$
Paris	Lille	AF	Paris	None	{(*From*,*City*)}
Paris	Lille	AF	Lille	AF	{(*To*,*City*), (*Airline*,*Discount*)}
NYC	Paris	AA	NYC	AA	{(*From*,*City*), (*Airline*,*Discount*)}
NYC	Paris	AA	Paris	None	{(*To*,*City*)}
Paris	NYC	AF	Lille	AF	{(*Airline*,*Discount*)}

Figure 4.4: Result of preprocessing for the Cartesian product from Figure 4.1.

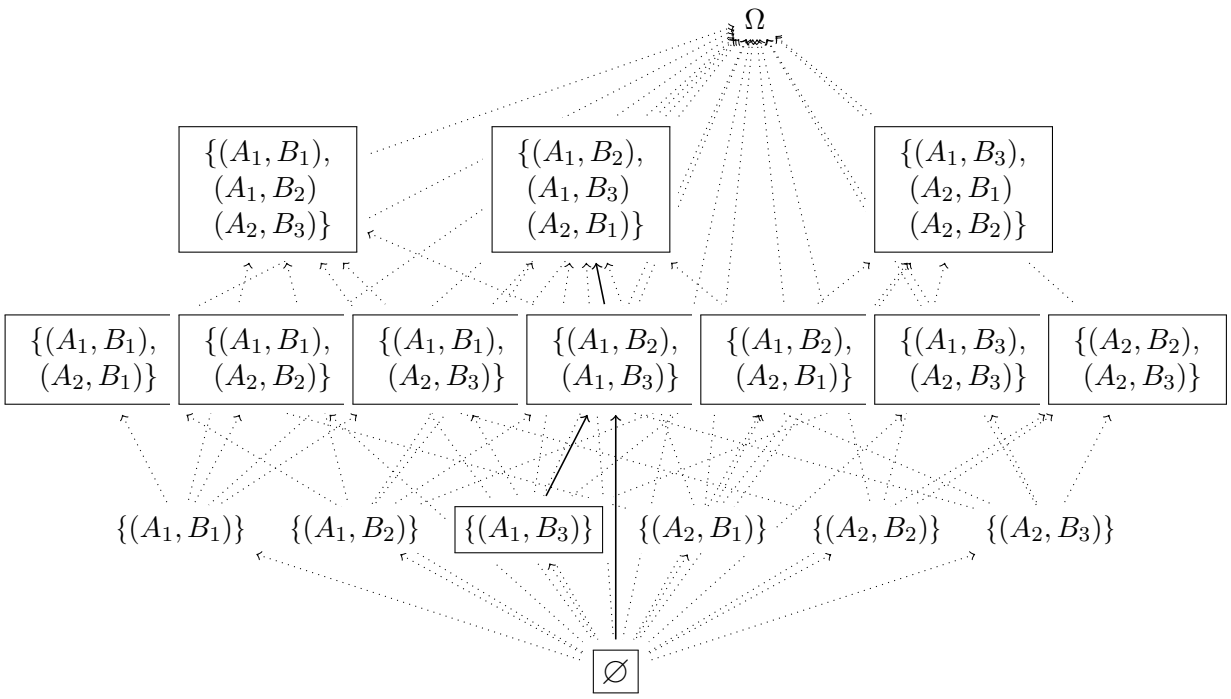


Figure 4.5: Lattice of join predicates for the instance from Example 4.2.1.

each predicate presented in a lattice node from Figure 4.3. Thus, we obtain the preprocessed instance from Figure 4.4, where we also present on the last column the corresponding $T(t)$ for each tuple. Notice that with this simple preprocessing procedure we have eliminated half of the tuples from the initial set, without altering the set of queries that can be learned on this instance. From now on, whenever we refer to an instance in the remainder, we mean in fact its preprocessed version where we have already eliminated all redundant tuples.

Even though we can easily imagine minor cases where preprocessing does not eliminate any tuple (e.g., for the instance from Example 4.2.1 whose lattice we depict in Figure 4.5), we have shown in the experimental study that it scales quite well for instances of billions of tuples that can be reduced to smaller sets of hundreds of tuples on which learning the goal query is clearly more efficient.

Local strategies. The principle behind the *local strategies* is that they propose tuples to the user following a simple order on the lattice. We call these strategies local because they do not take into account the quantity of information that labeling an informative tuple could bring to the inference process. As such, they differ from the lookahead strategies that we present in the next section. In this section we propose two local strategies, which essentially correspond to two basic variants of *navigating* in the lattice: the *bottom-up strategy* and the *top-down strategy*.

The *bottom-up strategy* (BU) (Algorithm 6) intuitively navigates the lattice of join predicates from the most general join predicate ($\varnothing$) towards the most specific one (Ω). It visits a minimal node of the lattice that has a corresponding informative tuple and asks the user to label it. If the label is positive, (at least) the part of the lattice above the node is pruned. If the label is negative, the current node is pruned (since the nodes below are not informative, they must have been pruned before). Recall the instance from Example 4.2.1 and its corresponding lattice in Figure 4.5. The BU strategy asks the user to label the tuple $t_0 = t_3 \cdot t'_1$ corresponding to $\varnothing$. If the label is positive, all nodes of the lattice are pruned and the empty join predicate returned. If the label is negative, the strategy selects the tuple $t_2 \cdot t'_1$ corresponding to the node $\theta_1 = \{(A_1, B_3)\}$ for labeling, etc. The BU strategy discovers quickly the goal join predicate $\varnothing$, but is inadequate to discover join predicates of bigger size. In the worst case, when the user provides only negative examples, the BU strategy might ask the user to label every tuple from the (preprocessed) Cartesian product.

The *top-down strategy* (TD) (Algorithm 7) intuitively starts to navigate

Algorithm 6 Bottom-up strategy BU(D, S)

1: **let** $m = \min(\{|T(t)| \mid t \in D$ such that t is informative$\})$
2: **return** informative t such that $|T(t)| = m$

in the lattice of join predicates from the most specific join predicate (Ω) to the most general one ($\varnothing$). It has basically two behaviors depending on the contents of the current sample. First, when there is no positive example yet (lines 1-2), this strategy chooses a tuple t corresponding to a $\subseteq$-maximal join predicate i.e., whose $T(t)$ has no other non-nullable join predicate above it in the lattice (line 2). For example, for the instance corresponding to the lattice from Figure 4.5, we first ask the user to label the tuple corresponding to $\{(A_1, B_1), (A_1, B_2), (A_2, B_3)\}$, then the tuple corresponding to $\{(A_1, B_2), (A_1, B_3), (A_2, B_1)\}$, etc. Note that the relative order among these tuples corresponding to $\subseteq$-maximal join predicates is arbitrary. If the user labels all $\subseteq$-maximal join predicates as negative examples, we are able to infer the goal Ω without asking her to label all the Cartesian product (using Lemma 4.4.6 or Lemma 4.4.7, depending on whether or not disjunction is present in the goal query class). Thus, the TD strategy overcomes the mentioned drawback of the BU. On the other hand, if there is at least one positive example, then the goal join predicate is a non-nullable one, and the TD strategy turns into BU (lines 3-5). As we later show in Section 4.6, the TD strategy seems a good practical compromise.

Algorithm 7 Top-down strategy TD(D, S)

1: **if** $S_+ = \varnothing$ **then**
2: **return** informative t such that $\nexists t' \in D.\ T(t) \subsetneq T(t')$
3: **else**
4: **let** $m = \min(\{|T(t)| \mid t \in D$ such that t is informative$\})$
5: **return** informative t such that $|T(t)| = m$

4.5.3 Lookahead strategies for general settings

In this section, we present the *lookahead strategies*. There are two key differences between them and the local strategies: (i) they are defined independently of the lattice of join predicates and can be thus employed in all considered settings (cf. Section 4.2); (ii) for all such settings, they take into account the *entropy* of an informative tuple i.e., the quantity of information that labeling that tuple could bring to the process of inference. More precisely, they continuously interleave the user's feedback and the inference

process by taking into account the labels already given by the user to adjust the order of presenting new tuples for labeling.

We need to introduce first some auxiliary notions. Given an informative tuple t from D and a sample S, let $u^{\alpha}_{t,S}$ be the number of tuples which become uninformative if the tuple t is labeled with α:

$$u^{\alpha}_{t,S} = |\mathit{Uninf}(I, S \cup \{(t,\alpha)\}) \setminus \mathit{Uninf}(I,S)|.$$

Now, the entropy of an informative tuple t w.r.t. a sample S, denoted $\mathit{entropy}_S(t)$, is the pair $(\min(u^{+}_{t,S}, u^{-}_{t,S}), \max(u^{+}_{t,S}, u^{-}_{t,S}))$, which captures the quantity of information that labeling the tuple t can provide. The entropy of uninformative tuples is undefined, however we never make use of it. In Figure 4.6 we present the entropy for each tuple from the Cartesian product of the instance from Example 4.2.1, for an empty sample.

	T	$u^{+}_{t,S}$	$u^{-}_{t,S}$	$\mathit{entropy}_S$
$t_1 \cdot t'_1$	$\{(A_1,B_3),(A_2,B_1),(A_2,B_2)\}$	0	2	(0,2)
$t_1 \cdot t'_2$	$\{(A_1,B_1),(A_2,B_2)\}$	0	1	(0,1)
$t_1 \cdot t'_3$	$\{(A_1,B_2),(A_1,B_3)\}$	1	2	**(1,2)**
$t_2 \cdot t'_1$	$\{(A_1,B_3)\}$	2	1	**(1,2)**
$t_2 \cdot t'_2$	$\{(A_1,B_1),(A_2,B_3)\}$	1	1	(1,1)
$t_2 \cdot t'_3$	$\{(A_1,B_2),(A_1,B_3),(A_2,B_1)\}$	0	4	(0,4)
$t_3 \cdot t'_1$	$\varnothing$	11	0	(0,11)
$t_3 \cdot t'_2$	$\{(A_1,B_3),(A_2,B_3)\}$	0	2	(0,2)
$t_3 \cdot t'_3$	$\{(A_1,B_1),(A_2,B_1)\}$	0	1	(0,1)
$t_4 \cdot t'_1$	$\{(A_1,B_1),(A_1,B_2),(A_2,B_3)\}$	0	2	(0,2)
$t_4 \cdot t'_2$	$\{(A_1,B_2),(A_2,B_1)\}$	1	1	(1,1)
$t_4 \cdot t'_3$	$\{(A_2,B_2),(A_2,B_3)\}$	0	1	(0,1)

Figure 4.6: The Cartesian product corresponding to the instance from Example 4.2.1 and the entropy for each tuple, for an initial empty sample.

Given two entropies $e = (a,b)$ and $e' = (a',b')$, we say that e *dominates* e' if $a \geqslant a'$ and $b \geqslant b'$. For example, $(1,2)$ dominates $(1,1)$ and $(0,2)$, but it does not dominate $(2,2)$ nor $(0,3)$. Next, given a set of entropies E, we define the *skyline* of E, denoted $\mathit{skyline}(E)$, as the set of entropies e that are not dominated by any other entropy of E. For example, for the set of entropies of the tuples from Figure 4.6, the skyline is $\{(1,2),(0,11)\}$.

Next, we present the *one-step lookahead skyline strategy* ($\mathrm{L}^1\mathrm{S}$) (Algorithm 8). We illustrate this strategy for the instance from Example 4.2.1, for an initial empty sample. First (line 1), we compute the entropy for each informative tuple from the Cartesian product. This corresponds to computing

the last column from Figure 4.6. Then (line 2), we calculate the maximal value among all minimal values of the entropies computed at the previous step. For our example, this value is 1. Finally (lines 3-4), we return an informative tuple whose entropy is in the skyline of all entropies, and moreover, has as minimal value the number computed at the previous step. For our example, the skyline is $\{(1,2),(0,11)\}$, thus the entropy corresponding to the value computed previously (i.e., 1) is $(1,2)$. Consequently, we return one of the tuples having the entropy $(1,2)$, more precisely either $t_1 \cdot t'_3$ or $t_2 \cdot t'_1$. Intuitively, according to L^1S strategy, we choose to ask the user to label a tuple which permits to eliminate at least one and at most two additional tuples. Note that by min (resp. max) we denote the minimal (resp. maximal) value from either a given set or a given pair of numbers, depending on the context.

Algorithm 8 One-step lookahead skyline $\text{L}^1\text{S}(D, S)$

1: **let** $E = \{entropy_S(t) \mid t \in D$ such that t is informative$\}$
2: **let** $m = \max(\{\min(e) \mid e \in E\})$
3: **let** e the entropy in $skyline(E)$ such that $\min(e) = m$
4: **return** informative t such that $entropy_S(t) = e$

The L^1S strategy naturally extends to *k-steps lookahead skyline strategy* (L^kS). The difference is that instead of taking into account the quantity of information that labeling *one* tuple could bring to the inference process, we take into account the quantity of information for labeling k tuples. Note that if k is greater than the total number of informative tuples in the Cartesian product, then the strategy becomes optimal and thus inefficient. For such a reason, in the experiments we focus on a lookahead of two steps, which is a good trade-off between keeping a relatively low computation time and minimizing the number of interactions. Therefore, we present such strategy in the remainder.

We need to extend first the notion of entropy of a tuple to the notion of *entropy*^{2} of a tuple. Given an informative tuple t and a sample S, the *entropy*^{2} of t w.r.t. S, denoted $entropy^2_S(t)$, intuitively captures the minimal quantity of information that labeling t and another tuple can bring to the inference process. The construction of the *entropy*^{2} is quite technical (Algorithm 9) and we present an example below. Take the sample $S = \{(t_1 \cdot t'_3, +), (t_3 \cdot t'_1, -)\}$. Note that $Uninf(I, S) = \{(t_2 \cdot t'_3, +), (t_1 \cdot t'_2, -), (t_2 \cdot t'_2, -), (t_3 \cdot t'_3, -), (t_4 \cdot t'_3, -)\}$. There are five informative tuples left: $t_1 \cdot t'_1$, $t_2 \cdot t'_1$, $t_3 \cdot t'_2$, $t_4 \cdot t'_1$, and $t_4 \cdot t'_2$. Let compute now the *entropy*^{2} of $t_2 \cdot t'_1$ w.r.t. S using Algorithm 9. First, take $\alpha = +$ (line 1), then $S' = S \cup \{(t_2 \cdot t'_1, +)\}$ (line 2), note that there is no other informative tuple left (line 3), and therefore, $e_+ = (\infty, \infty)$ (lines 4-5). This intuitively means that given the sample S, if

Algorithm 9 $entropy^2_S(t)$

1: **for** $\alpha \in \{+, -\}$ **do**
2: **let** $S' = S \cup \{(t, \alpha)\}$
3: **if** $\nexists t' \in D$ such that t' is informative w.r.t. S' **then**
4: **let** $e_\alpha = (\infty, \infty)$
5: **continue**
6: **let** $E = \varnothing$
7: **for** $t' \in D$ s.t. i is informative w.r.t. S' **do**
8: **let** $u^+ = |Uninf(I, S \cup \{(t, \alpha), (t', +)\}) \setminus Uninf(I, S)|$
9: **let** $u^- = |Uninf(I, S \cup \{(t, \alpha), (t', -)\}) \setminus Uninf(I, S)|$
10: $E := E \cup \{(\min(u^+, u^-), \max(u^+, u^-))\}$
11: **let** $m = \max(\{\min(e) \mid e \in E\})$
12: **let** e_α the entropy in $skyline(E)$ s.t. $\min(e_\alpha) = m$
13: **let** $m = \min(\{\min(e_+), \min(e_-)\})$
14: **return** e_α such that $\min(e_\alpha) = m$

the user labels the tuple $t_2 \cdot t'_1$ as positive example, then there is no informative tuple left and we can stop the interactions. Next, take $\alpha = -$ (line 1), then $S' = S \cup \{(t_2 \cdot t'_1, -)\}$ (line 2), and the only tuples informative w.r.t. S' are $t_4 \cdot t'_1$ and $t_4 \cdot t'_2$, we obtain $E = \{(3, 3)\}$ (lines 6-10), and $e_- = (3, 3)$ (lines 11-12). Finally, $entropy^2_S(t_2 \cdot t'_1) = (3, 3)$ (lines 13-14), which means that if we ask the user to label the tuple $t_2 \cdot t'_1$ and any arbitrary tuple afterwards, then there are at least three other tuples that become uninformative. The computation of the entropies of the other informative tuples w.r.t. S is done in a similar manner. The *2-steps lookahead skyline strategy* (L^2S) (Algorithm 10) returns a tuple corresponding to the "best" $entropy^2$ in a similar manner to L^1S. In fact, Algorithm 10 has been obtained from Algorithm 8 by simply replacing *entropy* by $entropy^2$. As we have already mentioned, the approach can be easily generalized to $entropy^k$ and L^kS, respectively.

Algorithm 10 2-steps lookahead skyline $L^2S(D, S)$

let $E = \{entropy^2_S(t) \mid t \in D$ such that t is informative$\}$
let $m = \max(\{\min(e) \mid e \in E\})$
let e the entropy in $skyline(E)$ such that $\min(e) = m$
return informative t such that $entropy^2_S(t) = e$

4.6 Experiments

In this section, we present an experimental study devoted to gauging the efficiency and effectiveness of the join inference strategies presented above. Precisely, we compare three classes of strategies: the random strategy (RND), the local strategies (BU and TD), and the lookahead strategies (L^1S and L^2S). For each input database instance and for each goal query, we have considered three measures: the *number of user interactions* (i.e., the number of tuples that need to be presented to the user to label as examples in order to infer the join predicate), the *total time* needed to infer the goal join predicate, and the *time between examples*, using each of the above strategies as strategy Υ (cf. Section 4.5.1), and reiterating the user interactions until no informative tuple is left (halt condition Γ). Throughout the experimental section, by *join size* we intend the number of equalities from a (disjunctive) join predicate, while by *lattice size* we denote the number of nodes in the lattice of join predicates (cf. Section 4.5.2).

In our experiments, we have employed two datasets: the TPC-H benchmark datasets (Section 4.6.1) and a synthetic dataset we have built in the spirit of our introductory motivating example on Flight&Hotel (Section 4.6.2). The presented results cover the entirety of the TPC-H queries involving joins. Moreover, to cope with the lack of disjunction in the TPC-H benchmark queries, we have built a synthetic Flight&Hotel dataset generator to be able to gauge our learning algorithms when disjunction is allowed. We discuss the results for the two datasets in Section 4.6.3.

It is also important to point out that in our experiments we focused only on learning settings where the input and output signatures coincide (i.e., on learning equijoins) and there are two reasons for this choice. First, we wanted to be able to fairly compare local and lookahead strategies (recall that the local strategies are meaningful only in such learning settings). Second, we recall that when the input and output signatures differ (i.e., for semijoins), there are cases where the problems of interest become intractable (cf. Theorem 4.3.4 and Theorem 4.4.5). Since we focused only on settings where the input and output signatures coincide, we were also able to apply a preprocessing technique in the spirit of Section 4.5.2 to remove redundant tuples before asking the user to label any tuple. Thus, from an initial large instance we have constructed a preprocessed smaller one that basically contains as many tuples as nodes in the lattice.

