Querying Paradigms for the Web

Thesis submitted for the degree of
“Doctor of Philosophy”
by
Benny Kimelfeld

Submitted to the Senate of The Hebrew University
August, 2008
Querying Paradigms for the Web

Thesis submitted for the degree of
“Doctor of Philosophy”
by

Benny Kimelfeld

Submitted to the Senate of The Hebrew University
August, 2008
This work was carried out under the supervision of:

Prof. Yehoshua Sagiv
Acknowledgments

This thesis could not have been accomplished without the guidance and endless support of my advisor, Shuky Sagiv. Since I was an undergraduate student, Shuky has been teaching me how to do research, write scientific articles, present my research in public and promote my academic career. No one has believed in my potential and skills more than Shuky has, and for that I am extremely grateful. I am also grateful for Shuky’s priceless advice, for inspiring my research, and for encouraging me and giving me the freedom to explore new directions in research. Beyond all, Shuky has been much more than a doctoral advisor. His advice and guidance have spread over all aspects of life, from buying a car to caring for my health. I am extremely thankful to Shuky for that and, specifically, for rushing me to the emergency room in the middle of the night at the SIGMOD conference in Vancouver.

I wish to thank my great friends and research fellows Sara Cohen and Yaron Kanza. Sara and Yaron have guided and supported me throughout my studies and I have always regarded them as additional doctorate (and masters) advisors. Doing research with them has been a great joy, inspiring and highly fruitful. In particular, I thank Sara for joining the Hebrew University and closely collaborating with me during the last year, which led to a significant promotion of this thesis. I also thank Yaron for the collaboration in research, useful advice and long discussions about research, academy and life in general.

I also wish to thank the rest of my colleagues and collaborators for the fruitful work and great research environment. I particularly thank Itzhak Fadida, Yuri Kosharovsky and Konstantin Golenberg, the collaboration with whom led to important research results that were published in the top-ranked database conferences. I also thank Catriel Beeri for insightful discussions and for his useful suggestions on improving my work.

I am also grateful to my mother for continuously supporting my wife and me, and for constantly offering more and more aid. Her belief in me and pride in my achievements provided me with a lot of encouragement.

Finally, very special thanks go to my wife, Mali. She has been extensively and unconditionally supporting me since the beginning of my studies. By no means could this thesis
have been accomplished without her help. I especially thank Mali for taking upon herself most of the burden of raising our child, Jonathan, who was born during my Ph.D. studies. Interestingly, I felt that my research capabilities have been improved following the birth of Jonathan, and I greatly thank him for the inspiration.
Abstract

Knowledge discovery on the World-Wide Web poses new challenges to data-management systems. The heterogeneity of data sources requires effective query languages that are flexible in the sense that the user should be able to pose a query without having a complete (or any) knowledge of the underlying representation (schema). For that reason, current research largely focuses on paradigms that loosely couple the query with the actual data representation. Another issue is the uncontrolled nature of the Web, which leads to different degrees of incompleteness and impreciseness of its data sources. Effective query engines for such sources should be able to retrieve all the answers that are relevant to the query (in presence of data deficiencies) and, at the same time, distinguish between valuable and meaningless ones. Thus, it is essential to properly rank the answers. This dissertation explores various paradigms for querying Web data. These paradigms correspond to various types of data deficiencies and different levels of familiarity of the user with the underlying representation. Our focus is on the theoretical aspects, namely, establishment of formal frameworks for describing problems and solving them analytically, exploration of the underlying computational complexity and development of algorithmic techniques for query evaluation.

The first studied paradigm is that of keyword proximity search, namely, keyword search over data with structure (such as relational, semi-structured or XML data). This task is inherently different from the one of standard IR, since the goal is not merely to find individual data items that are relevant to the keywords, but also to discover the semantic relationships that hold among the keywords. We establish a formal framework in which the quality and performance of engines can be analyzed. Within this framework, we present the first algorithms that are provably correct and efficient.

Under the paradigm of maximal answers, query engines deal with data incompleteness by finding the answers that satisfy the conditions of the query in a maximal (rather than
complete) manner. In this paradigm, the full-disjunction operator corresponds to the traditional relational join. We have developed the first algorithm that provably produces the full disjunction of given relations as a stream of tuples (i.e., with polynomial delay) and it is shown that this algorithm significantly outperforms the state-of-art solutions. Our approach and results are extended to the general task of enumerating maximal induced subgraphs that satisfy a given hereditary graph property (i.e., a property that is closed w.r.t. induced subgraphs).

Keyword proximity search and maximal answers are two extreme paradigms. In the first, queries have extremely limited expressiveness (as they include no structural constraints at all) and answers are ranked according to their likelihood of matching any “reasonable” interpretation of the query. In the second, the query captures the exact meaning of the user and is answered in a best-effort manner. The next paradigm, ranked flexible querying (or just ranked querying for short) lies between these two extremes: the user poses an expressive query with vague (or flexible) conditions, and relies on a ranking mechanism to identify the relevant answers. We have explored different languages that enable users to actually express such queries, namely, to combine their available knowledge about the data with a specification on how matches to their vague conditions should be ranked. In one of these languages, the semantics allows answers to be incomplete (yet, maximal) while the ranking function can take the amount of incompleteness into account. We have devised evaluation algorithms that are efficient in the sense that for very general ranking functions, answers to queries can be produced in a provably incremental manner (in particular, the first answers are generated in an early stage without needing to generate all of the answers).

Probabilistic databases, and probabilistic XML as a special case, are used for representing data with uncertainties. Technically, data items are associated with probabilistic events and, consequently, the database is a probability distribution over ordinary (random) instances. Query evaluation in such databases was extensively studied in relational models. The literature contains a plethora of probabilistic XML models, but very little was known about querying in these models. In particular, it was not known whether the basic (yet highly important) projection operator can be handled efficiently in queries over probabilistic-
tic XML. Moreover, these models are defined by means of inherently different specifications and the exact relationship between them was unknown. We develop the abstract model of p-documents in which previously studied models are embedded. Within this model, we find the hierarchy that shows when an instance of one model can be efficiently translated into another. We prove that there is one maximal concrete model (which generalizes most of the ones in the literature) in which evaluation of twig queries with projection is tractable. Essentially, the ability to query efficiently necessitates an assumption of independence among the probabilistic items of a document.

To support real-life dependencies in probabilistic XML, we propose two approaches. The first is approximate query evaluation. The second is a novel approach that is based on constraints. In this approach, the probabilistic database is described by means of a p-document (from the above maximal concrete model) and a set of constraints that specify data integrity. The probability space is the sub-space of the p-document consisting of all the instances that satisfy the constraints. We show that, although highly complicated (yet natural) dependencies can be represented, this model supports efficient evaluation of twig queries with projection. Moreover, the tasks of testing well definedness and sampling can be performed efficiently as well.

Finally, we have studied a paradigm that combines both data uncertainty and data incompleteness, namely, maximal answers in probabilistic data. In particular, we have considered a well known model of probabilistic relational databases and studied the problem of maximally joining probabilistic relations. In addition to a thorough complexity analysis, we have developed efficient algorithms for various important tractable cases. We have also considered one of the above concrete families of p-documents (which is known as the “ProTDB” model) and presented an efficient algorithm for finding maximal matches of twig patterns. We have found that these problems are inherently different from their two components, i.e., maximal answers in deterministic data on the one hand and complete answers in probabilistic data on the other hand.

Many algorithms are involved in obtaining the results that are described above. In general, these algorithms have inherent differences and each requires special techniques.
However, some of the techniques are more general and are used across different paradigms and settings. Several of these techniques are well known whereas other have been developed by us. For example, we have developed the technique of partitioning the output space (combined with coroutines) which is used for the task of keyword proximity search. It is also used for finding maximal answers, in either ordinary or probabilistic data, along with a general technique that we have proposed for enumerating maximal induced subgraphs w.r.t. hereditary graph properties. The above techniques are used not only for constructing efficient algorithms, but also for proving computational hardness of enumeration problems. Lawler’s procedure (which was devised in the early seventies) is used by us in keyword proximity search and in various different settings of ranked querying. Finally, the traditional dynamic-programming technique is used in different tasks of query evaluation over probabilistic XML. Although these techniques have central roles in the algorithms, the intricate part is to adapt the relevant technique to the specific task at hand.