All our algorithms have been implemented in C. All our experiments were run on an Intel Core i7 with 4×2.9 GHz CPU and 8 GB RAM.

4.6.1 Setup of experiments on TPC-H

The TPC-H benchmark [1] contains the following eight tables (with the corresponding abbreviation in parenthesis): *part* (P), *supplier* (S), *partsupp* (PS), *lineitem* (L), *orders* (O), *customer* (C), *nation* (N), *region* (R).

Out of the 22 queries provided by the TPC-H benchmark, 20 exhibit join predicates. Since many of the queries contain aggregates, arithmetic expressions, groupby statements, etc. that fall beyond the scope of our learning techniques, and similarly to [ZEPS13], we modify the TPC-H queries by dropping all such operators while maintaining the join predicates. By performing this operation, some of the TPC-H queries actually become equivalent. Thus, we obtain a total of 15 different queries that we present in Table 4.4. As an example, the first line of the table indicates that the TPC-H queries 4 and 12 have the same join condition "L[orderkey] = O[orderkey]" between the tables L and O. When a table appears more than once in a query, we use Arabic numbers to differentiate between these occurrences (e.g., N1 and N2 for query 7). Additionally, we saturate the join predicates by adding all join conditions that result by transitivity e.g., for query 21 we have "L1[orderkey] = L2[orderkey]" and "L1[orderkey] = L3[orderkey]" implies that "L2[orderkey] = L3[orderkey]". In Table 4.4, we also illustrate the sizes of the considered joins, which span from 1 to 8.

In our TPC-H experiments, we extracted lattices based on the instances found in the *ref_ data* directory that is provided with the TPC-H benchmark. There are eight such instances (that correspond to scaling factors from 1 to 100000). The goal of the preprocessing phase was to translate such instances in lattices on which we implemented our interactive learning strategies. The lattices that we considered for TPC-H have at most around 200 elements and we report these numbers in Figure 4.7. Recall that these instances have not been generated with different scaling factors. They correspond to instances that are provided with the TPC-H benchmark and have not the same scaling factors of generated instances. Since the lattice of join predicates depends on the tables on which the user wants to specify her join query, and moreover, each query targets a different combination of tables, we had a different lattice for each query (and each scaling factor). Hence, our experiments can be seen like: given a lattice of a certain complexity (defined mainly by the number of its elements), how many examples from the lattice should the user be asked to label to be able to learn an arbitrary query that she has in mind? For us, the TPC-H instances from the *ref_ data* directory were the basis to construct such lattices. The fact that we considered eight instances implies

[1] http://www.tpc.org/

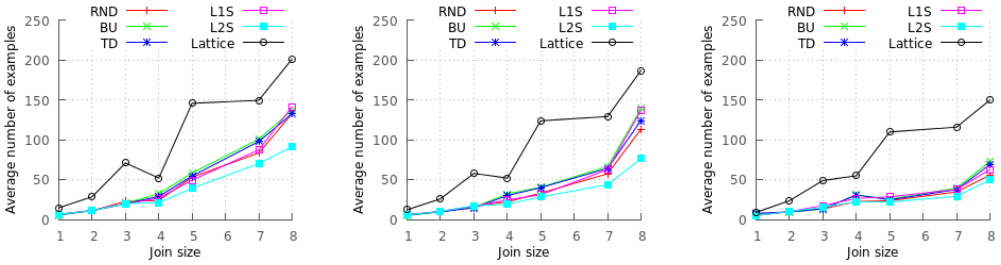
Query	*Tables*	*Join predicate*	*Size*
4,12	L,O	L[orderkey] = O[orderkey]	1
13,22	C,O	C[custkey] = O[custkey]	1
14,17,19	L,P	L[partkey] = P[partkey]	1
15	L,S	L[suppkey] = S[suppkey]	1
11	PS,S,N	PS[suppkey] = S[suppkey] $\wedge$ S[nationkey] = N[nationkey]	2
16	PS,S,P	PS[partkey] = P[partkey] $\wedge$ PS[suppkey] = S[suppkey]	2
3,18	C,O,L	C[custkey] = O[custkey] $\wedge$ L[orderkey] = O[orderkey]	2
10	C,O,L,N	C[custkey] = O[custkey] $\wedge$ L[orderkey] = O[orderkey] $\wedge$ C[nationkey] = N[nationkey]	3
2	P,S,PS,N,R	PS[partkey] = P[partkey] $\wedge$ PS[suppkey] = S[suppkey] $\wedge$ S[nationkey] = N[nationkey] $\wedge$ N[regionkey] = R[regionkey]	4
7	S,L,O,C,N1,N2	S[suppkey] = L[suppkey] $\wedge$ L[orderkey] = O[orderkey] $\wedge$ C[custkey] = O[custkey] $\wedge$ S[nationkey] = N1[nationkey] $\wedge$ C[nationkey] = N2[nationkey]	5
20	P,S,PS,N,L	PS[partkey] = P[partkey] $\wedge$ PS[partkey] = L[partkey] $\wedge$ P[partkey] = L[partkey] $\wedge$ PS[suppkey] = S[suppkey] $\wedge$ PS[suppkey] = L[suppkey] $\wedge$ S[suppkey] = L[suppkey] $\wedge$ S[nationkey] = N[nationkey]	7
5	C,O,L,S,N,R	C[custkey] = O[custkey] $\wedge$ L[orderkey] = O[orderkey] $\wedge$ S[suppkey] = L[suppkey] $\wedge$ C[nationkey] = S[nationkey] $\wedge$ S[nationkey] = N[nationkey] $\wedge$ C[nationkey] = N[nationkey] $\wedge$ N[regionkey] = R[regionkey]	7
8	P,S,L,O,C,N1,N2,R	L[partkey] = P[partkey] $\wedge$ S[suppkey] = L[suppkey] $\wedge$ L[orderkey] = O[orderkey] $\wedge$ C[custkey] = O[custkey] $\wedge$ C[nationkey] = N1[nationkey] $\wedge$ N1[regionkey] = R[regionkey] $\wedge$ S[nationkey] = N2[nationkey]	7
9	P,S,L,PS,O,N	S[suppkey] = L[suppkey] $\wedge$ PS[suppkey] = L[suppkey] $\wedge$ PS[suppkey] = S[suppkey] $\wedge$ PS[partkey] = L[partkey] $\wedge$ L[partkey] = P[partkey] $\wedge$ PS[partkey] = P[partkey] $\wedge$ L[orderkey] = O[orderkey] $\wedge$ S[nationkey] = N[nationkey]	8
21	S,O,N,L1,L2,L3	L1[orderkey] = L2[orderkey] $\wedge$ L1[orderkey] = L3[orderkey] $\wedge$ L2[orderkey] = L3[orderkey] $\wedge$ L1[orderkey] = O[orderkey] $\wedge$ L2[orderkey] = O[orderkey] $\wedge$ L3[orderkey] = O[orderkey] $\wedge$ L1[suppkey] = S[suppkey] $\wedge$ S[nationkey] = N[nationkey]	8

Table 4.4: The TPC-H joins.

only slightly diverse input instances, which however induced lattices of about the same size. Actually, since the results for the eight instances were very similar, we report the results for only three of them among which those corresponding to the extreme scaling factors i.e., 1 and 100000.

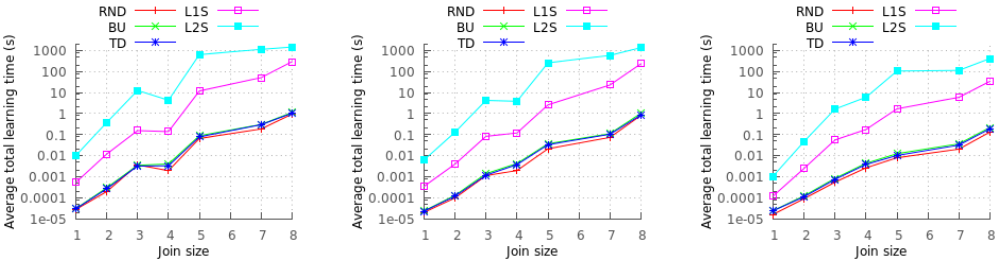
We recall that our interactive strategies are ignoring the integrity constraints from the TPC-H schema and propose tuples to present to the user only by reasoning on the user annotations. The goal of such experiments on TPC-H is to evict the join predicates that rely on integrity constraints and that belong to the user's goal query. Nevertheless, attributes other than the ones involved in the keys and foreign keys may be involved in the join predicates as they exhibit compatible types. For instance, a value "15" of an attribute of a tuple may as well represent a key, a size, a price, or a quantity, etc. Actually, for a *ref_ data* instance corresponding to a higher scaling factor, the number of equalities between such attributes exhibiting compatible types decreases because the domains of the attributes are sparser. This explains the fact that the instances corresponding to larger scaling factors induce smaller lattices (cf. Figure 4.7).

As already mentioned, the lattice of join predicates depends on the query that the user has in mind in the sense that the lattice is constructed based on the tables that the user wants to join. This is rather a natural assumption, since we assume that the user knows what tables are targeted by the query that she has in mind. The preprocessing transforms these tables into a lattice on which the actual learning is done. In situations where one relation appears more than once (as in some of the TPC-H queries), we simply take that relation with different aliases. Thus, we can accommodate join paths of an arbitrary length, with multiple occurrences of the same relation. Notice that there are two aspects of complexity of the lattice. One of them is the number of its elements, that we report in Figure 4.7 and changes from query to query since each query is formulated over a different combination of tables. The other aspect is the size of an element of the lattice i.e., the number of equalities that can be formulated over the given instance. Although this number is rather small in case of joins across two tables only, there are cases such as those where a relation is taken several times when the size of an element can easily become large (e.g., for the TPC-H query 21, where Lineitem is taken 3 times, there are over 200 equalities, out of which more than half are due to equalities between attributes of Lineitem in the different occurrences of that table). In such cases, we do not construct the lattice such that it contains precisely one element per combination of equalities as described in the Section 4.5.2 (that could go in the worst case up to 2^{200} elements in the aforementioned example), but we randomly extract a subset of it, that we fixed of size as large as the number of equalities (e.g., 200).



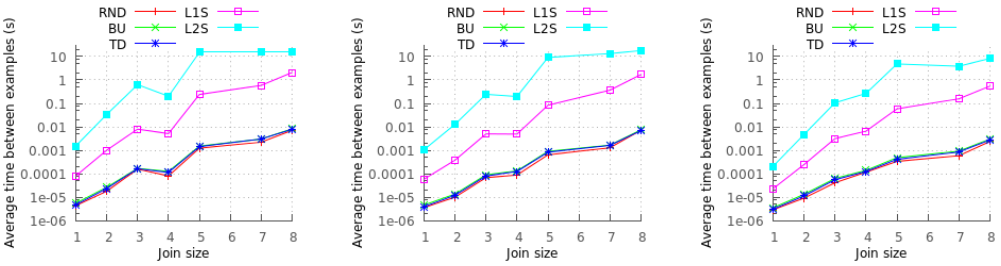
(a) SF 1. (b) SF 100. (c) SF 100000.

Figure 4.7: Number of examples for learning TPC-H joins.



(a) SF 1. (b) SF 100. (c) SF 100000.

Figure 4.8: Total learning time for the TPC-H joins.

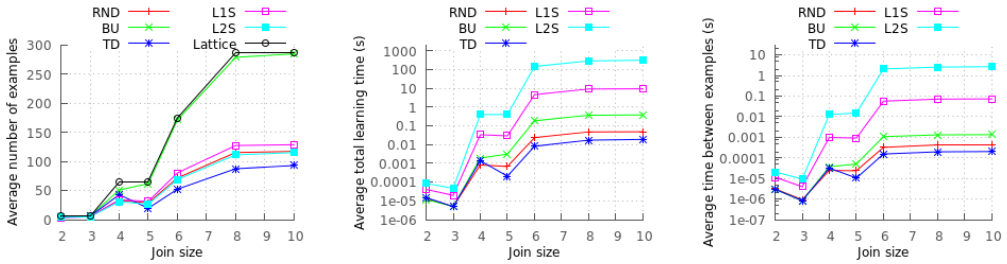


(a) SF 1. (b) SF 100. (c) SF 100000.

Figure 4.9: Time between examples for the TPC-H joins.

Query	*Tables*	*Disjunctive join predicate*	*Size*
1	F, H	F.From = H.City $\vee$ F.To = H.City	2
2	F, H	(F.From = H.City $\vee$ F.To = H.City) $\wedge$ F.Airline = H.Discount	3
3	F1, H, F2	(F1.To = H.City $\vee$ H.City = F2.From) $\wedge$ (F1.Airline = H.Discount $\vee$ H.Discount = F2.Airline)	4
4	F1, H, F2	F1.To = H.City $\wedge$ F1.To = F2.From $\wedge$ H.City = F2.From $\wedge$ (F1.Airline = H.Discount $\vee$ H.Discount = F2.Airline)	5
5	F1, H, F2	F1.From = F2.To $\wedge$ F1.To = H.City $\wedge$ F1.To = F2.From $\wedge$ H.City = F2.From $\wedge$ (F1.Airline = H.Discount $\vee$ H.Discount = F2.Airline)	6
6	F1, H1, F2, H2	F1.To = H1.City $\wedge$ F1.To = F2.From $\wedge$ H1.City = F2.From $\wedge$ F2.To = H2.City $\wedge$ (F1.Airline = H1.Discount $\vee$ F2.Airline = H2.Discount)	6
7	F1, H1, F2, H2	F1.To = H1.City $\wedge$ F1.To = F2.From $\wedge$ H1.City = F2.From $\wedge$ F2.To = H2.City $\wedge$ (F1.Airline = H1.Discount $\vee$ F1.Airline = H2.Discount $\vee$ F2.Airline = H1.Discount $\vee$ F2.Airline = H2.Discount)	8
8	F1, H1, F2, H2	F1.To = H1.City $\wedge$ F1.To = F2.From $\wedge$ H1.City = F2.From $\wedge$ F2.To = H2.City $\wedge$ (F1.Airline = H1.Discount $\vee$ F1.Airline = H2.Discount) $\wedge$ (F2.Airline = H1.Discount $\vee$ F2.Airline = H2.Discount)	8
9	F1, H1, F2, H2	F1.From = F2.To $\wedge$ F1.From = H2.City $\wedge$ F1.To = H1.City $\wedge$ F1.To = F2.From $\wedge$ H1.City = F2.From $\wedge$ F2.To = H2.City $\wedge$ (F1.Airline = H1.Discount $\vee$ F1.Airline = H2.Discount $\vee$ F2.Airline = H1.Discount $\vee$ F2.Airline = H2.Discount)	10
10	F1, H1, F2, H2	F1.From = F2.To $\wedge$ F1.From = H2.City $\wedge$ F1.To = H1.City $\wedge$ F1.To = F2.From $\wedge$ H1.City = F2.From $\wedge$ F2.To = H2.City $\wedge$ (F1.Airline = H1.Discount $\vee$ F1.Airline = H2.Discount) $\wedge$ (F2.Airline = H1.Discount $\vee$ F2.Airline = H2.Discount)	10

Table 4.5: The Flight&Hotel joins.



(a) Number of examples. (b) Total learning time. (c) Time between examples.

Figure 4.10: Summary of results for the Flight&Hotel joins.

We present the experimental results on TPC-H in Figure 4.7 (the number of needed examples), Figure 4.8 (the total learning time), and Figure 4.9 (the time between two interactions). We discuss these experiments along with the synthetic ones in Section 4.6.3.

4.6.2 Setup of experiments on synthetic data

Since the TPC-H join predicates have only conjunctions, we have implemented a synthetic datasets generator to tweak our strategies also in the context of learning disjunctive join queries. We have generated tables of flights and hotels, in the spirit of our motivating example. More precisely, the relation Flight has 3 attributes (From, To, Airline) and the relation Hotel has 2 attributes (City, Discount). We abbreviate them by F and H, respectively. Moreover, when a table appears more than once in a query, we use Arabic numbers to differentiate between these occurrences (e.g., F1 and F2). In Table 4.5, we present the list of studied queries, which contain between 2 and 10 equalities. We illustrate the experimental results in Figure 4.10 for both number of examples and learning time, and we discuss them along with the TPC-H ones in Section 4.6.3.

4.6.3 Discussion

In this section, we discuss the experimental results for the two aforementioned settings.

First, we would like to point out that our implementation of queries and lattices is not specific to the problem of learning join queries. Thus, our strategies are applicable to any problem that can be reduced to finding informative nodes on a lattice, which can be represented with any general encoding. More precisely, for both datasets, we implemented a query as

a Boolean array, having as length the total number of equalities that can be formulated over the given instance. For example, for our Flight&Hotel running example, there are 3 such equalities (From=City, To=City, and Airline=Discount, respectively), hence a query is a Boolean array of length 3. In particular, the query To=City$\wedge$Airline=Discount can be seen as [0,1,1]. Then, a lattice is a collection of such Boolean arrays. For example, the lattice from Figure 4.3 can be seen as {[0,0,0], [0,0,1], [0,1,0], [1,0,0], [0,1,1], [1,0,1]}. Hence, as long as a problem can be reduced to finding informative nodes on a lattice, one can leverage our interactive strategies, which therefore have applications beyond the classes of queries studied in the paper.

The lattice size intuitively captures the complexity of an instance i.e., the larger it is, the more join predicates are in the lattice (cf. Section 4.5.2), and therefore, more interactions are needed to infer a join query on that instance. This explains the experimental results for the number of needed examples that we show in Figure 4.7 and Figure 4.10, respectively. We can observe that in the majority of cases TD and L^2S are better than the other strategies w.r.t. minimizing the number of interactions. However, none of them seems to be a winning strategy overall. In fact, their performance essentially depends on both the size of the join predicate (reported in Table 4.4 and Table 4.5) and the lattice size (reported in Figure 4.7 and Figure 4.10), but also on whether or not we allow disjunction in the goal query class.

It is important to point out that on all presented figures we included on the X axis the size of the goal join query, which spans between 1 and 8 for TPC-H, and between 2 and 10 for Flight&Hotel. In each graph, a value corresponding to a certain join size is actually obtained by averaging over all queries of that size. Thus, we can easily observe that a goal query of smaller size can be learned with less interactions. Such a trend is confirmed for each dataset.

Next, let us discuss how the number of interactions needed for each strategy depends on the lattice size. The essential insight is that L^2S exhibits a better performance than TD with significant lattice size. A large lattice entails a higher number of join predicates in the lattice, and therefore, in such cases the inference requires more interactions. This explains the fact that for the most complex TPC-H joins L^2S is the best strategy w.r.t. minimizing the number of examples. By opposite, with small lattices, and, consequently, fewer join predicates, the lookahead strategy might not be necessarily useful. Interestingly, when we also allow the disjunction in the synthetic experiments, we observe that L^2S is no longer better than TD either. Intuitively, this happens because in the presence of disjunction the characterizations able to prune parts of the lattice are less aggressive than without disjunction (cf. Lemma 4.4.6 and Lemma 4.4.7, respectively). Thus, the entropies computed

by the algorithm are smaller and choosing the best one among them is not necessarily better than a simple, local strategy.