Due to formal limitations of The Hebrew University on the length of the thesis, the details (e.g., algorithms and proofs) are given for only some of the above results, namely, keyword proximity search and probabilistic XML. For the rest, we give an overview of the contributions.
# Table of Contents

1 Introduction

   1.1 Querying Paradigms ........................................ 2
   1.2 Principal Algorithmic Techniques ............................. 14
   1.3 Thesis Structure ........................................... 19

2 Preliminaries

   2.1 Complexity of Query Evaluation ............................. 20
   2.2 Enumeration and Top-k Answers ............................... 21
   2.3 Efficiency of Enumeration and Top-k Algorithms .......... 22
   2.4 Graphs and Trees ........................................... 23

3 Efficient Engines for Keyword Proximity Search

   3.1 Formal Framework ........................................... 25
   3.2 The Formal Results .......................................... 28
   3.3 The Basic Algorithm ......................................... 29
   3.4 Generating Tractable Constraints ............................ 34
   3.5 Computing Supertrees ......................................... 37
   3.6 Minimal Constrained Answers ................................ 43
   3.7 Adaptation to UQFs and SQFs ............................... 50
| 7  Conclusions | 103 |
| Bibliography | 108 |
| List of Publications | 108 |
| Additional References | 110 |
Chapter 1

Introduction

The World-Wide Web is a vast, constantly growing information store. Knowledge discovery in this environment has special characteristics that pose new challenges to data-management systems. For one, the uncontrolled nature of the Web results in data that is often partial and imprecise. Moreover, unlike the traditional scenario of querying well-known and established databases, users who wish to extract information from Web resources often need to do so with only a partial (or no) knowledge of the underlying data representation (e.g., schema). Taking these characteristics into account, when querying, necessarily leads to a high variance in the quality (or relevancy) of the answers. Thus, answers need to be ranked. In short, Web data is uncertain, incomplete, unknown and necessitates ranking of answers.

The goal of this thesis is to develop and explore paradigms of data extraction that are suitable to the Web, namely, ones that take into account deficiencies of data like uncertainty and incompleteness on the one hand, and on the other hand provide flexibility of query formulation that corresponds to different levels of familiarity of users with the data and its structure. The focus of the thesis is on the theoretical aspects of query evaluation in different querying paradigms, where the aim is to establish techniques and principles that would serve as the basis for practical tools.

In this chapter, we give an overview of all the work done during the Ph.D. research. In particular, we describe the studied paradigms and our contribution therein. Due to formal limitations posed by The Hebrew University on the length of the dissertation, the technical
results that are included in the following chapters concentrate only on some of our work about two of these paradigms. A technical abstract of all the results published during the Ph.D. work is given in Chapter 6.

1.1 Querying Paradigms

Various querying paradigms have been studied in this work. Some of these paradigms are aimed at coping with one of the above characteristic of the Web (e.g., unknown schema or impreciseness) while others handle several or all of them. The first querying paradigm that was studied is the highly popular keyword search.

1.1.1 Keyword Proximity Search

Perhaps the most extreme type of flexible queries is keyword search, namely, the query is merely a set of keywords and there are no structural constraints. This paradigm has been extensively explored by the information retrieval (IR) community in the past decades. There, the goal is to find documents that are relevant to the given keywords. IR was also applied to textual documents in XML format, where the goal is similar, namely, to search for relevant fragments (e.g., sections, paragraphs, etc.) rather than whole documents [49, 50, 100]. In recent years, there is a growing interest in applying keyword search to structured data sources, e.g., relational databases, semistructured data [91] and XML documents. This task is inherently different from the one of standard IR, since the goal is not merely to find individual data items (i.e., tuples, objects or XML elements) that are relevant to the keywords, but also to discover the semantic relationships that hold among the keywords.

As an example, Figure 1.1 depicts a tiny portion of the Mondial database. Suppose that a user poses the search query “Antwerp Belgium” to the tables of Figure 1.1. Observe that no tuple contains both keywords. Rather, a relevant answer is obtained by joining multiple tuples, e.g., the tuples $t_1$ and $t_5$ (namely, Antwerp is a city of Belgium). As another example, for the query “Brussels Belgium” there are two relevant answers, namely,
Brussels is a (capital) city of Belgium (the join of \( t_1 \) and \( t_6 \)) and Belgium is a member of the European Union which is located at Brussels. The latter answer is obtained by joining the path of tuples that comprises \( t_1 \), \( t_9 \), \( t_3 \) and \( t_6 \), respectively.

The quality of search results is determined by the frequency of the keywords as well as the associations among the keywords in the (structured) result. The number of elements that participate in the association is a major factor in the strength (relevancy) of a result. In principle, strong connections involve a small number of elements. Thus, we refer to this type of search as keyword proximity search.