Additionally, we observe that the total learning time for all strategies is always within a reasonable range, even though it may vary within this range with the different strategies. While for the random and local strategies the total learning time is always under a second, it can go up to at most a thousand of seconds for the lookahead strategies for the most complex TPC-H queries. However, such queries are the ones on which the lookahead strategy is most beneficial, which explains the difference in terms of learning time. As for the time between two examples, for TPC-H it is of roughly up to 10 seconds for two-steps lookahead, 1 second for one-step lookahead, and 0.01 seconds for local and random strategies. For the Flight&Hotel synthetic queries that also consider disjunction, the time between examples is always up to 3 seconds for two-step lookahead and less than 0.1 seconds for the others. Thus, this time is reasonably small for both datasets.

To summarize our experimental results, we point out that TD and L^2S can be considered as a good trade-off between optimizing our antagonistic goals: minimizing both the number of user examples and the learning time. More precisely, if on one hand we have only conjunctions and a small lattice, or we allow disjunctions, TD is the best strategy; conversely, if on the other hand we have only conjunctions and more complex queries on dense lattices, L^2S is the strategy to be chosen.

4.7 Related work

A wealth of research on using computational learning theory [KV94] has been recently conducted in databases [AAP+13, BGNV10, LMN10, SW12, tCDK13]. Very recently, algorithms for learning relational queries (e.g., quantifiers [AAP+13]) or XML queries (e.g., tree patterns [SW12]) have been proposed. Besides learning queries, researchers have investigated the learnability of relational schema mappings [tCDK13], as well as schemas [BGNV10] and transformations [LMN10] for XML. In this section, we discuss the positioning of our own work w.r.t. these and other papers.

Our work follows a very recent line of research on the inference of relational queries [ZEPS13, TCP09, DSPGMW10]. [ZEPS13] have focused on computing a join query starting from a database instance, its complete schema, and an output table. Clearly, their assumptions are different from ours. In particular, we do not assume any knowledge of the integrity constraints or the query result. In our approach, the latter has to be incrementally constructed via multiple interactions with the user, along with the join

predicate itself. [ZEPS13] consider more expressive queries than we do, but when the integrity constraints are unknown, one can leverage our algorithms to yield those and apply their approach thereafter. Moreover, [TCP09] have investigated the query by output problem: given a database instance, a query statement and its output, construct an instance-equivalent query to the initial statement. [DSPGMW10] have studied the view definition problem i.e., given a database instance and a corresponding view instance, find the most succinct and accurate view definition. Both [TCP09] and [DSPGMW10] essentially use decision trees to classify tuples as selected or not selected in the query output or in the view output, respectively. We differ from their work in two ways: we do not know a priori the query output, and we need to discover it from user interactions; we have no initial query statement to start with.

The learnability definition that we employ in Section 4.3 is based on the standard framework of *language identification in the limit with polynomial time and data* [Gol78] adapted to learning join queries. Then, the interactive scenario studied in Section 4.4 is inspired by the well-known framework of *learning with membership queries* [Ang88].

A problem closely related to learning is *definability*. More precisely, [Ban78] and [Par78] have studied the decision problem, given a pair of relational instances, whether there exists a relational algebra expression which maps the first instance to the second one. Their research led to the notion of *BP-completeness*. Their results were later extended to the nested relational model [VG87] and to sequences of input-output pairs [FGPVG09]. Learning and definability have in common the fact that they look for a query consistent with a set of examples. The difference is that learning allows the query to select or not the tuples that are not explicitly labeled as positive or negative examples while definability requires the query to select nothing else than the set of positive examples (i.e., all the other tuples are implicitly negative).

[FGLX11] have worked on discovering conditional functional dependencies using data mining techniques. We focus on simpler join constraints, and exploit an interactive scenario to discover them by interacting with the users.

Since our goal is to find the most informative tuples and ask the user to label them, our research is also related to the work of [YZI+13]. However, we do not consider keyword-based queries. Another work strongly related to ours has been done by [AAP+13, AHS12], who have formalized a query learning model using membership questions [Ang88]. They focus on learning quantified Boolean queries for the nested relational model and their main results are optimal algorithms for learning some subclasses of such queries [AAP+13] and a system that helps users specify quantifiers [AHS12]. Primary-foreign key relationships between attributes are used to place quantified constraints

and help the user tune her query, whereas we do not assume such knowledge. The goal of their system is somewhat different, in that their goal is to disambiguate a natural language specification of the query, whereas we focus on raw data to guess the "unknown" query that the user has in mind. The theoretical foundations of learning with membership queries have been studied in the context of schema mappings [tCDK13]. Moreover, [AtCKT11a, AtCKT11b] have proposed a system which allows a user to interactively design and refine schema mappings via data examples. The problem of discovering schema mappings from data instances have been also studied in [GS10] and [QCJ12]. Our queries can be eventually seen as simple GAV mappings, even though our problem goes beyond data integration. Moreover, our focus is on proposing tuples to the user, while [AtCKT11a, AtCKT11b] assume that an expert user chooses the data examples. Additionally, our notions of certain and uninformative tuples have connections with the approach of [CW13] for XPath queries, even though joins are not considered there. Furthermore, our notion of entropy of a tuple is related to the work of [SK13] on exploratory querying big data collections.

Chapter 5

Learning path queries on graph databases

In this chapter, we investigate the problem of learning graph queries by exploiting user examples. The input consists of a graph database in which the user has labeled a few nodes as *positive* or *negative examples*, depending on whether or not she would like the nodes as part of the query result. Our goal is to handle such examples to find a query whose output is what the user expects. This kind of scenario is pivotal in several application settings where unfamiliar users need to be assisted to specify their queries. We focus on *path queries* defined by *regular expressions*, we identify fundamental difficulties of our problem setting, we formalize what it means to be *learnable*, and we prove that the class of queries under study enjoys this property. We additionally investigate an interactive scenario where we start with an empty set of examples and we identify the *informative nodes* i.e., those that contribute to the learning process. Then, we ask the user to label these nodes and iterate the learning process until she is satisfied with the learned query. Finally, we present an experimental study on both real and synthetic datasets devoted to gauging the effectiveness of our learning algorithm and the improvement of the interactive approach.

5.1 Context

Graph databases [Woo12] are becoming pervasive in several application scenarios such as the Semantic Web [AP11], social [RS09] and biological [PNLH09] networks, and geographical databases [AG08], to name a few. A graph database is essentially a directed, edge-labeled graph. As an example, consider in Figure 5.1 a graph representing a geographical database having as nodes the neighborhoods of a city area (N_1 to N_6), along with cinemas (C_1 and C_2), and restaurants (R_1 and R_2) in such neighborhoods. The edges represent public transportation facilities from a neighborhood to another (using labels *tram* and *bus*), along with other kind of facilities (using labels *cinema* and *restaurant*). For instance, the graph indicates that one can travel by bus

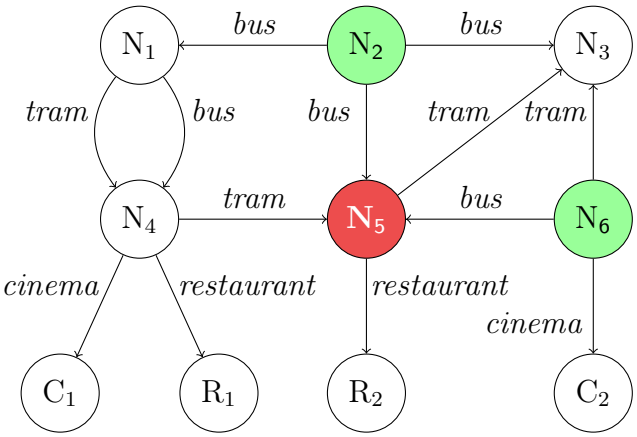


Figure 5.1: A geographical graph database.

between the neighborhoods N_2 and N_3, that in the neighborhood N_4 exists a cinema C_1, and so on.

Many mechanisms have been proposed to query a graph database, the majority of them being based on *regular expressions* [Bar13, Woo12]. By continuing on our running example, imagine that a user wants to know from which neighborhoods in the city represented in Figure 5.1 she can reach cinemas via public transportation. These neighborhoods can be retrieved using a *path query* defined by the following *regular expression*:

$$q = (tram + bus)^* \cdot cinema$$

The query q selects the nodes N_1, N_2, N_4, and N_6 as they are entailed by the following *paths* in the graph:

$$N_1 \xrightarrow{tram} N_4 \xrightarrow{cinema} C_1,$$
$$N_2 \xrightarrow{bus} N_1 \xrightarrow{tram} N_4 \xrightarrow{cinema} C_1,$$
$$N_4 \xrightarrow{cinema} C_1,$$
$$N_6 \xrightarrow{cinema} C_2.$$

Although very expressive, graph query languages are difficult to understand by non-expert users who are unable to specify their queries with a formal syntax. Similarly to the previous chapter, we address the problem of *assisting non-expert users to specify their queries* and we argue that it becomes even more difficult to tackle for graph databases. Indeed, graph databases usually do not carry proper metadata as they lack schemas and/or do not exhibit a clear distinction between instances and schemas. The absence of metadata

along with the difficulty of visualizing possibly large graphs make unfeasible traditional query specification paradigms for non-expert users, such as query by example [Zlo75]. Our work follows the recent trend of specifying graph queries by example [MLVP14, JKL+14]. Precisely, we focus on graph queries using regular expressions, which are fundamental building blocks of graph query languages [Bar13, Woo12], while both [MLVP14, JKL+14] consider simple graph patterns.

While the problem of executing path queries defined by regular expressions on graphs has been extensively studied recently [BBG13, LM13, KL12], no research has been done on how to actually specify such queries. Our work focuses on the problem of assisting non-expert users to specify such path queries, by exploiting elementary user input.

By continuing on our running example, we assume that the user is not familiar with any formal syntax of query languages, while she still wants to specify the above query q on the graph database in Figure 5.1 by providing examples of the query result. In particular, she would positively or negatively label some graph nodes according to whether or not they would be selected by the targeted query. Thus, let us imagine that the user labels the nodes N_2 and N_6 as *positive examples* because she wants these nodes as part of the result. Indeed, one can reach cinemas from N_2 and N_6, respectively, through the following paths:

$$N_2 \xrightarrow{bus} N_1 \xrightarrow{tram} N_4 \xrightarrow{cinema} C_1,$$
$$N_6 \xrightarrow{cinema} C_2.$$

Similarly, the user labels the node N_5 as a *negative example* since she would not like it as part of the query result. Indeed, there is no path starting in N_5 through which the user can reach a cinema. We also observe that the query q above is *consistent* with the user's examples because q selects all positive examples and none of the negative ones. Unfortunately, there may exist an infinite number of queries consistent with the given examples. Therefore, we are interested to find either the "exact" query that the user has in mind or, alternatively, an equivalent query, which is close enough to the user's expectations.

Apart from assisting unfamiliar users to specify queries, our research has other crucial applications, such as mining *scientific workflows*. Regular expressions have already been used in the literature as a well-suited mechanism for inter-workflow coordination [Hei01]. The path queries that we study can be applied to assist scientists in identifying interrelated workflows that are of interest for them. For instance, assume that a biologist is interested in retrieving all interrelated workflows having a pattern that starts with protein

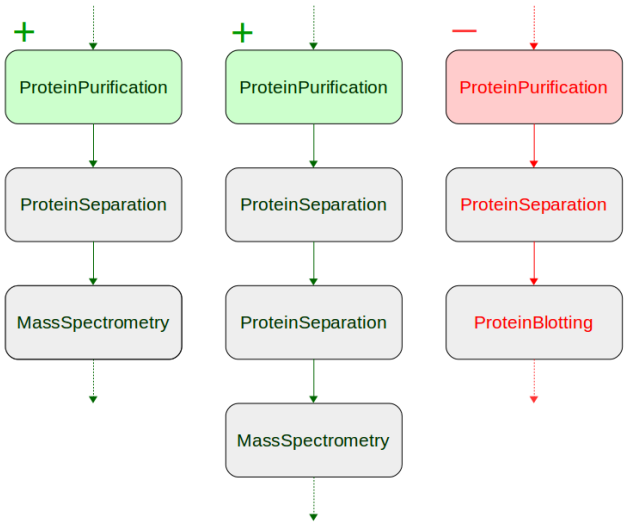


Figure 5.2: A set of scientific workflows examples.

purification, continues with an arbitrary number of protein separation steps, and ends with mass spectrometry. This corresponds to the following regular expression:

$$ProteinPurification \cdot ProteinSeparation^* \cdot MassSpectrometry.$$

Instead of specifying such a pattern in a formal language, the biologist may be willing to label some sequences of modules from a set of available workflows as positive or negative examples, as illustrated in Figure 5.2. Our algorithms can be thus applied to infer the workflow pattern that the biologist has in mind. Typically in graphs representing workflows the labels are attached to the nodes (e.g., as in Figure 5.2) instead of the edges. In our work, we have opted for edge-labeled graphs rather than node-labeled graphs, but our algorithms and learning techniques for the considered class of queries are applicable to the latter class of graphs in a seamless fashion. The problem of mining scientific workflows has been considered in recent work [Bel14], which leverages data instances as representatives of the input and output of a workflow module. In our approach, we rely on simpler user feedback, namely Boolean labeling of sequences of modules across interrelated workflows.

Since our goal is to infer the user queries while minimizing the amount of user feedback, our research is also applicable to *crowdsourcing* scenarios [FKK+11], in which such minimization typically entails lower financial costs. Indeed, we can imagine that crowdworkers provide the set of positive and negative examples mentioned above for path query learning. Moreover,

our work can be used in assisting non-expert users in other fairly complex tasks, such as specifying *schema mappings* [YMHF01] i.e., logical assertions between two path queries, one on a source schema and another on a target schema.

To the best of our knowledge, our work is the first to study the problem of learning path queries defined by regular expressions on graphs via user examples. More precisely, we make the following main contributions:

- We investigate a learning framework inspired by computational learning theory [KV94], in particular by grammatical inference [dlH10] and we identify fundamental difficulties of such a framework. We consequently propose a definition of learnability adapted to our setting.
- We propose a learning algorithm and we precisely characterize the conditions that a graph must satisfy to guarantee that every user's goal query can be learned. Essentially, the main theoretical result of the chapter states that for every query q there exists a polynomial set of examples that given as input to our learning algorithm guarantees the learnability of q. Additionally, our learning algorithm is guaranteed to run in polynomial time, whether or not the aforementioned set of examples is given as input.
- We investigate an interactive scenario, which bootstraps with an empty set of examples and builds it along the way. Indeed, the learning algorithm finely interacts with the user by proposing nodes that can be labeled and repeats the interactions until the goal query is learned. More precisely, we analyze what it means for a node to be informative for the learning process, we show the intractability of deciding whether a node is informative or not, and we propose efficient strategies to present examples to the user.
- To evaluate our approach, we have run experiments on both real-world and synthetic datasets. Our study shows the effectiveness of the learning algorithm and the advantage of using an interactive strategy, which significantly reduces the number of examples needed to learn the goal query.

It is important to point out that we have published this chapter as a conference paper [BCL15b]. Additionally, we have included in this chapter the proofs that have been omitted in [BCL15b] due to space restrictions. We also point out that we have employed the aforementioned interactive scenario as the core of a system for interactive path query specification on graph databases [BCL15a].

Finally, we would like to spend a few words on the class of queries that we investigate in this chapter. As already mentioned, we focus on regular expressions, which are fundamental for graph query languages [Bar13, Woo12] and lately used in the definition of SPARQL property paths[1]. Graph queries defined by regular expressions have been known as *regular path queries*. Intuitively, such queries retrieve pairs of nodes in the graph s.t. one can navigate between them with a path in the language of a given regular expression [Bar13, Woo12]. Although the usual semantics of regular path queries is *binary* (i.e., selects pairs of nodes), in this chapter we consider a generalization of this semantics that we call *monadic*, as it outputs only the originated nodes of the paths. The motivation behind using a monadic semantics is essentially threefold. First, it entails a larger space of potential solutions than a binary semantics. Indeed, with the latter semantics the end node of a path is fixed, which basically corresponds to have a smaller number of candidate paths that start at the originated node and that can be possibly labeled by the user. Second, in our learning framework, the amount of user effort should be kept as minimal as possible (which led to design an interactive scenario) and thus we let the user focus solely on the originated nodes of the paths rather than on pairs of nodes. Third, the development of the learning algorithm for monadic queries is extensible to binary queries and n-ary queries in a straightforward fashion, as we point out later on in the chapter.

Organization. In Section 5.2, we introduce some basic notions. In Section 5.3, we define our framework for learning from a set of examples, we present our learning algorithm, and we prove our learnability results. In Section 5.4, we propose an interactive algorithm and characterize the quantity of information of a node. In Section 5.5, we experimentally evaluate the performance of our algorithms. In Section 5.6, we discuss related work.

5.2 Graph databases and queries

In this section we define the concepts that we manipulate throughout the chapter.

Alphabet and words. An *alphabet* Σ is a finite, ordered set of symbols. A *word* over Σ is a sequence $a_1 \dots a_n$ of symbols from Σ. By $|w|$ we denote the *length* of a word w. The *concatenation* of two words $w_1 = a_1 \dots a_n$ and $w_2 = b_1 \dots b_m$, denoted $w_1 \cdot w_2$, is the word $a_1 \dots a_n b_1 \dots b_m$. By ε we denote

[1] http://www.w3.org/TR/sparql11-query/

the *empty word*. A *language* is a set of words. By Σ^* we denote the language of all words over Σ. We extend the order on Σ to the standard lexicographical order $\leqslant_{lex}$ on words over Σ and define a well-founded *canonical order* $\leqslant$ on words: $w \leqslant u$ iff $|w| < |u|$ or $|w| = |u|$ and $w \leqslant_{lex} u$.

Graph databases. A graph database is a finite, directed, edge-labeled graph [Bar13, Woo12]. Formally, a *graph* (*database*) G over an alphabet Σ is a pair (V, E), where V is a set of *nodes* and $E \subseteq V \times \Sigma \times V$ is a set of *edges*. Each edge in G is a triple $(\nu_o, a, \nu_e) \in V \times \Sigma \times V$, where ν_o is the *origin* of the edge, ν_e is the *end* of the edge, and a is the *label* of the edge. We often abuse notation and write $\nu \in G$ and $(\nu_o, a, \nu_e) \in G$ instead of $\nu \in V$ and $(\nu_o, a, \nu_e) \in E$, respectively. For example, take in Figure 5.3 the graph G_0 containing 7 nodes and 15 edges over the alphabet $\{a, b, c\}$.

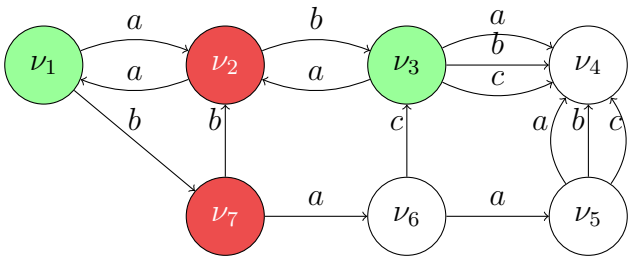


Figure 5.3: A graph database G_0.