Our first step in exploring this paradigm was to inspect the existing solutions in the literature. Not surprisingly, we found that there has been a lot of research on this issue. For instance, DBXplorer\[19\], BANKS\[24\], DISCOVER\[63\] and NUI TS\[105\] are systems that implement keyword search in relational databases. Algorithms for this task are also described in \[39\]\[62\]\[72\]\[81\]. XKeyword\[64\] is an extension of the techniques used in DISCOVER to keyword search in XML. The “backward search” algorithm used in BANKS is improved in \[69\] to a “bidirectional search.” Nevertheless, we found that it is very difficult to assess the quality of the different solutions since there is no agreement on the exact formulation of the problem as well as what constitutes an adequate (e.g., efficient) solution. For instance, the solutions of \[24\]\[39\]\[69\]\[105\] do not necessarily find both of the above answers for the query “Brussels Belgium,” whereas those of \[62\]\[63\]\[81\] find them but at a

<table>
<thead>
<tr>
<th>Countries</th>
<th></th>
<th>Organizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>capital</td>
<td>government</td>
</tr>
<tr>
<td>B Belgium BRU Monarchy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOR Norway OSL Monarchy</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cities</th>
<th></th>
<th>Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>country</td>
<td>population</td>
</tr>
<tr>
<td>ANT Antwerp B 455,148</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRU Brussels B 141,312</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OSL Oslo NOR 533,050</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.1: A small portion of the Mondial relational database
high cost of inefficiency (that is, exponential time in the database, even if only two keywords are involved).

So, our first action was to establish a firm theoretical framework for keyword proximity search over database. This framework unifies the goals of the existing work and was published in [7]. Based on this framework, we found some minimal requirements that an engine must satisfy in order to be acceptable. The first requirement is that none of the answers is missed (if the engine runs until termination). However, satisfying this requirement alone is not enough, since the number of answers is usually huge. Thus, the second requirement is that the order in which the answers are produced has some guarantee (e.g., ascending weight or an approximation thereof). But the second requirement is meaningless if efficiency is ignored, so the third requirement is that the answers are generated in an incremental order with (at most) a polynomial-time delay between successive answers (a.k.a. “polynomial delay”).

Notwithstanding the fact that the above three requirements are elementary, we found that none of the existing solutions satisfies all of them (see Appendix for a detailed discussion on these solutions). Thus, during the research, we devised various algorithms (published in [5] [8] [9] [15]) that meet these requirements. Our algorithms differ in the manner and extent of satisfying the requirements. In general, these algorithms run with polynomial delay and, as illustrated in Figure 1.2, cover the range of tradeoff between efficiency and desirability of the order in which answers are generated. Note that ”Heuristically sorted” means that, intuitively, the order is expectedly close to the optimal order but there is no guarantee. In contrast, in “Approximately sorted” (which is a formal definition in the spirit of traditional approximation algorithms), there is a provable bound on the extent to which the actual order can deviate from the optimal one. In [5], for example, we presented an algorithm in the “Approximately sorted” category, along with an implementation, developed by Konstantin Golenberg in his master’s thesis, which showed the practical effectiveness of that algorithm. Other algorithms in this category, as well as the ones for “Sorted” order are presented in Chapter 3.

---

2 Please note that according to the Hebrew University’s rules, the appendix is in a separate booklet.
3 The different types of orders are formally defined in Section 2.2.
1.1.2 Maximal Answers

In keyword proximity search, the underlying assumption is that the database contains the exact real-world data but the user has no knowledge about the data representation. In a sense, the paradigms for handling incomplete and uncertain data take the opposite approach: The user formulates a query that is supposed to exactly retrieve the desired information (e.g., by filling out a form that is translated into SQL), but the data itself has deficiencies, namely, some information may be missing and/or uncertain. The source of such deficiencies can be, for example, the fact that the data is obtained by integrating several uncorrelated sources. The goal is to take the irregular nature of the data into account when querying so that answers will be neither wrong nor missed. Towards this end, various approaches have been proposed where one of them is that of generating maximal (rather than complete) answers [1, 32, 33, 52, 70, 72, 92].

As an example, consider a database containing touristic information in the relations whose schemata are depicted in Figure 1.3. We would like to find a place to visit, based on the information of our database. The natural join of the three relations above may not yield the desired result, since a join may cause interesting data to be lost due to missing information in the tables. For some cities, for example, the database may contain no sites

<table>
<thead>
<tr>
<th>Relation</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Climates</td>
<td>country, climate</td>
</tr>
<tr>
<td>Accommodations</td>
<td>country, city, hotel, stars</td>
</tr>
<tr>
<td>Sites</td>
<td>country, city, site</td>
</tr>
</tbody>
</table>

Figure 1.3: Integrated schema for tourist information
at all. For such cities, there are no corresponding tuples in the Sites relation. As another example, some national sites may be out of any specific city and, hence, the tuples of these sites will not join with any tuple of Accommodations. For dealing with this problem, the full-disjunction operator \[33, 52, 72\] has been proposed.