Paths. A word $w = a_1 \dots a_n$ *matches* a sequence of nodes $\nu_0\nu_1 \dots \nu_n$ if, for each $1 \leqslant i \leqslant n$, the triple (ν_{i-1}, a_i, ν_i) is an edge in G. For example, for the graph G_0 in Figure 5.3, the word aba matches the sequences of nodes $\nu_1\nu_2\nu_3\nu_4$ and $\nu_3\nu_2\nu_3\nu_4$, respectively, but does not match the sequence $\nu_1\nu_2\nu_7\nu_2$. Note that the empty word ε matches the sequence $\nu\nu$ for every $\nu \in G$. Given a node $\nu \in G$, by $paths_G(\nu)$ we denote the language of all words that match a sequence of nodes from G that starts by ν. In the sequel, we refer to such words as *paths*, and moreover, we say that a path w is *covered* by a node ν if $w \in paths_G(\nu)$. Paths are ordered using the canonical order $\leqslant$. For example, for the graph G_0 in Figure 5.3 we have $paths_{G_0}(\nu_5) = \{\varepsilon, a, b, c\}$. Note that $\varepsilon \in paths_G(\nu)$ for every $\nu \in G$. Moreover, note that $paths_G(\nu)$ is finite iff there is no cycle reachable from ν in G. For example, for the graph G_0 in Figure 5.3, $paths_{G_0}(\nu_1)$ is infinite. We naturally extend the notion of paths to a set of nodes i.e., given a set of nodes X from a graph G, by $paths_G(X) = \bigcup_{\nu \in X} paths_G(\nu)$.

Regular expressions and automata A *regular language* is a language defined by a *regular expression* i.e., an expression of the following grammar:

$$q := \varepsilon \mid a \quad (a \in \Sigma) \mid q_1 + q_2 \mid q_1 \cdot q_2 \mid q^*,$$

where by "$\cdot$" we denote the *concatenation*, by "$+$" we denote the *disjunction*, and by "$*$" we denote the *Kleene star*. By $L(q)$ we denote the *language* of q, defined in the natural way [HU79]. For instance, the language of $(a \cdot b)^* \cdot c$ contains words like $c, abc, ababc$, etc.

Regular languages can alternatively be represented by automata [HU79]. A *nondeterministic finite word automaton* (*NFA*) A is a tuple $(Q, \Sigma, \delta, I, F)$, where Q is a finite set of states, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states, and $\delta : Q \times \Sigma \to 2^Q$ is the transition function. A *run* of A on a word $w = a_1 \dots a_n$ is a sequence of states $q_0 q_1 \dots q_n$ such that $q_i \in \delta(q_{i-1}, a_i)$ for $1 \leqslant i \leqslant n$ and $q_0 \in I$. A run on w is *accepting* if $q_n \in F$. A word w is *accepted* by A if there is an accepting run of A on w. By $L(A)$ we denote the set of words accepted by A. A *deterministic finite word automaton* (*DFA*) A is a NFA $(Q, \Sigma, \delta, I, F)$ such that I is a singleton, and $\delta(q, a)$ is either a singleton or the empty set (for each $q \in Q$ and $a \in \Sigma$).

In particular, we represent every regular language by its *canonical DFA* that is the unique smallest DFA that describes the language. For example, we present in Figure 5.4 the canonical DFA for $(a \cdot b)^* c$.

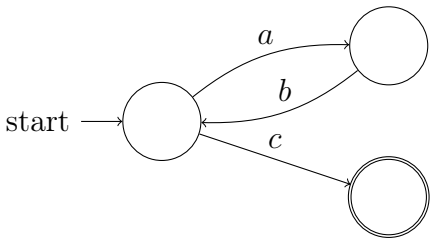


Figure 5.4: Canonical DFA for $(a \cdot b)^* c$.

Path queries. We focus on the class of path queries defined by regular expressions i.e., that select nodes having at least one *path* in the language of a given regular expression. Formally, given a graph G and a query q, we define the set of nodes *selected* by q on G:

$$q(G) = \{\nu \in G \mid L(q) \cap paths_G(\nu) \neq \varnothing\}.$$

For example, given the graph G_0 in Figure 5.3, the query a selects all nodes except ν_4, the query $(a \cdot b)^* \cdot c$ selects the nodes ν_1 and ν_3, and the query

$b \cdot b \cdot c \cdot c$ selects no node. In the rest of the chapter, we denote the set of all path queries by PQ and we refer to them simply as *queries*. We represent a query by its canonical DFA, hence the *size* of a query is the number of states in the canonical DFA of the corresponding regular language. For example, the size of the query $(a \cdot b)^* \cdot c$ is 3 (cf. Figure 5.4).

Equivalent queries. Two queries q and q' are *equivalent* if for every graph G they select exactly the same set of nodes i.e., $q(G) = q'(G)$. For example, the queries a and $a \cdot b^*$ are equivalent since each node having a path $ab \dots b$ has also a path a. This example can be easily generalized and yields to defining the class of prefix-free queries. Formally, we say that a query q is *prefix-free* if for every word from $L(q)$, none of its prefixes belongs to $L(q)$. Given a query q, there exists a unique prefix-free query equivalent to q, which, moreover, can be constructed by simply removing all outgoing transitions of every final state in the canonical DFA of q. Our interest in prefix-free queries is that they can be seen as *minimal representatives* of *equivalence classes* of queries and as such they are desirable queries for learning. Indeed, every prefix-free query q is in fact equivalent to an infinite number of queries $q \cdot (q' + \varepsilon)$, where q' can be every PQ. In the remainder, we assume w.l.o.g. that all queries that we manipulate are prefix-free.

5.3 Learning from a set of examples

The input of a learning algorithm consists of a graph on which the user has annotated a few nodes as *positive* or *negative examples*, depending on whether or not she would like the nodes as part of the query result. Our goal is to exploit such examples to find a query that satisfies the user. In this chapter, we explore two learning protocols: (i) the user provides a *sample* (i.e., a set of examples) that remains *fixed* during the learning process, and (ii) the learning algorithm *interactively* asks the user to label more examples until the learned query behaves exactly as the user wants.

First, we concentrate on the case of a fixed set of examples. We identify the challenges of such an approach, we show the unfeasability of the standard framework of *language identification in the limit* [Gol78] and slightly modify it to propose a learning framework with *abstain* (Section 5.3.1). Next, we present a learning algorithm for the class of PQ (Section 5.3.2) and we identify the conditions that a graph and a sample must satisfy to allow polynomial learning of the user's goal query (Section 5.3.3). We also point out that the learnability results can be easily extended to different semantics of path

queries, such as binary and n-ary (Section 5.3.4). We study the case of query learning from user interactions in Section 5.4.

5.3.1 Learning framework

Given a graph $G = (V, E)$, an *example* is a pair (ν, α), where $\nu \in V$ and $\alpha \in \{+, -\}$. We say that an example of the form $(\nu, +)$ is a *positive example* while an example of the form $(\nu, -)$ is a *negative example*. A *sample* S is a set of examples i.e., a subset of $V \times \{+, -\}$. Given a sample S, we denote the set of positive examples $\{\nu \in V \mid (\nu, +) \in S\}$ by S_+ and the set of negative examples $\{\nu \in V \mid (\nu, -) \in S\}$ by S_-. A sample is *consistent* (with the class of PQ) if there exists a (PQ) query that selects all positive examples and none of the negative ones. Formally, given a graph G and a sample S, we say that S is *consistent* if there exists a query q s.t. $S_+ \subseteq q(G)$ and $S_- \cap q(G) = \varnothing$. In this case we say that q is *consistent with* S. For instance, take the graph G_0 in Figure 5.3 and the sample S s.t. $S_+ = \{\nu_1, \nu_3\}$ and $S_- = \{\nu_2, \nu_7\}$; S is consistent because there exist queries like $(a \cdot b)^* \cdot c$ or $c + (a \cdot b \cdot c)$ that are consistent with S.

Next, we want to formalize what means for a class of queries to be *learnable*, as it is usually done in the context of *grammatical inference* [dlH10]. The standard learning framework is *language identification the limit* (*in polynomial time and data*) [Gol78], which requires a learning algorithm to operate in time *polynomial* in the size of its input, to be *sound* (i.e., always return a query consistent with the examples given by the user or a special *null* value if no such query exists), and *complete* (i.e., able to produce every query with a sufficiently rich set of examples).

Since we aim at a polynomial time algorithm that returns a query consistent with a sample, we must first investigate the *consistency checking* problem i.e., deciding whether such a query exists. To this purpose, we first identify a necessary and sufficient condition for a sample to be consistent.

Lemma 5.3.1 *Given a graph G and a sample S, S is consistent iff for every $\nu \in S_+$ it holds that $paths_G(\nu) \not\subseteq paths_G(S_-)$.*

Proof For the *if* part, for $1 \leqslant i \leqslant |S_+|$, for ν_i in S_+, let p_i be the path witnessing $paths_G(\nu_i) \not\subseteq paths_G(S_-)$. Then, the query $p_1 + \ldots + p_{|S_+|}$ is consistent with S.

The *only if* part follows directly from the semantics of queries and the definition of consistency. □

Next, we derive that the fundamental problem of consistency checking is PSPACE-complete.

Lemma 5.3.2 *Given a graph G and a sample S, deciding whether S is consistent is PSPACE-complete.*

Proof The membership to PSPACE follows from Lemma 5.3.1 and the known result that deciding the inclusion of NFAs is PSPACE-complete [SM73].

Next, we show the PSPACE-hardness by reduction from the universality of the union problem for DFAs, known as PSPACE-complete [Koz77], and by using a proof technique inspired by [ANS13]. The reduction works as follows. Take n DFAs $D_1, \ldots, D_n$ over Σ and two fresh symbols s_1, s_2 which are not in Σ. We construct a graph G as the disjoint union of $n+2$ graphs over $\Sigma \cup \{s_1, s_2\}$:

- For each DFA $D_i = (Q_i, \Sigma, \delta_i, I_i, F_i)$ (with $1 \leqslant i \leqslant n$), let $G_i = (V_i, E_i)$ s.t. $V_i = Q_i \cup \{\nu_i, \nu_i'\}$ and

  $$E_i = \{(\nu, a, \nu') \mid \delta(\nu, a) = \nu'\} \cup \{(\nu_i, s_1, q) \mid q \in I_i\} \cup \{(q, s_2, \nu_i') \mid q \in F_i\},$$

- Let $G_{n+1} = (V_{n+1}, E_{n+1})$ s.t. $V_{n+1} = \{\nu_{n+1}, u_1\}$ and $E_{n+1} = \{(\nu_{n+1}, s_1, u_1)\} \cup \{(u_1, a, u_1) \mid a \in \Sigma\}$.

- Let $G_{n+2} = (V_{n+2}, E_{n+2})$ s.t. $V_{n+2} = \{\nu_{n+2}, u_2, \nu'_{n+2}\}$ and $E_{n+2} = \{(\nu_{n+2}, s_1, u_2), (u_2, s_2, \nu'_{n+2})\} \cup \{(u_2, a, u_2) \mid a \in \Sigma\}$.

Then, take the sample S s.t. $S_+ = \{\nu_{n+2}\}$ and $S_- = \{\nu_1, \ldots, \nu_n, \nu_{n+1}\}$. We present in Figure 5.5 the graph and the sample that we construct by the described reduction.

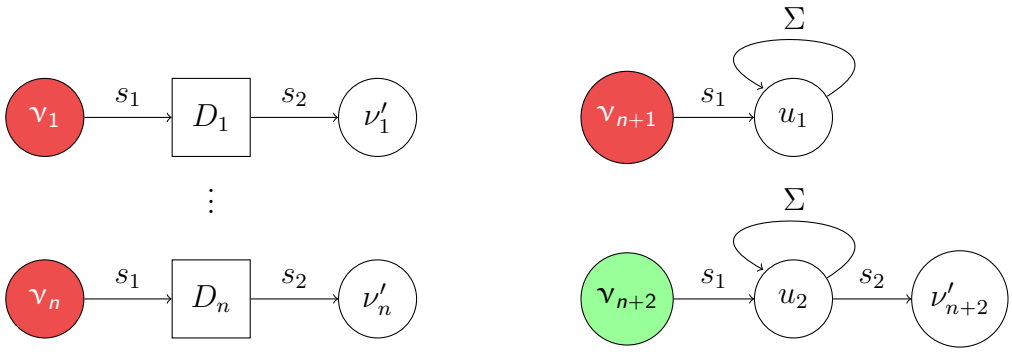


Figure 5.5: Constructed graph and sample.

We claim that S is consistent iff the union of the DFAs $D_1, \ldots, D_n$ is not universal i.e., $\bigcup_{i=1}^{n} L(D_i) \neq \Sigma^*$.

For the *if* part, let $w \in \Sigma^*$ be the word witnessing the non-universality of the DFAs $D_1, \ldots, D_n$. Then, take the query $q = s_1 \cdot w \cdot s_2$ and notice

that $\nu_{n+2} \in q(G)$. Since $w \notin L(D_i)$, we infer that $\nu_i \notin q(G)$ (for $1 \leqslant i \leqslant n$). Moreover, since w ends with an s_2 and $s_2 \notin \Sigma$, we infer that $\nu_{n+1} \notin q(G)$. We conclude that q is a witness of the consistency of S.

For the *only if* part, the consistency of S implies by Lemma 5.3.1 that there exists a path $p \in paths_G(\nu_{n+2})$ s.t. $p \notin paths_G(S_-)$. Since $p \in paths_G(\nu_{n+2})$ and $p \notin paths_G(\nu_{n+1})$, we infer that p is of the form $s_1 \cdot w \cdot s_2$, with $w \in \Sigma^*$. Since $paths_G(\{\nu_1, \ldots, \nu_n\})$ is the set of prefixes of all paths of the form $s_1 \cdot w' \cdot s_2$, where $w' \in L(D_1) \cup \ldots \cup L(D_n)$, we infer that $w \notin L(D_1) \cup \ldots \cup L(D_n)$, hence w is a witness of the non-universality of the DFAs $D_1, \ldots, D_n$.

Note that the described reduction works in polynomial time. ▫

This implies that an algorithm able to always answer *null* in polynomial time when the sample is inconsistent does not exist, hence our class of queries is not learnable in the classical framework. One solution could be to study less expressive classes of queries. However, as shown by the following Lemma, consistency checking remains intractable even for a very restricted class of queries, referred as "SORE(·)" in [ANS13].

Lemma 5.3.3 *Given a graph G and a sample S, deciding whether there exists a query of the form $a_1 \cdot \ldots \cdot a_n$ (pairwise distinct symbols) consistent with S is NP-complete.*

Proof To show the membership of the problem to NP, we point out that a non-deterministic Turing machine guesses a query q of the form $a_1 \cdot \ldots \cdot a_n$ with pairwise distinct symbols (hence of length bounded by $|\Sigma|$) and then checks whether q is consistent with S.

Next, we show the NP-hardness by reduction from 3SAT, known as NP-complete and by using a proof technique inspired by [ANS13]. The reduction works as follows. Take a 3CNF formula $\varphi = C_1 \wedge \ldots \wedge C_k$ over the variables $x_1, \ldots, x_n$. Take the alphabet $\Sigma = \{s_1, s_2, a_{11}, a_{12}, a_{13}, \ldots, a_{k1}, a_{k2}, a_{k3}, \}$. Note that for $1 \leqslant i \leqslant k$, the labels a_{i1}, a_{i2}, a_{i3} *correspond* to the literals from the clause C_i on the position 1, 2, and 3, respectively. Next, construct the graph G as the disjoint union of at most $n + 2$ graphs over Σ:

- Let $G_\varphi^+ = (V_\varphi^+, E_\varphi^+)$ be the graph that intuitively encodes the formula φ. Formally, $V_\varphi^+ = \{\nu_\varphi^+, u_1^+, \ldots, u_{k+1}^+, \nu_\varphi'^+\}$ and $E_\varphi^+ = \{(\nu_\varphi^+, s_1, u_1^+), (u_{k+1}^+, s_2, \nu_\varphi'^+)\} \cup \{(u_i^+, a_{ij}, u_{i+1}^+) \mid 1 \leqslant i \leqslant k, 1 \leqslant j \leqslant 3\}$,

- Let $G_\varphi^- = (V_\varphi^-, E_\varphi^-)$ be the graph that intuitively forces any possible consistent query to end with an s_2. Formally, $V_\varphi^- = \{\nu_\varphi^-, u_1^-, \ldots, u_{k+1}^-\}$ and $E_\varphi^- = \{(\nu_\varphi^-, s_1, u_1^-)\} \cup \{(u_i^-, a_{ij}, u_{i+1}^-) \mid 1 \leqslant i \leqslant k, 1 \leqslant j \leqslant 3\}$,

- For $1 \leqslant i \leqslant n$, if the variable x_i appears in both positive and negative literals in clauses of φ, construct G_i as the graph that intuitively encodes the fact that a variable x_i cannot be assigned both the value true and false. If a variable x_i appears only in positive or only in negative literals, then such graphs are not of interest. Before defining G_i formally, let $T_i = \{a_{jl} \mid$ the clause C_j has on position l the positive literal $x_i\}$ and $F_i = \{a_{jl} \mid$ the clause C_j has on position l the negative literal $\neg x_i\}$. Next, let $G_i = \{V_i, E_i\}$ s.t. $V_i = \{\nu_{i1}, \ldots, \nu_{i5}\}$ and

$$\begin{aligned} E_i = \ & \{(\nu_{i1}, s_1, \nu_{i2})\} \cup \{(\nu_{i2}, a, \nu_{i2}) \mid a \in \Sigma \backslash \{s_2\} \backslash T_i \backslash F_i\} \cup \{(\nu_{i5}, a, \nu_{i5}) \mid a \in \Sigma\} \\ & \cup \{(\nu_{i2}, a, \nu_{i3}) \mid a \in F_i\} \cup \{(\nu_{i3}, a, \nu_{i3}) \mid a \in \Sigma \backslash \{s_2\} \backslash T_i\} \cup \{(\nu_{i3}, a, \nu_{i5}) \mid a \in T_i\} \\ & \cup \{(\nu_{i2}, a, \nu_{i4}) \mid a \in T_i\} \cup \{(\nu_{i4}, a, \nu_{i4}) \mid a \in \Sigma \backslash \{s_2\} \backslash F_i\} \cup \{(\nu_{i4}, a, \nu_{i5}) \mid a \in F_i\}. \end{aligned}$$

Finally, take the sample S_φ s.t. $S_{\varphi_+} = \{\nu_\varphi^+\}$ and $S_{\varphi_-} = \{\nu_\varphi^-\} \cup \{\nu_{i1} \mid 1 \leqslant i \leqslant n$ and x_i appears in both positive and negative literals in $\varphi\}$.

For example, we present in Figure 5.6 the graph and the sample constructed for the formula $\varphi_0 = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$. Notice that x_1 is the only variable that appears in both positive and negative literals in φ_0 hence we have constructed its corresponding subgraph and labeled the node ν_{11} as a negative example.

We claim that there exists a query of the form $a_1 \cdot \ldots \cdot a_n$ (with pairwise distinct symbols) consistent with S iff $\varphi \in$ 3SAT.

For the *if* part, take the valuation $v : \{x_1, \ldots, x_n\} \to \{true, false\}$ that makes φ satisfiable. Then, for each clause C_i (for $1 \leqslant i \leqslant k$) let a_{il_i} (with $1 \leqslant l_i \leqslant 3$) be the label corresponding to a literal satisfied by the valuation v. Take the query $q = s_1 \cdot a_{1l_1} \cdot \ldots \cdot a_{kl_k} \cdot s_2$. Clearly $\nu_\varphi^+ \in q(G)$. Since q encodes a valuation, none of the existing ν_{i1} (with $1 \leqslant i \leqslant n$) is in $q(G)$. Moreover, since q ends with s_2, we infer that $\nu_\varphi^- \notin q(G)$. We conclude that q is a query consistent with S, and indeed, it has the form of a path with pairwise distinct symbols.

For the *only if* part, take the query q consistent with S. Since $\nu_\varphi^+ \in q(G)$ and $\nu_\varphi^- \notin q(G)$, we infer that q has the form $s_1 \cdot a_{1l_1} \cdot \ldots \cdot a_{kl_k} \cdot s_2$, where each a_{il_i} (for $1 \leqslant i \leqslant k$) corresponds to a literal from the clause C_i. Moreover, since none of the existing ν_{i1} (with $1 \leqslant i \leqslant n$) is in $q(G)$, we infer that the path w encodes a valuation $v : \{x_1, \ldots, x_n\} \to \{true, false\}$. Since $\nu_\varphi^+ \in q(G)$, we conclude that the valuation v makes φ satisfiable.

Note that the described reduction works in polynomial time. □

The proofs of Lemma 5.3.2 and 5.3.3 rely on techniques inspired by the definability problem for graph query languages [ANS13]. We also point out

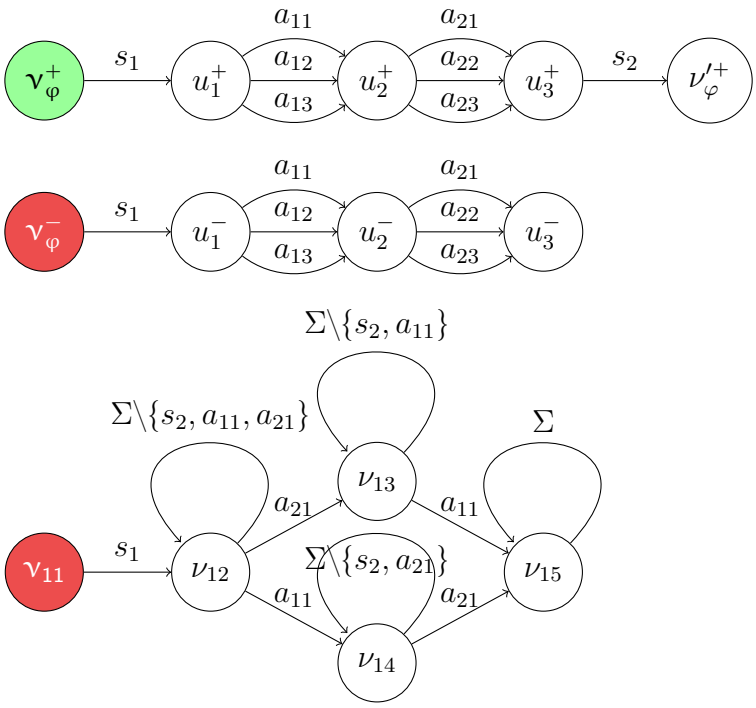


Figure 5.6: Graph and sample constructed for the formula $\varphi_0 = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$.

that the same intractability results for consistency checking hold for binary semantics.

Another way to overcome the intractability of our class of queries is to relax the soundness condition and adopt a learning framework with *abstain*, similarly to what has been recently done for learning XML transformations [LLN+14]. More precisely, we allow the learning algorithm to answer a special value *null* whenever it cannot efficiently construct a consistent query. In practice, the *null* value is interpreted as "not enough examples have been provided". However, the learning algorithm should always return in polynomial time either a consistent query or *null*. As an additional clause, we require a learning algorithm to be *complete* i.e., when the input sample contains a *polynomially-sized characteristic sample* [dlH10, Gol78], the algorithm must return the goal query. More formally, we have the following.

Definition 5.3.4 *A class of queries $\mathcal{Q}$ is* learnable with abstain in polynomial time and data *if there exists a polynomial learning algorithm learner that is:*

1. **Sound with abstain.** *For every graph G and sample S over G, the*

algorithm learner(G, S) *returns either a query in* $\mathcal{Q}$ *that is consistent with* S*, or null if no such query exists or it cannot be constructed efficiently.*

2. **Complete.** *For every query* $q \in \mathcal{Q}$*, there exists a graph* G *and a polynomially-sized characteristic sample* CS *on* G *s.t. for every sample* S *extending* CS *consistently with* q *(i.e.,* $CS \subseteq S$ *and* q *is consistent with* S*), the algorithm learner*(G, S) *returns* q*.*

Note that the polynomiality depends on the choice of a representation for queries and recall that we represent each PQ with its canonical DFA. Next, we present a polynomial learning algorithm fulfilling the two aforementioned conditions and we point out the construction of a polynomial characteristic sample to show the learnability of PQ.

5.3.2 Learning algorithm

In a nutshell, the idea behind our learning algorithm is the following: for each positive node, we seek the path that the user followed to label such a node, then we construct the disjunction of the paths obtained in the previous step, and we end by generalizing this disjunction while remaining consistent with both positive and negative examples.

More formally, the algorithm consists of two complementary steps that we describe next: *selecting the smallest consistent paths* and *generalizing* them.

Selecting the smallest consistent paths (SCPs). Since the labeled nodes in a graph may be the origin of multiple paths, classical algorithms for learning regular expressions from words, such as RPNI [OG92], are not directly applicable to our setting. Indeed, we have a set of nodes in the graph from which we have to first select (from a potentially infinite set) the paths responsible for their selection. Therefore, the first challenge of our algorithm is to select for each positive node a path that is not covered by any negative. We call such a path a *consistent path*. One can select consistent paths by simply enumerating (according to the canonical order $\leqslant$) the paths of each node labeled as positive and stopping when a consistent path for each node is found. We refer to the obtained set of paths as the set of *smallest consistent paths* (SCPs) because they are the smallest (w.r.t. $\leqslant$) consistent paths for each node. As an example, for the graph G_0 in Figure 5.3 and a sample s.t. $S_+ = \{\nu_1, \nu_3\}$ and $S_- = \{\nu_2, \nu_7\}$, we obtain the SCPs abc and c for ν_1 and ν_3, respectively. Notice that in this case the disjunction of the SCPs (i.e., the query $c + (a \cdot b \cdot c)$) is consistent with the input sample and one may think that

a learning algorithm should return such a query. The shortcoming of such an approach is that the learned query would be always very simple in the sense that it uses only concatenation and disjunction. Since we want a learning algorithm that covers all the expressibility of PQ (in particular including the Kleene star), we need to extend the algorithm with a further step, namely the generalization. We detail such a step at the end of this section.

Another problem is that the user may provide an inconsistent sample, by labeling a positive node having no consistent path (cf. Lemma 5.3.1). To clarify when such a situation occurs, consider a simple graph such as the one in Figure 5.7 having one positive node (labeled with +) and two negative ones (labeled with −). We observe that the positive node has an infinite number

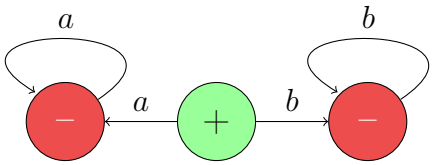


Figure 5.7: A graph with an inconsistent sample.

of paths. However, all of them are covered by the two negative nodes, thus yielding an inconsistent sample. This example also shows that the simple procedure described above (i.e., enumerate the paths of the positive nodes and stop when finding consistent ones) fails because it leads to enumerating an infinite number on paths and never halting. On the other hand, we cannot perform consistency checking before selecting the SCPs since this fundamental problem is intractable (cf. Lemma 5.3.2 and 5.3.3). As a result, to avoid this possibly infinite enumeration, we choose to fix the maximal length of a SCP to a bound k. This bound engenders a new issue: for a fixed k it may not be possible to detect SCPs for all positive nodes. For instance, assume a graph that the user labels consistently with the query $(a \cdot b)^* \cdot c$, in particular she labels three positive nodes for which the SCPs are c, abc, and $ababc$. For a fixed $k = 3$, the SCP $ababc$ is not detected and the disjunction of the first two SCPs (i.e., $c + (a \cdot b \cdot c)$) is not a query consistent with the sample.

For all these reasons, we introduce a generalization phase in the algorithm, which permits to solve the two mentioned shortcomings i.e., (i) to learn a query that covers all the expressibility of PQ, and (ii) to select all positive examples even though not all of them have SCPs shorter than k.

Generalizing SCPs. We have seen how to select, whenever possible, a SCP of length bounded by k for each positive example. Next, we show how

we can employ these SCPs to construct a more general query. The *learning algorithm* (Algorithm 11) takes as input a graph G and a sample S, and outputs a query q consistent with S whenever such query exists and can be built using SCPs of length bounded by k; otherwise, the algorithm outputs a special value *null*.

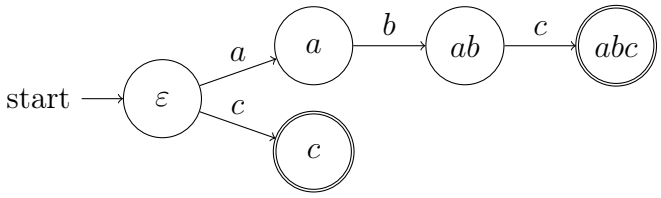
Algorithm 11 Learning algorithm – $learner(G, S)$.

Input: graph G, sample S
Output: query q consistent with S or *null*
Parameter: fixed $k \in \mathbb{N}$ //maximal length of a SCP

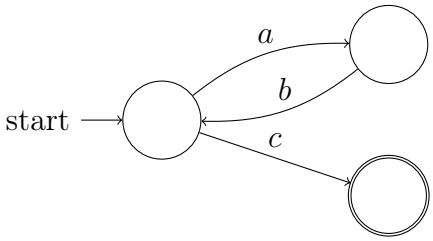
1: **for** $\nu \in S_+$. $\exists p \in \Sigma^{\leqslant k}$. $p \in paths_G(\nu) \backslash paths_G(S_-)$ **do**
2: $P := P \cup \{\min_{\leqslant}(paths_G(\nu) \backslash paths_G(S_-))\}$
3: **let** A be the prefix tree acceptor for P
4: **while** $\exists s, s' \in A$. $L(A_{s' \to s}) \cap paths_G(S_-) = \varnothing$ **do**
5: $A := A_{s' \to s}$
6: **if** $\forall \nu \in S_+$. $L(A) \cap paths_G(\nu) \neq \varnothing$ **then**
7: **return** query q represented by the DFA A
8: **return** *null*

We illustrate the algorithm on the graph G_0 in Figure 5.3 with a sample S s.t. $S_+ = \{\nu_1, \nu_3\}$ and $S_- = \{\nu_2, \nu_7\}$. For ease of exposition, we assume a fixed $k = 3$ (we explain how to obtain the value of k theoretically in Section 5.3.3 and empirically in Section 5.5). At first (lines 1-2), the algorithm constructs the set P of SCPs bounded by k for the positive nodes from which these paths in P can be constructed. Note that by $\Sigma^{\leqslant k}$ we denote the set of all paths of length at most k. For instance, on our example in Figure 5.3, we obtain $P = \{abc, c\}$. Then (line 3), we construct the *PTA* (*prefix tree acceptor*) [dlH10] of P, which is basically a tree-like DFA accepting only the paths in P and having as states all their prefixes. Figure 5.8(a) illustrates the obtained PTA A for our example. Then (lines 4-5), we *generalize* A by merging two of its states if the obtained DFA selects no negative node. Note that by $A_{s' \to s}$ we denote the DFA obtained from A by modifying each occurrence of the state s' in s. Recall that on our example we have $P = \{abc, c\}$ and the PTA A in Figure 5.8(a). Next, we try to merge states of A: the states ε and a cannot be merged (because the obtained DFA would select the path bc that is covered by the negative ν_2), the states ε and c cannot be merged (because the path ε is covered by both negatives), while the states ε and ab can be merged without covering any negative example. On our example, we obtain the DFA in Figure 5.8(b), where no further states can be merged. Finally, the algorithm checks whether the query represented by A selects all the positive examples (not only those from whose SCPs we have constructed A), and if

this is the case, it outputs the query (lines 6-7). In our case, the obtained $(a \cdot b)^* \cdot c$ selects all positive nodes hence is returned.



(a) Prefix tree acceptor.



(b) Result of generalization.

Figure 5.8: DFAs considered by *learner* for Figure 5.3.

5.3.3 Learnability results

Recall that in Section 5.3.1 we have formally defined what means for a class of queries to be learnable while in Section 5.3.2 we have proposed a learning algorithm for PQ. In this section, we prove that the proposed algorithm satisfies the conditions of Definition 5.3.4, thus showing the main theoretical result of the chapter. We conclude the section with practical observations related to our learnability result.

By $\text{PQ}^{\leq n}$ we denote the PQ of size at most n.

Theorem 5.3.5 *The query class* $\text{PQ}^{\leq n}$ *is learnable with abstain in polynomial time and data, using the algorithm learner with the parameter* k *set to* $2 \times n + 1$.

Proof First, we point out that *learner* works in *polynomial time* in the size of the input. Indeed, the number of paths to explore for each positive node (lines 1-2) is polynomial since the length of paths is bounded by a fixed k. Then, both testing the consistency of queries considered during the generalization (lines 3-5) and testing whether the computed query selects all positive nodes (lines 6-7) reduce to deciding the emptiness of the intersection of two NFAs, known as being in PTIME [LR92]. Moreover, *learner* is *sound*

with abstain since it returns either a query consistent with the input sample if it is possible to construct it using SCPs of length bounded by k, or *null* otherwise.

As for the *completeness* of *learner*, we show, for every $q \in \text{PQ}$, the construction of a graph G and of a characteristic sample CS on G with the properties from Definition 5.3.4 i.e., for every sample S that extends CS consistently with q (i.e., $CS \subseteq S$ and q is consistent with S), the algorithm $learner(G, S)$ returns q. The idea behind the construction is the following: we know that RPNI [OG92] is an algorithm that takes as input positive and negative word examples and outputs a DFA describing a consistent regular language, and moreover, the core of RPNI is based on DFA generalization via state merges similarly to *learner*; hence, for a query q, we want a graph and a sample on it s.t. when *learner* selects the SCPs, it should get exactly the words that RPNI needs for learning q if we see it as a regular language. Then, since RPNI is guaranteed to learn the goal regular language from these words, we infer that if *learner* selects and generalizes the SCPs corresponding to them, then *learner* is also guaranteed to learn the goal PQ.

We exemplify the construction on the query $q = (a \cdot b)^* \cdot c$. First, given q, we construct two sets of words P_+ and P_- that correspond to a characteristic sample used by RPNI to infer the regular language of q. In our case, we obtain $P_+ = \{c, abc\}$ and $P_- = \{\varepsilon, a, ab, ac, bc\}$. Then, the characteristic graph for learning the graph query q needs (i) for each $p \in P_+$, a node $\nu \in CS_+$ s.t. $p = \min_{\leqslant}(L(q) \cap paths_G(\nu))$, (ii) for each $p \in P_-$, a node $\nu \in CS_-$ s.t. $p \in paths_G(\nu)$, and (iii) for each p' that is smaller (w.r.t. the canonical order $\leqslant$) than a word $p \in P_+$ and is not prefixed by any word in $L(q)$, a node $\nu \in CS_-$ s.t. $p' \in paths_G(\nu)$. For our query $(a \cdot b)^*c$, (i) implies two positive nodes: a node ν s.t. $c \in paths_G(\nu)$ and another node ν' s.t. $abc \in paths_G(\nu')$ and $c \notin paths_G(\nu')$, while (ii) and (iii) imply a node ν'' s.t. $\nu'' \notin q(G)$ and $\{\varepsilon, a, ab, ac, bc\} \subseteq paths_G(\nu'')$ (cf. ii) and $\{\varepsilon, a, b, aa, ab, ac, ba, bb, bc, aaa, aab, aac, aba, abb\} \subseteq paths_G(\nu'')$ (cf. iii). In Figure 5.9 we illustrate such a graph and we highlight the two positive and one negative node examples.

Recall that the size of a query is the number of states of its canonical DFA. According to the above construction, we need $|CS_+| = |P_+|$ and $|CS_-| = 1$. Since $|P_+|$ is polynomial in the size of the query [OG92], we infer that $|CS|$ is also polynomial in the size of the query. Moreover, to learn the regular language of a query of size n, the longest path in P_+ is of size $2 \times n + 1$ [OG92]. Hence, to be able to select this path with *learner* (assuming the presence of a characteristic sample), we need the parameter k of *learner* to be at least $2 \times n + 1$. Thus, for each possible size n of the goal query there exists a polynomial learning algorithm satisfying the conditions of Definition 5.3.4, which concludes the proof. □

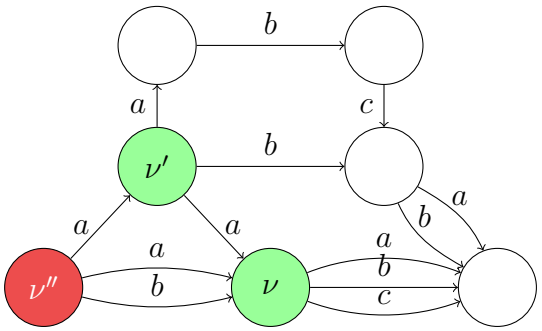


Figure 5.9: Graph from the proof of Theorem 5.3.5.

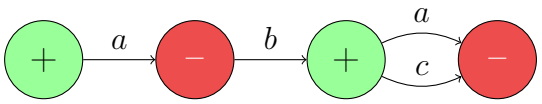


Figure 5.10: A graph and a sample for $(a \cdot b)^* \cdot c$.

We end this section with some practical observations related to our learnability result.

First, we point out that although Theorem 5.3.5 requires a certain theoretical value for k to guarantee learnability of queries of a certain size, our experiments indicate that small values of k (between 2 and 4) are enough to cover most practical cases. Then, even though the definition of learnability requires that one characteristic sample exists, in practice there may be many of such samples and there exists an infinite number of graphs on which we can build them. In fact, a graph that contains a subgraph with a characteristic sample is also characteristic.

Second, we also point out that a practical sample may be characteristic without having all negative paths on the same node as required by the aforementioned construction. For instance, the sample that we have used to illustrate the learning algorithm (i.e., the sample s.t. $S_+ = \{\nu_1, \nu_3\}$ and $S_- = \{\nu_2, \nu_7\}$ on the graph in Figure 5.3) is characteristic for $(a \cdot b)^* \cdot c$ and all above mentioned negatives paths are covered by two negative nodes.

Third, if a graph does not own a characteristic sample, the user's goal query on that graph cannot be exactly identified. In such a case, the learning algorithm returns a query equivalent to the goal query on the graph and hence *indistinguishable* by the user (i.e., they select exactly the same set of nodes). For instance, take the graph in Figure 5.10 and assume a user labeling the nodes w.r.t. the goal $(a \cdot b)^* \cdot c$. The learning algorithm returns the query a that selects exactly the same set of nodes on this graph.

5.3.4 Extensions to binary and n-ary semantics

In this section, we point out that the presented learning techniques are directly applicable to different query semantics such as binary and n-ary queries, which are outside the scope of this chapter. Recall that the main semantics that we investigate in this chapter is monadic i.e., we consider the class of queries that select the nodes having at least one path in the language of a given regular expression.

Intuitively, to learn a binary query, the only change w.r.t. the learning algorithm for monadic semantics (i.e., Algorithm 11) is that each positive example implies a smaller number of candidate paths from which we have to choose a consistent one (since the destination node is also known). Then, for the n-ary case (i.e., when an example is a tuple of nodes labeled with $+$ or $-$), we have simply to apply the previous algorithm to learn a query for each position in the tuple and then to combine those.

First, let us introduce some auxiliary notions. Given a graph $G = (V, E)$ and two nodes ν, ν' from V, by $paths^2_G(\nu, \nu')$ we denote the *set of all paths between the nodes ν and ν'*. Formally, a word $w = a_1 \dots a_n$ belongs to $paths^2_G(\nu, \nu')$ if there exists a sequence of nodes $\nu\nu_1 \dots \nu_{n-1}\nu'$ such that the triples (ν, a_1, ν_1), (ν_{i-1}, a_i, ν_i) (with $2 \leqslant i \leqslant n-1$), and (ν_{n-1}, a_n, ν') belong to E. We naturally extend the notion of paths between two nodes to a set of pairs of nodes $X \subseteq V \times V$ i.e., $paths^2_G(X) = \bigcup_{(\nu,\nu')\in X} paths^2_G(\nu, \nu')$.

An *n-ary path query* Q is a sequence of regular expressions $(q_1, \dots, q_{n-1})$. Given a graph $G = (V, E)$ and an n-ary path query $Q = (q_1, \dots, q_{n-1})$, the set of *tuples of nodes of G selected by Q*, denoted $Q(G)$ is as follows.

$$Q(G) = \{(\nu_1, \dots, \nu_n) \mid \forall 1 \leqslant i \leqslant n-1.\ paths^2_G(\nu_i, \nu_{i+1}) \cap L(q_i) \neq \varnothing\}.$$

By NPQ we denote the set of all n-ary path queries (or simply n-ary queries).

When $n = 2$, we have the particular case of binary semantics. In such a case, a query Q consists of a single regular expression q and selects the pairs of nodes of the graph that are linked via a path in the language of q. An example for learning is now a pair of nodes (ν, ν') labeled with $+$ or $-$. To learn a binary query, the only change to Algorithm 11 (cf. Section 5.3.2) is that each positive example implies a smaller set of candidate paths from which we have to choose a consistent one (since the destination node is also known). Algorithm 12 illustrates this idea.

For the n-ary case, an example is $((\nu_1, \dots, \nu_n), \alpha)$ where $\alpha \in \{+, -\}$ and $(\nu_1, \dots, \nu_n)$ is a tuple of nodes of the graph. We have simply to apply the previous algorithm to learn a query for each position in the tuple and then to combine those. Algorithm 13 illustrates this idea.

Algorithm 12 Learning algorithm for binary semantics – $learner^2(G, S)$.

Input: graph G, sample S
Output: query Q consistent with S or *null*
Parameter: fixed $k \in \mathbb{N}$ //maximal length of a SCP

for $(\nu, \nu') \in S_+$. $\exists p \in \Sigma^{\leqslant k}$. $p \in paths^2_G(\nu, \nu') \backslash paths^2_G(S_-)$ **do**
    $P := P \cup \{\min_{\leqslant}(paths^2_G(\nu, \nu') \backslash paths^2_G(S_-))\}$
let A be the prefix tree acceptor for P
while $\exists s, s' \in A$. $L(A_{s' \to s}) \cap paths^2_G(S_-) = \varnothing$ **do**
    $A := A_{s' \to s}$
if $\forall (\nu, \nu') \in S_+$. $L(A) \cap paths^2_G(\nu, \nu') \neq \varnothing$ **then**
    return query $Q = (q)$ where q is represented by the DFA A
return *null*

Algorithm 13 Learning algorithm for n-ary semantics – $learner^n(G, S)$.

Input: graph G, sample S
Output: query Q consistent with S or *null*
Parameter: fixed $k \in \mathbb{N}$ //maximal length of a SCP

for $i \in \{1, \ldots, n-1\}$ **do**
    let $S^i_+ = \{(\nu_i, \nu_{i+1}) \mid \forall (\nu_1, \ldots, \nu_n) \in S_+)\}$
    let $S^i_- = \{(\nu_i, \nu_{i+1}) \mid \forall (\nu_1, \ldots, \nu_n) \in S_-)\}$
    let $q_i = learner^2(G, S^i)$
    if q_i is *null* **then**
        return *null*
return query $Q = (q_1, \ldots, q_{n-1})$

Let NPQ$^{\leqslant s}$ be the subset of NPQ consisting of the n-ary path queries Q s.t. the maximal size of a regular language in Q is s (recall that the size is the number of states in its canonical DFA). Finally, using Algorithm 13 and the same proof technique as for the learnability of (monadic) path queries (i.e., Theorem 5.3.5), we can state the following result.

Corollary 5.3.6 *The query class* NPQ$^{\leqslant s}$ *is learnable with abstain in polynomial time and data, using the algorithm learner*^{n} *with the parameter k set to* $2 \times s + 1$.

5.4 Learning from user interactions

In this section, we investigate the problem of query learning from a different perspective. In Section 5.3 we have studied the setting of a fixed set of examples provided by the user and no interaction with her during the learning process. In this section, we consider an *interactive scenario* where the learning algorithm starts with an empty sample and continuously interacts with the user and asks her to label additional nodes until she is satisfied with the output of the learned query. To this purpose, we first propose the *interactive scenario* (Section 5.4.1). Then, we discuss different parameters for it, in particular what means for a node to be *informative* and what is a *practical strategy* of proposing nodes to the user (Section 5.4.2).

5.4.1 Interactive scenario

We consider the following *interactive scenario*. The user is presented with a node of the graph and indicates whether the node is selected or not by the query that she has in mind. We repeat this process until a sufficient knowledge of the goal query has been accumulated (i.e., there exists at most one query consistent with the user's labels). This scenario is inspired by the well-known framework of *learning with membership queries* [Ang88] In Figure 5.11, we describe the current instantiation for path queries on graphs and we detail next its different steps.

① ② We consider as input a graph database G. Initially, we assume an empty sample that we enrich via simple interactions with the user. The interactions continue until a *halt condition* is satisfied. A natural halt condition is to stop the interactions when there is exactly one consistent query with the current sample. In practice, we can imagine weaker conditions e.g., the user may stop the process earlier if she is satisfied by some candidate query proposed at some intermediary stage during the interactions.

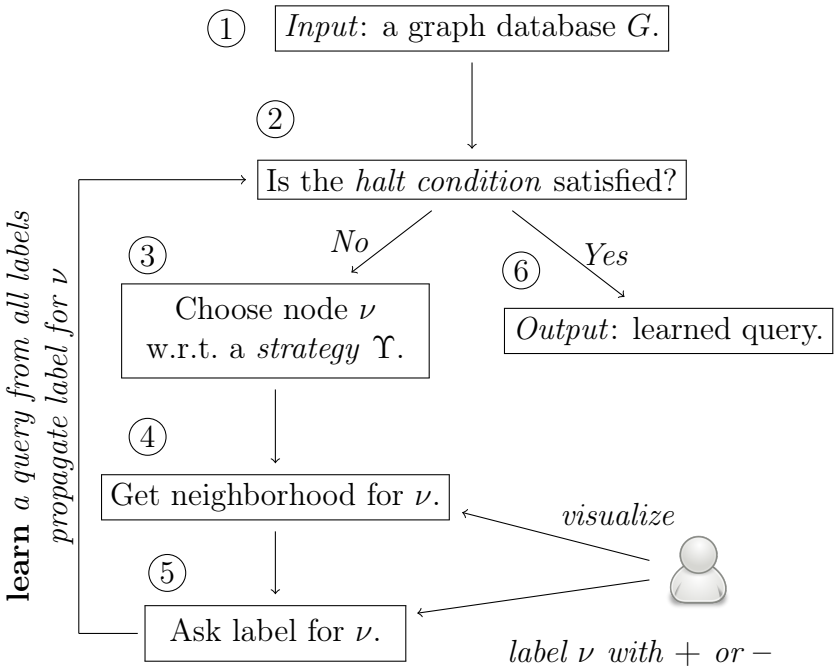


Figure 5.11: Interactive scenario.

③ We propose nodes of the graph to the user according to a *strategy* Υ i.e., a function that takes as input a graph G and a sample S, and returns a node from G. Since our goal is to minimize the amount of effort needed to learn the user's goal query, a smart strategy should avoid proposing to the user nodes that do not bring any information to the learning process. The study of such strategies yields to defining the notion of *informative nodes* that we formalize in the next section.

④ A node by itself does not carry enough information to allow the user to understand whether it is part of the query result or not. Therefore, we have to enhance the information of a node by zooming out on its *neighborhood* before actually showing it to the user. This step has the goal of producing a small, easy to visualize fragment of the initial graph, which permits the user to label the proposed node as a positive or a negative example. More concretely, in a practical scenario, all nodes situated at a distance k (as the parameter of Algorithm 11 explained in Section 5.3.2) should be sufficient for the user to decide whether she wants or not the proposed node. In any case, the user has neither to visualize all the graph that can be potentially large, nor to look by herself for interesting nodes because our interactive scenario proposes such nodes to the user.

⑤ ⑥ The user visualizes the neighborhood of a given node ν and labels ν w.r.t. the goal query that she has in mind. Then, we propagate the given label in the rest of the graph and prune the nodes that become uninformative. Moreover, we run the learning algorithm *learner* (i.e., Algorithm 11 from Section 5.3.2), which outputs in polynomial time either a query consistent with all labels provided by the user, or *null* if such a query does not exist or cannot be constructed efficiently. When the halt condition is satisfied, we return the latest output of *learner* to the user. In particular, the halt condition may take into account such an intermediary learned query q e.g., when the user is satisfied by the output of q on the instance and wants to stop the interactions.

In the next section, we precisely describe what means for a node to be informative for the learning process and what is a practical strategy of proposing nodes to the user.

5.4.2 Informative nodes and practical strategies

Before explaining the informative nodes, we first define the set of all queries consistent with a sample S over a graph G:

$$\mathcal{C}(G, S) = \{q \in \text{PQ} \mid S_+ \subseteq q(G) \wedge S_- \cap q(G) = \varnothing\}.$$

The set $\mathcal{C}(G, S)$ is not empty if the user has labeled the examples in S consistently with some goal query that she has in mind. Recall that we have defined a similar notion in the previous chapter for the inference of relational joins. Moreover, notice that for a consistent sample S and an unlabeled node ν from G, the two possible labels of ν split $\mathcal{C}(G, S)$ in two disjoint sets. Formally, we have the following.

Lemma 5.4.1 *Given a graph G, a consistent sample S over G, and an unlabeled node ν from G, it holds that*

$$\mathcal{C}(G, S \cup \{(\nu, +)\}) \cup \mathcal{C}(G, S \cup \{(\nu, -)\}) = \mathcal{C}(G, S) \text{ and}$$
$$\mathcal{C}(G, S \cup \{(\nu, +)\}) \cap \mathcal{C}(G, S \cup \{(\nu, -)\}) = \varnothing.$$

Proof The subset $\mathcal{C}(G, S \cup \{(\nu, +)\}) \subseteq \mathcal{C}(G, S)$ is the set of queries in $\mathcal{C}(G, S)$ that select ν while the subset $\mathcal{C}(G, S \cup \{(\nu, -)\}) \subseteq \mathcal{C}(G, S)$ is the set of queries in $\mathcal{C}(G, S)$ that do not select ν. Notice that the intersection of the two subsets is empty and their union is $\mathcal{C}(G, S)$. ▫

Assuming that the user labels the nodes consistently with some goal query q, the set $\mathcal{C}(G, S)$ always contains q. Initially, $S = \varnothing$ and $\mathcal{C}(G, S) = \text{PQ}$.

Therefore, an ideal *strategy* of presenting nodes to the user is able to get us quickly from $S = \varnothing$ to a sample S s.t. $\mathcal{C}(G, S) = \{q\}$. In particular, a good strategy should not propose to the user the *certain nodes* i.e., nodes not yielding new information when labeled by the user. Formally, given a graph G, a sample S, and an unlabeled node $\nu \in G$, we say that ν is *certain* (w.r.t. S) if it belongs to one of the following sets:

$$Cert_+(G, S) = \{\nu \in G \mid \forall q \in \mathcal{C}(G, S).\ \nu \in q(G)\},$$
$$Cert_-(G, S) = \{\nu \in G \mid \forall q \in \mathcal{C}(G, S).\ \nu \notin q(G)\}.$$

In other words, a node is certain with a label α if labeling it explicitly with α does not eliminate any query from $\mathcal{C}(G, S)$. For instance, take the graph in Figure 5.12 with a positive, a negative, and an unlabeled node, which belongs to $Cert_+$ because it is selected by the unique (prefix-free) query in $\mathcal{C}(G, S)$ (i.e., b). Additionally, we observe that labeling it otherwise (i.e., with a $-$) leads to an inconsistent sample. The notion of certain nodes is inspired by possible world semantics and certain answers [ILJ84] and already employed for XML querying for non-expert users [CW13]. Recall that we have defined a similar notion in the previous chapter for the inference of relational joins.

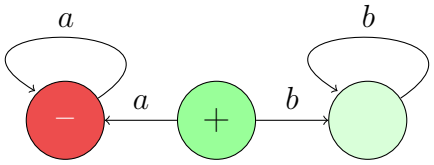


Figure 5.12: Two labeled nodes and a certain node.

Next, we give necessary and sufficient conditions for a node to be certain, for both positive and negative labels:

Lemma 5.4.2 *Given a sample S and a node ν from G:*

1. $\nu \in Cert_+(G, S)$ *iff there exists* $\nu' \in S_+$ *s.t.* $paths_G(\nu') \subseteq paths_G(S_-) \cup paths_G(\nu)$,
2. $\nu \in Cert_-(G, S)$ *iff* $paths_G(\nu) \subseteq paths_G(S_-)$.

Proof From Lemma 5.4.1, we can derive that given a graph G, a consistent sample S over G, and an unlabeled node ν from G, ν is informative iff both $S \cup \{(\nu, +)\}$ and $S \cup \{(\nu, -)\}$ are consistent. Then, we can rewrite this equivalence and obtain a necessary and sufficient condition for a node to be certain i.e., ν is certain iff (i) $S \cup \{(\nu, +)\}$ is not consistent or (ii) $S \cup \{(\nu, -)\}$ is not consistent. From this observation and the characterization of a sample to be consistent (Lemma 5.3.1) we infer that the two parts of the Lemma are true. □

Additionally, given a graph G, a sample S, and a node ν, we say that ν is *informative* (w.r.t. S) if ν has not been labeled by the user nor it is certain. Unfortunately, we derive the following.

Lemma 5.4.3 *Given a graph G and a sample S, deciding whether a node ν is informative is PSPACE-complete.*

Proof The membership to PSPACE follows from Lemma 5.4.2 and the known result that deciding the inclusion of NFAs is PSPACE-complete [SM73].

For the PSPACE-hardness, take the same reduction from the proof of Lemma 5.3.2, the only difference being that the node ν_{n+2} is unlabeled (i.e., it is no more a positive example). Hence, we have the same graph, the same set of negative examples $S_- = \{\nu_1, \ldots, \nu_n, \nu_{n+1}\}$ and an empty set of positive examples $S_+ = \varnothing$. We claim that the union of the DFAs $D_1, \ldots, D_n$ is not universal iff ν_{n+2} is informative. From Lemma 5.4.1, we can derive that this is equivalent to saying that the union of the DFAs $D_1, \ldots, D_n$ is not universal iff both $S \cup \{(\nu_{n+2}, +)\}$ and $S \cup \{(\nu_{n+2}, -)\}$ are consistent. Notice that $S \cup \{(\nu_{n+2}, -)\}$ is clearly consistent since any query a (where $a \in \Sigma$) is consistent with it. Thus, we have to prove that $S \cup \{(\nu_{n+2}, +)\}$ is consistent iff the union of the DFAs $D_1, \ldots, D_n$ is not universal, which follows exactly as in the proof of Lemma 5.3.2. ▫

An intelligent strategy should propose to the user only informative nodes. Since deciding the informativeness of a node is intractable (cf. Lemma 5.4.3), we need to explore *practical strategies* that efficiently compute the next node to label. Consequently, we propose two simple but effective strategies that we detail next and that we have evaluated experimentally. The basic idea behind them is to avoid the intractability of deciding informativeness of a node by looking only at a small number of paths of that node. More precisely, we say that a node is *k-informative* if it has at least one path of length at most k that is not covered by a negative example. If a node is k-informative, then it is also informative, otherwise we are not able to establish its informativeness w.r.t. the current k. Then, strategy kR consists of taking *randomly* a k-informative node while strategy kS consists of taking the k-informative node having the *smallest* number of non-covered k-paths, thus favoring the nodes for which computing the SCPs is easier. In the next section, we discuss the performance of these strategies as well as how we set the k in practice.

5.5 Experiments

In this section, we present an experimental study devoted to gauge the performance of our learning algorithms. In Section 5.5.1, we introduce the

used datasets: the *AliBaba biological graph* and randomly generated *synthetic graphs*. In Section 5.5.2 and Section 5.5.3, we present the results for the two settings under study: *static* and *interactive*, respectively. Our algorithms have been implemented in C and our experiments have been run on an Intel Core i7 with 4×2.9 GHz CPU and 8 GB RAM.

5.5.1 Datasets

Despite the increasing popularity of graph databases, benchmarks allowing to assess the performance of emerging graph database applications are still lacking or under construction [BFG+13]. In particular, there is no established benchmark devoted to graph queries defined by regular expressions. Due to this lack, we have adopted a real dataset recently used by [KL12] to evaluate the performance of optimization algorithms for regular path queries. This dataset, called *AliBaba* [PNLH09], represents a real graph from research on biology, extracted by text mining on PubMed. The dataset has a semantic part consisting of a network of protein-protein interactions and a textual part, reporting text co-occurrence for words. The first part was more appropriate to apply our learning algorithms than the second. Therefore, we have extracted the semantic part from the original graph, thus obtaining a subgraph of about 3k nodes and 8k edges. Similarly, from the set of real-life queries reported in [KL12], we have retained those that select at least one node on the graph to obtain at least one positive example for learning. Thus, we have used 6 biological queries (denoted by $bio_1, \ldots, bio_6$), which are structurally complex and have selectivities varying from 1 to a total of 711 nodes i.e., from 0.03% to 22% of the nodes of the graph. We summarize these queries in Table 5.1. By small letters a, b we denote symbols from the alphabet while by capital letters A, C, E, I we denote disjunctions of symbols from the alphabet i.e., expression of the form $a_1 + \ldots + a_n$. These disjunctions contain up to 10 symbols, with possibly overlapping ones among them.

	Query	*Selectivity*
bio_1	$b \cdot A \cdot A^*$	0.03%
bio_2	$C \cdot C^* \cdot a \cdot A \cdot A^*$	0.2%
bio_3	$C \cdot E$	3%
bio_4	$I \cdot I \cdot I^*$	11%
bio_5	$A \cdot A \cdot A^* \cdot I \cdot I \cdot I^*$	12%
bio_6	$A \cdot A \cdot A^*$	22%

Table 5.1: Biological queries.

Additionally, we have implemented a synthetic data generator, which

yields graphs of varying size and similar to real-world graphs. The latest feature let us generate *scale-free* graphs with a *Zipfian* edge label distribution [KL12]. We report here the results for generated graphs of size 10k, 20k, and 30k nodes, and with a number of edges three times larger. Moreover, we focus on synthetic queries that are similar in structure to the aforementioned real-life biological queries. In particular, the three queries that we report here (denoted by syn_1, syn_2, and syn_3) have the structure $A \cdot B^* \cdot C$, where A, B, and C are disjunctions of up to 10 symbols, with overlapping ones among them. The difference between these three queries is w.r.t. their selectivity: regardless the actual size of the graph, syn_1, syn_2, and syn_3 select 1%, 15%, and 40% of the graph nodes, respectively.

Before presenting the experimental results, we say a few words about how we set empirically the parameter k from *learner*. Since in our experiments we assume that the user labels the nodes of the graph consistently with some goal query, the input sample is always consistent. Hence, there exists a consistent path for each positive node and we dynamically discover the length of the SCPs. In particular, we start with $k = 2$; if for a given k, the query learned using SCPs shorter that k does not select all positive nodes, we increment k and iterate. For the interactive case, the aforementioned procedure becomes: start with $k = 2$; seek k-informative nodes (cf. Section 5.4.2) and increase k when the current k does not yield any k-informative node. In practice, in the majority of cases $k = 2$ is sufficient and it may reach values up to 4 in some isolated cases.

5.5.2 Static experiments

The setup of static experiments is as follows. Given a graph and a goal query, we take as positive examples some random nodes of the graph that are selected by the query and as negative examples some random nodes that are not selected by it. All these examples are given as input to *learner*, which returns a consistent query. We consider the learned query as a binary classifier and we measure the F1 score w.r.t. the goal query. Thus, for different percentages of labeled nodes in the graph, we measure the F1 score of the learned query along with the learning time. We present the summary of results in Figure 5.13 and 5.14, which show the F1 score and the learning time for the biological (a) and synthetic queries (b, c, d), respectively. Since the positive effect of the generalization in addition to the selection of SCPs is generally of 1% in F1 score, we do not highlight the two steps of the algorithm for the sake of figure readability.

We can observe that, not surprisingly, by increasing the percentage of labeled nodes of the graph, the F1 score also increases (Figure 5.13). Overall,

the F1 score is 1 or sufficiently close to 1 for all queries, except a few cases that we discuss below. The worst behavior is clearly exhibited by query bio_5, when we can observe that the F1 score converges to 1 less faster than the others. For this query, we can also observe that the learning time is higher (Figure 5.14(a)). This is due to the fact that the graph is not characteristic for it (cf. Section 5.3.3), hence the selection of SCPs yields paths that are not relevant for the target query.

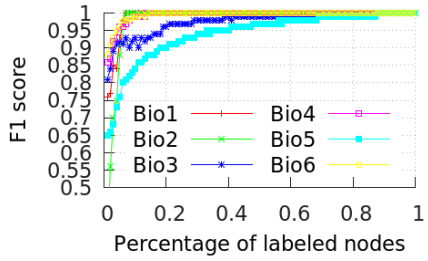
For what concerns the learning time, this remains reasonable (of the order of seconds) for all the queries and for both datasets. The most selective queries (bio_4, bio_5, bio_6) are more problematic since they entail a larger number of positive nodes in the step of selection of the SCPs. As a conclusion, notice that even when the F1 score is very high, these results on the static scenario are not fully beneficial since we need to label at least 7% of the graph nodes to have an F1 score equal to 1. As we later show with the interactive experiments, we can significantly reduce the number of labels needed to reach an F1 score equal to 1.

Finally, the synthetic experiments on various graph sizes and query selectivities confirm the aforementioned observations. In particular, when the queries are more selective (as syn_2 and syn_3), increasing the number of examples implies more visible changes in the learning time (cf. Figure 5.14). Additionally, we observe the goal queries with higher selectivity converge faster to a F1 score equal to 1 (cf. Figure 5.13). Intuitively, this is due to the fact that such cases imply a bigger number of positive examples from which the learning algorithm can benefit to generalize faster the goal query.

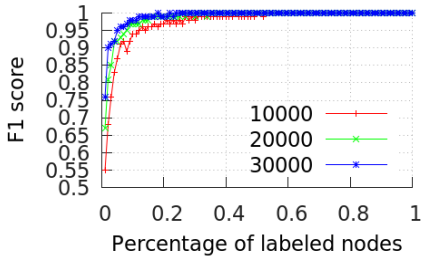
5.5.3 Interactive experiments

The setup of interactive experiments is as follows. Given a graph and a goal query, we start with an empty sample and we continuously select a node that we ask the user to label, until the learned query selects exactly the same set of nodes as the goal query or, in other words, until the goal query and the learned query are indistinguishable by the user (cf. Section 5.3.3). This corresponds to obtaining an F1 score of 1. In this setting, we measure the percentage of the labeled nodes of the graph and the learning time i.e., the time needed to compute the next node to label. In particular, the number of interactions corresponds to the total number of examples, the latter being the sum of the number of positive examples and the number of negative examples. The summary of interactive experiments is presented in Table 5.2.

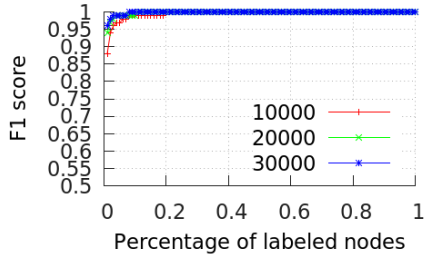
We can observe that, differently from the static scenario, labeling around 1% of the nodes of the graph suffices to learn a query with F1 score equal to 1. Even for the most difficult one (bio_5), we get a rather significant



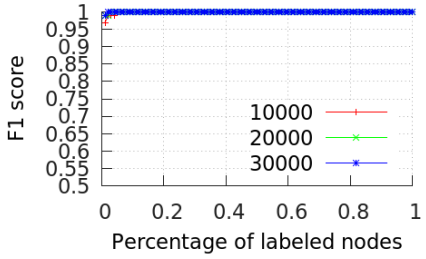
(a) Biological queries.



(b) Synthetic query syn_1.

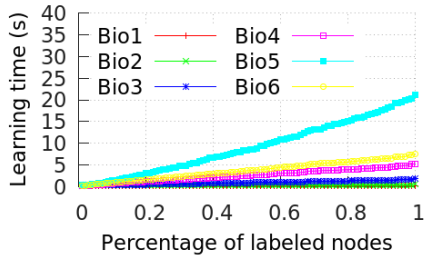


(c) Synthetic query syn_2.

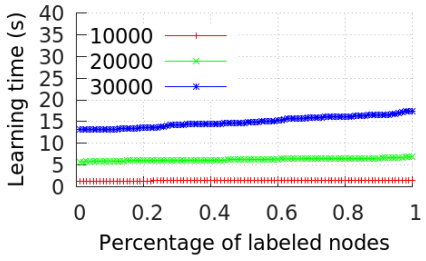


(d) Synthetic query syn_3.

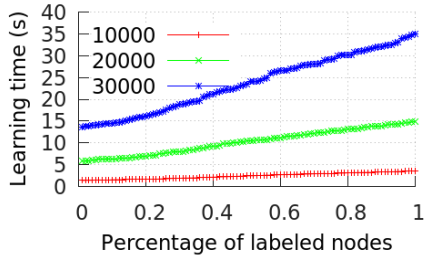
Figure 5.13: Summary of static experiments – F1 score.



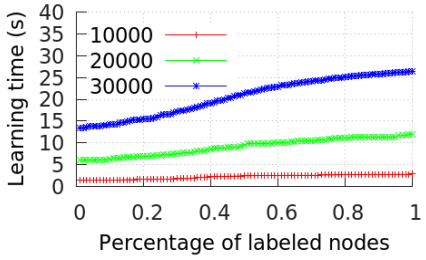
(a) Biological queries.



(b) Synthetic query syn_1.



(c) Synthetic query syn_2.



(d) Synthetic query syn_3.

Figure 5.14: Summary of static experiments – Learning time (seconds).

Dataset	*Bio query* / *Graph size*	*Labels needed for F1 score = 1* **without** *interactions*	*Interactive strategy*	*Labels needed for F1 score = 1* **with** *interactions*	*Time between interactions (seconds)*
Biological queries	bio_1	7%	kR	0.06%	0.19
			kS	0.06%	0.33
	bio_2	7%	kR	1.78%	0.26
			kS	3.13%	0.48
	bio_3	66%	kR	1.24%	0.34
			kS	1.49%	0.45
	bio_4	12%	kR	1.32%	0.23
			kS	0.22%	0.53
	bio_5	87%	kR	7.7%	3.45
			kS	7.39%	3.79
	bio_6	12%	kR	1.18%	0.24
			kS	0.35%	0.3
Synthetic query syn_1	10000	51%	kR	0.15%	1.33
			kS	0.17%	1.35
	20000	26%	kR	0.07%	5.83
			kS	0.06%	5.92
	30000	22%	kR	0.04%	13.5
			kS	0.04%	13.95
Synthetic query syn_2	10000	20%	kR	0.38%	1.57
			kS	0.36%	1.58
	20000	11%	kR	0.23%	6.63
			kS	0.22%	6.78
	30000	8%	kR	0.17%	15.24
			kS	0.16%	15.38
Synthetic query syn_3	10000	5%	kR	0.1%	1.32
			kS	0.1%	1.32
	20000	3%	kR	0.05%	5.66
			kS	0.05%	5.68
	30000	2%	kR	0.04%	13.15
			kS	0.04%	13.41

Table 5.2: Summary of interactive experiments.

improvement, as the percentage of interactions is drastically reduced to 7.7% of the nodes in the graph (while in the static case it was 87% of the nodes in the graph). Overall, these numbers prove that the interactive scenario considerably reduces the number of examples needed to infer a query of F1 score equal to 1. The synthetic experiments confirm this behavior for various graph sizes and query selectivities. While learning a query with F1 score equal to 1 corresponds to the strongest halt condition of our interactive scenario (cf. Figure 5.11), we believe that in practice the user may choose to stop the interactions earlier (and hence label less nodes) if she is satisfied by an intermediate query proposed by the learning algorithm. We consider such halt conditions in our system demo [BCL15a].

Moreover, these experiments also show that the two strategies (kS and kR, cf. Section 5.4.2) have a similar behavior, even though kS is slightly better (w.r.t. minimizing the number of interactions) for the most selective queries. Intuitively, this happens because such a strategy favors the nodes for which computing the SCPs has a smaller space of solutions (cf. Section 5.4.2). Finally, we can observe that the two strategies are also efficient, as they lead to a learning time of the order of seconds in all cases.

5.6 Related work

Learning queries from examples is a popular and interesting topic in databases, as we have already pointed out in the related work of the previous chapter (i.e., Section 4.7). In particular, in this chapter we have used *grammatical inference* [dlH10] i.e., the branch of machine learning that aims at constructing a formal grammar by generalizing a set of examples. Existing works on learning tree patterns [SW12], schemas [BGNV10], and transformations [LMN10, LLN+14] for XML are also based on it.

Our definition of learnability is inspired by the well-known framework of *language identification in the limit* [Gol78], which requires a learning algorithm to be polynomial in the size of the input, *sound* (i.e., always return a concept consistent with the examples given by the user or a special *null* value if such concept does not exist) and *complete* (i.e., able to produce every concept with a sufficiently rich set of examples). In our case, we show that checking the consistency of a given set of examples is intractable, which implies that there is no algorithm able to answer *null* in polynomial time when the sample is inconsistent. This leads us to the conclusion that path queries are not learnable in the classical framework. Consequently, we slightly modify the framework and require the algorithm to learn the goal query in polynomial time if a polynomially-sized characteristic set of examples is pro-

vided. This learning framework has been recently employed for learning XML transformations [LLN+14] and is referred to as learning with *abstain* since the algorithm can abstain from answering when the characteristic set of examples is not provided.

The classical algorithm for learning a regular language from positive and negative word examples is RPNI [OG92], which basically works as follows: (i) construct a DFA (usually called prefix tree acceptor or PTA [dlH10]) that selects all positive examples; (ii) generalize it by state merges while no negative example is covered. Unfortunately, RPNI cannot be directly adapted to our setting since the input positive and negative examples are not words in our case. Instead, we have a set of graph nodes starting from which we have to select (from a potentially infinite set) the paths that led the user to label them as examples. After selecting such paths, we generalize by state merges, similarly to RPNI.

A problem closely related to learning is *definability*, recently studied for graph databases [ANS13]. Learning and definability have in common the fact that they look for a query consistent with a set of examples. The difference is that learning allows the query to select or not the nodes that are not explicitly labeled as positive examples while definability requires the query to select nothing else than the set of positive examples (i.e., all other nodes are implicitly negative). Nonetheless, some of the intractability proofs for definability can be adapted to our learning framework to show the intractability of consistency checking (cf. Section 5.3). To date, no polynomial algorithms have been yet proposed to construct path queries from a consistent set of examples.

Conclusions

Summary of the contributions

In this thesis, we have first focused on *schema formalisms for unordered XML*. We have investigated languages of unordered words and proposed disjunctive interval multiplicity expressions (DIMEs), a subclass of unordered regular expressions for which two fundamental decision problems, membership of an unordered word to the language of a DIME and containment of two DIMEs, are tractable. Next, we have employed DIMEs to define languages of unordered trees and proposed disjunctive interval multiplicity schema (DIMS) and its restriction, disjunction-free interval multiplicity schema (IMS). DIMSs and IMSs can be seen as DTDs using restricted classes of regular expressions and interpreted under commutative closure to define unordered content models. These restrictions allow to maintain a relatively low computational complexity of basic static analysis problems while allowing to capture a significant part of the expressive power of practical DTDs.

Then, we have studied the problem of *relational-to-graph data exchange*. Our main results are the proofs of intractability of the existence of solutions and query answering that hold even under considerable restrictions of the problem setting.

Furthermore, we have investigated the problem of *learning schemas for unordered XML*. We have proven that the DIMEs and DIMSs are not learnable in the general case where positive and negative examples are allowed, by using the intractability of the consistency checking. Consequently, we have identified two practical restrictions that are learnable: one from positive examples only, and another one from both positive and negative examples. Additionally, for both learnable cases we have proposed learning algorithms that return minimal schemas consistent with the examples.

Next, we have studied the problem of *learning relational join queries* without the knowledge of referential integrity constraints. We have investigated various settings, depending on whether or not we allow disjunction and/or projection in the queries. We have precisely characterized the frontier between tractability and intractability for the following problems of interest in these settings: consistency checking, learnability, and deciding the informativeness of a tuple. Then, we have proposed several efficient strategies of presenting tuples to the user and we have discussed their performance on benchmark and synthetic datasets.

Finally, we have studied the problem of *learning path queries* defined by regular expressions on graph databases. We have identified fundamental difficulties of the problem setting, formalized what means for a class of queries to be learnable, and shown that the above class enjoys learnability. Additionally, we have investigated an interactive scenario, analyzed what means for a node to be informative, and proposed practical strategies of presenting examples to the user. Finally, we have shown the effectiveness of the algorithms and the improvements of using an interactive approach through an experimental study on both real and synthetic datasets.

Perspectives

Regarding our work on schema formalisms for unordered XML, we would like to study whether the restrictions imposed by the grammar of DIMEs can be relaxed while maintaining the tractability of the problems of interest. We also aim to apply the unordered schemas to query minimization [AYCLS02] i.e., given a query and a schema, find a smaller yet equivalent query in the presence of the schema.

Additionally, it would be interesting to study whether the expressive power of the learnable classes of DIMEs can be extended, and to use unordered schemas and optimization techniques to boost the learning algorithms for twig queries [SW12]. We also want to use a more specific schema learnability condition i.e., to require the size (instead of the cardinality) of the characteristic sample to be polynomial in the size of the alphabet, in order to fully adhere to the classical definition of the characteristic sample in the context of grammatical inference [dlH97]. We believe that this can be accomplished by using a compressed representation of the XML documents with directed acyclic graphs [LMN13] and in such a case the proposed schema learning algorithms will work without any alteration.

Another interesting direction of future work is to extend our approach for learning relational queries to other algebraic operators. Additionally, our study makes sense in realistic crowdsourcing scenarios, thus we could think of crowd users to provide positive and negative examples for join inference. Finally, we think at employing our approach in data cleaning scenarios, where the user may label pairs of possibly conflicting tuples and then our algorithms could infer the conflicts that determined the user's label and a potential conflict resolution.

We envision several directions of our work on learning path queries on graphs, one of which being to sample a graph and finding informative nodes on representative samples, in the spirit of [LF06]. Moreover, motivated by the

absence of benchmarks devoted to queries defined by regular expressions, we want to develop such a benchmark. This would permit to better analyze the performance of algorithms involving regular expressions on graphs, including our proposed learning algorithms.

Regarding our work on relational-to-graph data exchange, we would like to investigate the complexity upper bounds of the problems of interest and look for tractable fragments to have a complete picture of the difficulty of our setting. A natural question that remains open is how to query universal representatives consisting of a pair (graph pattern, set of target constraints). We would also like to investigate practical scenarios of relational-to-RDF data exchange and other classes of heterogeneous schema mappings. Additionally, it would be interesting to combine the learning techniques developed in the last two chapters of the thesis for relational and graph queries in order to propose algorithms that automatically infer relational-to-graph mappings from examples provided by the user.

Bibliography

[AAP+13] A. Abouzied, D. Angluin, C. H. Papadimitriou, J. M. Hellerstein, and A. Silberschatz. Learning and verifying quantified boolean queries by example. In *PODS*, pages 49–60, 2013. 4, 98, 141, 142

[ABV12] S. Abiteboul, P. Bourhis, and V. Vianu. Highly expressive query languages for unordered data trees. In *ICDT*, pages 46–60, 2012. 11

[ADLM14] S. Amano, C. David, L. Libkin, and F. Murlak. XML schema mappings: Data exchange and metadata management. *J. ACM*, 61(2):12, 2014. 68

[AG08] R. Angles and C. Gutiérrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1), 2008. 145

[AGW01] J. Albert, D. Giammarresi, and D. Wood. Normal form algorithms for extended context-free grammars. *Theor. Comput. Sci.*, 267(1-2):35–47, 2001. 44

[AHS12] A. Abouzied, J. M. Hellerstein, and A. Silberschatz. Playful query specification with DataPlay. *PVLDB*, 5(12):1938–1941, 2012. 98, 142

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995. 104

[AL08] M. Arenas and L. Libkin. XML data exchange: Consistency and query answering. *J. ACM*, 55(2), 2008. 3, 57, 68

[Ang80] D. Angluin. Inductive inference of formal languages from positive data. *Information and Control*, 45(2):117–135, 1980. 81

[Ang82] D. Angluin. Inference of reversible languages. *J. ACM*, 29(3):741–765, 1982. 81

[Ang88] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988. 100, 117, 142, 167

[ANS13] T. Antonopoulos, F. Neven, and F. Servais. Definability problems for graph query languages. In *ICDT*, pages 141–152, 2013. 67, 155, 156, 157, 178

[AP11] M. Arenas and J. Pérez. Querying semantic web data with SPARQL. In *PODS*, pages 305–316, 2011. 145

[AtCKT11a] B. Alexe, B. ten Cate, P. G. Kolaitis, and W. C. Tan. Designing and refining schema mappings via data examples. In *SIGMOD Conference*, pages 133–144, 2011. 98, 100, 143

[AtCKT11b] B. Alexe, B. ten Cate, P. G. Kolaitis, and W. C. Tan. EIRENE: Interactive design and refinement of schema mappings via data examples. *PVLDB*, 4(12):1414–1417, 2011. 98, 100, 143

[AYCLS02] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Tree pattern query minimization. *VLDB J.*, 11(4):315–331, 2002. 3, 13, 77, 180

[Ban78] F. Bancilhon. On the completeness of query languages for relational data bases. In *MFCS*, pages 112–123, 1978. 4, 142

[Bar13] P. Barceló. Querying graph databases. In *PODS*, pages 175–188, 2013. 4, 146, 147, 150, 151

[BBC15] I. Boneva, A. Bonifati, and R. Ciucanu. Graph data exchange with target constraints. In *EDBT/ICDT Workshops – GraphQ*, pages 171–176, 2015. 7, 59

[BBG13] G. Bagan, A. Bonifati, and B. Groz. A trichotomy for regular simple path queries on graphs. In *PODS*, pages 261–272, 2013. 4, 147

[BBH11] M. Berglund, H. Björklund, and J. Högberg. Recognizing shuffled languages. In *LATA*, pages 142–154, 2011. 56

[BBR11] Z. Bellahsene, A. Bonifati, and E. Rahm, editors. *Schema Matching and Mapping*. Springer, 2011. 57

[BCL15a] A. Bonifati, R. Ciucanu, and A. Lemay. Interactive path query specification on graph databases. In *EDBT*, pages 505–508, 2015. 7, 149, 177

[BCL15b] A. Bonifati, R. Ciucanu, and A. Lemay. Learning path queries on graph databases. In *EDBT*, pages 109–120, 2015. 7, 149

[BCLS14] A. Bonifati, R. Ciucanu, A. Lemay, and S. Staworko. A paradigm for learning queries on big data. In *Data4U*, pages 7–12, 2014. 7

[BCS13] I. Boneva, R. Ciucanu, and S. Staworko. Simple schemas for unordered XML. In *WebDB*, pages 13–18, 2013. 7, 15, 16, 78, 85, 94

[BCS14a] I. Boneva, R. Ciucanu, and S. Staworko. Schemas for unordered XML on a DIME. *Theory Comput. Syst.*, 2014. 7, 15, 16

[BCS14b] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive inference of join queries. In *EDBT*, pages 451–462, 2014. 7, 101

[BCS14c] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive join query inference with JIM. *PVLDB*, 7(13):1541–1544, 2014. 7

[BCS14d] A. Bonifati, R. Ciucanu, and S. Staworko. Learning join queries from user examples (under journal submission), 2014. 7, 101

[Bel14] K. Belhajjame. Annotating the behavior of scientific modules using data examples: A practical approach. In *EDBT*, pages 726–737, 2014. 148

[BFG08] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. *J. ACM*, 55(2), 2008. 15, 39, 41, 56

[BFG+13] P. A. Boncz, I. Fundulaki, A. Gubichev, J.-L. Larriba-Pey, and T. Neumann. The linked data benchmark council project. *Datenbank-Spektrum*, 13(2):121–129, 2013. 172

[BGNV10] G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. *TWEB*, 4(4), 2010. 81, 95, 141, 177

[BKW98] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Inf. Comput.*, 142(2):182–206, 1998. 15, 55

[BLR11] P. Barceló, L. Libkin, and J. L. Reutter. Querying graph patterns. In *PODS*, pages 199–210, 2011. 58, 64, 65

[BM99] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *ICDT*, pages 296–313, 1999. 14, 54, 55

[BMS13] H. Björklund, W. Martens, and T. Schwentick. Validity of tree pattern queries with respect to schema information. In *MFCS*, pages 171–182, 2013. 56

[BNST06] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise DTDs from XML data. In *VLDB*, pages 115–126, 2006. 95

[BNSV10] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren. Inference of concise regular expressions and DTDs. *ACM Trans. Database Syst.*, 35(2), 2010. 23, 54, 55, 95

[BNV07] G. J. Bex, F. Neven, and S. Vansummeren. Inferring XML schema definitions from XML data. In *VLDB*, pages 998–1009, 2007. 96

[BNVdB04] G. J. Bex, F. Neven, and J. Van den Bussche. DTDs versus XML Schema: A practical study. In *WebDB*, pages 79–84, 2004. 12, 54, 95, 96

[BPR13] P. Barceló, J. Pérez, and J. L. Reutter. Schema mappings and data exchange for graph databases. In *ICDT*, pages 189–200, 2013. 3, 57, 58, 59, 60, 62, 64, 65, 69

[BT05] I. Boneva and J. Talbot. Automata and logics for unranked and unordered trees. In *RTA*, pages 500–515, 2005. 54, 55

[BTT05] I. Boneva, J. Talbot, and S. Tison. Expressiveness of a spatial logic for trees. In *LICS*, pages 280–289, 2005. 54, 55

[BV84] C. Beeri and M. Y. Vardi. A proof procedure for data dependencies. *J. ACM*, 31(4):718–741, 1984. 58, 59, 60

[CDLM13] W. Czerwinski, C. David, K. Losemann, and W. Martens. Deciding definability by deterministic regular expressions. In *FoSSaCS*, pages 289–304, 2013. 55

[CG04] L. Cardelli and G. Ghelli. TQL: a query language for semistructured data based on the ambient logic. *Mathematical Structures in Computer Science*, 14(3):285–327, 2004. 55

[CGK13] A. Calì, G. Gottlob, and M. Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *J. Artif. Intell. Res. (JAIR)*, 48:115–174, 2013. 68

[CGLN07] J. Carme, R. Gilleron, A. Lemay, and J. Niehren. Interactive learning of node selecting tree transducer. *Machine Learning*, 66(1):33–67, 2007. 2

[CGPS13] D. Colazzo, G. Ghelli, L. Pardini, and C. Sartiani. Almost-linear inclusion for XML regular expression types. *ACM Trans. Database Syst.*, 38(3):15, 2013. 23, 56

[CGS09] D. Colazzo, G. Ghelli, and C. Sartiani. Efficient inclusion for a class of XML types with interleaving and counting. *Inf. Syst.*, 34(7):643–656, 2009. 23, 56

[Ciu13] R. Ciucanu. Learning queries for relational, semi-structured, and graph databases. In *SIGMOD/PODS PhD Symposium*, pages 19–24, 2013. 7

[CS13] R. Ciucanu and S. Staworko. Learning schemas for unordered XML. In *DBPL*, pages 31–40, 2013. 7, 78

[CW13] S. Cohen and Y. Weiss. Certain and possible XPath answers. In *ICDT*, pages 237–248, 2013. 119, 143, 170

[dlH97] C. de la Higuera. Characteristic sets for polynomial grammatical inference. *Machine Learning*, 27(2):125–138, 1997. 77, 78, 80, 180

[dlH10] C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010. 149, 154, 158, 161, 177, 178

[DSPGMW10] A. Das Sarma, A. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. In *ICDT*, pages 89–103, 2010. 4, 141, 142

[DZL03] S. Dal-Zilio and D. Lugiez. XML schema, tree logic and sheaves automata. In *RTA*, pages 246–263, 2003. 55

[FGLX11] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE Trans. Knowl. Data Eng.*, 23(5):683–698, 2011. 142

[FGPVG09] G. Fletcher, M. Gyssens, J. Paredaens, and D. Van Gucht. On the expressive power of the relational algebra on finite sets of relation pairs. *IEEE Trans. Knowl. Data Eng.*, 21(6):939–942, 2009. 142

[FK13] D. D. Freydenberger and T. Kötzing. Fast learning of restricted regular expressions and DTDs. In *ICDT*, pages 45–56, 2013. 95

[FKK+11] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: answering queries with crowdsourcing. In *SIGMOD Conference*, pages 61–72, 2011. 4, 100, 148

[FKMP05] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005. 3, 57, 60, 62, 63, 65, 69

[Flo05] D. Florescu. Managing semi-structured data. *ACM Queue*, 3(8):18–24, 2005. 3, 77

[GCS08] G. Ghelli, D. Colazzo, and C. Sartiani. Linear time membership in a class of regular expressions with interleaving and counting. In *CIKM*, pages 389–398, 2008. 23, 56

[GGR+03] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: Learning document type descriptors from XML document collections. *Data Min. Knowl. Discov.*, 7(1):23–56, 2003. 95

[GM13] S. Grijzenhout and M. Marx. The quality of the XML web. *J. Web Sem.*, 19:59–68, 2013. 12, 54, 85, 94, 95, 96

[GMN09] W. Gelade, W. Martens, and F. Neven. Optimizing schema languages for XML: Numerical constraints and interleaving. *SIAM J. Comput.*, 38(5):2021–2043, 2009. 56

[Gol78] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978. 5, 77, 78, 80, 95, 100, 107, 142, 153, 154, 158, 177

[GS10] G. Gottlob and P. Senellart. Schema mapping discovery from data instances. *J. ACM*, 57(2), 2010. 143

[GV90] P. Garcia and E. Vidal. Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(9):920–925, 1990. 81

[Hei01] C. Heinlein. Workflow and process synchronization with interaction expressions and graphs. In *ICDE*, pages 243–252, 2001. 147

[HKIF11] K. Hashimoto, Y. Kusunoki, Y. Ishihara, and T. Fujiwara. Validity of positive XPath queries with wildcard in the presence of DTDs. In *DBPL*, 2011. 56

[Hov12] D. Hovland. The membership problem for regular expressions with unordered concatenation and numerical constraints. In *LATA*, pages 313–324, 2012. 56

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979. 152

[ILJ84] T. Imielinski and W. Lipski Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984. 119, 170

[JCE+07] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *SIGMOD Conference*, pages 13–24, 2007. 1

[JKL+14] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri. Towards a query-by-example system for knowledge graphs. In *GRADES*, 2014. 147

[KL12] A. Koschmieder and U. Leser. Regular path queries on large graphs. In *SSDBM*, pages 177–194, 2012. 4, 147, 172, 173

[Koz77] D. Kozen. Lower bounds for natural proof systems. In *FOCS*, pages 254–266, 1977. 155

[KPSS14] P. G. Kolaitis, R. Pichler, E. Sallinger, and V. Savenkov. Nested dependencies: structure and reasoning. In *PODS*, pages 176–187, 2014. 3, 57

[KPT06] P. G. Kolaitis, J. Panttaja, and W. C. Tan. The complexity of data exchange. In *PODS*, pages 30–39, 2006. 68

[KT10] E. Kopczynski and A. To. Parikh images of grammars: Complexity and applications. In *LICS*, pages 80–89, 2010. 14, 20, 55

[KV94] M. J. Kearns and U. V. Vazirani. *An introduction to computational learning theory.* MIT Press, 1994. 77, 141, 149

[LF06] J. Leskovec and C. Faloutsos. Sampling from large graphs. In *KDD*, pages 631–636, 2006. 180

[LLN+14] G. Laurence, A. Lemay, J. Niehren, S. Staworko, and M. Tommasi. Learning sequential tree-to-word transducers. In *LATA*, pages 490–502, 2014. 158, 177, 178

[LM13] K. Losemann and W. Martens. The complexity of regular expressions and property paths in SPARQL. *ACM Trans. Database Syst.*, 38(4):24, 2013. 4, 147

[LMN10] A. Lemay, S. Maneth, and J. Niehren. A learning algorithm for top-down XML transformations. In *PODS*, pages 285–296, 2010. 141, 177

[LMN13] M. Lohrey, S. Maneth, and E. Noeth. XML compression via DAGs. In *ICDT*, pages 69–80, 2013. 180

[LNG06] A. Lemay, J. Niehren, and R. Gilleron. Learning n-ary node selecting tree transducers from completely annotated examples. In *ICGI*, pages 253–267, 2006. 2

[LR92] K.-J. Lange and P. Rossmanith. The emptiness problem for intersections of regular languages. In *MFCS*, pages 346–354, 1992. 162

[MAC03] J.-K. Min, J.-Y. Ahn, and C.-W. Chung. Efficient extraction of schemas for XML documents. *Inf. Process. Lett.*, 85(1):7–12, 2003. 95

[MLVP14] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. Exemplar queries: Give me an example of what you need. *PVLDB*, 7(5):365–376, 2014. 147

[MN05] W. Martens and F. Neven. On the complexity of typechecking top-down XML transformations. *Theor. Comput. Sci.*, 336(1):153–180, 2005. 55

[MNG08] W. Martens, F. Neven, and M. Gyssens. Typechecking top-down XML transformations: Fixed input or output schemas. *Inf. Comput.*, 206(7):806–827, 2008. 55

[MNS04] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. In *MFCS*, pages 889–900, 2004. 54

[MNS09] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for XML schemas and chain regular expressions. *SIAM J. Comput.*, 39(4):1486–1530, 2009. 15

[MS94] A. J. Mayer and L. J. Stockmeyer. Word problems-this time with interleaving. *Inf. Comput.*, 115(2):293–311, 1994. 14

[MS04] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1):2–45, 2004. 42, 51, 53

[MWK+11] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. *PVLDB*, 5(1):13–24, 2011. 100

[MWM07] M. Montazerian, P. T. Wood, and S. R. Mousavi. XPath query satisfiability is in PTIME for real-world DTDs. In *XSym*, pages 17–30, 2007. 23

[NJ11] A. Nandi and H. V. Jagadish. Guided interaction: Rethinking the query-result paradigm. *PVLDB*, 4(12):1466–1469, 2011. 1

[NS] F. Neven and T. Schwentick. XML schemas without order. 1999. 14, 54, 55

[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*, 2(3), 2006. 15, 42, 56

[OG92] J. Oncina and P. García. Inferring regular languages in polynomial update time. *Pattern Recognition and Image Analysis*, pages 49–61, 1992. 159, 163, 178

[Opp78] D. C. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *J. Comput. Syst. Sci.*, 16(3):323–332, 1978. 21, 23

[Pap94] C. H. Papadimitriou. *Computational complexity.* Addison-Wesley, 1994. 87

[Par78] J. Paredaens. On the expressive power of the relational algebra. *Inf. Process. Lett.*, 7(2):107–111, 1978. 4, 142

[PLC+08] A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. *J. Data Semantics*, 10:133–173, 2008. 3, 57

[PNLH09] P. Palaga, L. Nguyen, U. Leser, and J. Hakenberg. High-performance information extraction with AliBaba. In *EDBT*, pages 1140–1143, 2009. 145, 172

[PV00] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *PODS*, pages 35–46, 2000. 54

[PVM+02] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating web data. In *VLDB*, pages 598–609, 2002. 3, 57

[QCJ12] L. Qian, M. J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In *SIGMOD Conference*, pages 73–84, 2012. 143

[RN10] S. J. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.).* Pearson Education, 2010. 124

[RS09] R. Ronen and O. Shmueli. SoQL: A language for querying and creating data in social networks. In *ICDE*, pages 1595–1602, 2009. 145

[SAM12] J. Sequeda, M. Arenas, and D. P. Miranker. On directly mapping relational databases to RDF and OWL. In *WWW*, pages 649–658, 2012. 3, 57

[SBG+15] S. Staworko, I. Boneva, J. E. L. Gayo, S. Hym, E. G. Prud'hommeaux, and H. R. Solbrig. Complexity and expressiveness of ShEx for RDF. In *ICDT*, pages 195–211, 2015. 3, 14, 23

[Sch78] T. J. Schaefer. The complexity of satisfiability problems. In *STOC*, pages 216–226, 1978. 21

[Sch04] T. Schwentick. Trees, automata and XML. In *PODS*, page 222, 2004. 15

[SK13] T. Sellam and M. L. Kersten. Meet Charles, big data query advisor. In *CIDR*, 2013. 143

[SM73] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time: Preliminary report. In *STOC*, pages 1–9, 1973. 14, 21, 155, 171

[SS07] L. Segoufin and C. Sirangelo. Constant-memory validation of streaming XML documents against DTDs. In *ICDT*, pages 299–313, 2007. 18, 38

[SSM03] H. Seidl, T. Schwentick, and A. Muscholl. Numerical document queries. In *PODS*, pages 155–166, 2003. 54, 55

[SSM08] H. Seidl, T. Schwentick, and A. Muscholl. Counting in trees. In *Logic and Automata*, pages 575–612, 2008. 21, 23, 54

[SV02] L. Segoufin and V. Vianu. Validating streaming XML documents. In *PODS*, pages 53–64, 2002. 18, 38

[SW12] S. Staworko and P. Wieczorek. Learning twig and path queries. In *ICDT*, pages 140–154, 2012. 2, 77, 78, 141, 177, 180

[SWK+02] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, pages 974–985, 2002. 54, 85, 94

[tCDK13] B. ten Cate, V. Dalmau, and P. G. Kolaitis. Learning schema mappings. *ACM Trans. Database Syst.*, 38(4):28, 2013. 141, 143

[TCP09] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *SIGMOD Conference*, pages 535–548, 2009. 4, 141, 142

[VG87] D. Van Gucht. On the expressive power of the extended relational algebra for the unnormalized relational model. In *PODS*, pages 302–312, 1987. 142

[W3C] W3C. XML Path language (XPath) 1.0, 1999. 13, 17

[WLK+13] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *SIGMOD Conference*, pages 229–240, 2013. 100

[Woo12] P. T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012. 4, 145, 146, 147, 150, 151

[YMHF01] L.-L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data-driven understanding and refinement of schema mappings. In *SIGMOD Conference*, pages 485–496, 2001. 149

[YZI+13] Z. Yan, N. Zheng, Z. G. Ives, P. P. Talukdar, and C. Yu. Actively soliciting feedback for query answers in keyword search-based data integration. *PVLDB*, 6(3):205–216, 2013. 142

[ZEPS13] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *SIGMOD Conference*, pages 809–820, 2013. 4, 134, 141, 142

[Zlo75] M. M. Zloof. Query by example. In *AFIPS National Computer Conference*, pages 431–438, 1975. 1, 97, 147