The full disjunction is an associative and commutative generalization of the outerjoin operator. Hence, it is suitable for integrating any number of relations. Intuitively, for a given set of relations, the full disjunction is obtained by joining maximal sets of related tuples (i.e., tuples that agree on all the common attributes). Since every given tuple is included in at least one tuple of the result, all the available information is preserved.

We developed algorithms for efficiently computing full disjunctions as a stream of tuples. On the theoretical aspect, these are the first algorithms that run with polynomial delay and also the first ones that run in total time that is linear in the size of the output. In his master’s thesis, Itzhak Fadida implemented these algorithms in the open-source PostgreSQL database system and demonstrated their practicality and superiority over the state-of-art algorithms (the algorithms and their implementation are described in [1]). One of these algorithms was later generalized in to the task of enumerating maximal induced subgraphs that satisfy a given hereditary property (a problem that we studied in [3]).

Later, we describe our research on other types of queries with the semantics of maximal answers. In Section 1.1.3 we combine maximal answers with ranking, and in Section 1.1.4 they are combined with uncertainty.

1.1.3 Ranked Querying

In a sense, keyword proximity search and production of maximal answers form two extreme querying paradigms. In the first type, the user provides very limited information (only keywords) regarding the desired information, whereas in the second one she formulates the exact query while the engine makes the best effort in finding matches. An approach in between is that of formulating vague (or flexible) queries based on all of the available knowledge that the user has about the underlying data, and relying on a ranking mechanism to identify the relevant answers. Thus, the next paradigm that we studied is that of ranked
querying, namely, incorporating ranking in traditional query languages.

An important problem that arises in vague queries is the high potential of a huge number of answers. So, to avoid overwhelming the user with meaningless information, the answers need to be given to her in ranked order. The naive way to do just that is to generate all the answers and then appropriately sort them. Clearly, this approach is inefficient since just generating all answers may take a long time (resulting in a bad latency). Moreover, the vaguer and more involved (e.g., longer) the query is, the larger is the set of all answers. Typically, the user is interested in only a few (namely, the top-k) answers, so generating all of them is often highly wasteful. The existing approaches for handling this problem (e.g., \cite{26,59,85}) are heuristics which are not efficient from a theoretical point of view, since, again, the time before producing the top answer (or any answer thereafter) could be exponential.

So, we were interested in devising algorithms that are truly efficient in the sense that they generate the answers in ranked order incrementally while, at each point, invest only the effort that is needed for generating the answers thus far. Our first step was to formalize this efficiency requirement. The usual yardstick of efficiency of query evaluation is polynomial time under data complexity, namely, the query is assumed to be fixed and the input consists of only the database. But, this yardstick does not enable us to distinguish between efficient and inefficient ranked evaluations, since the size of the output is polynomial in that of the input. In particular, the above naive approach is efficient in these terms. Therefore, we employed the following yardstick. An evaluation algorithm is efficient if it generates the answers in ranked order and with polynomial delay under query-and-data complexity (i.e., both the query and the data are given as input). Obviously, the naive approach (as well as each of the existing heuristics) does not have these properties.

We first considered conjunctive queries over relational databases. For this class of queries, query evaluation is intractable under query-and-data complexity \cite{27} but efficient algorithms exist \cite{108} for the class of acyclic conjunctive queries.\footnote{In fact, acyclic conjunctive queries form one of the largest classes for which query evaluation is tractable under query-and-data complexity.} We showed (and published in \cite{10}) that under the above notion of efficiency, there are efficient incremental
evaluations for acyclic conjunctive queries with rather general ORDER BY clauses. More generally, we showed that for a given ranking function over conjunctive queries, such an evaluation exists if and only if finding just the first (i.e., top-1) answer is tractable. Similar results were shown for twig queries with projection over XML graphs, where the need for ranking is more evident than in the usual tree models of XML. The result of that work was a novel query language (published in [11]) for facilitating effective querying of complex XML graphs.

In the above settings, queries can be vague but the answers must satisfy the query in a complete manner. Clearly, when allowing maximal answers, the extent to which an answer is partial is a factor in its relevancy to the user. It is, therefore, highly desirable to combine the paradigms of maximal answers and ranked querying, namely, to produce the maximal answers in ranked order while the ranking function takes into account the degree of incompleteness. Take, for example, the full-disjunction operator. We would like the answers that involve more relations to appear first. Unfortunately, this specific task is generally intractable. The difficulty of this task lies in the cyclic nature of full disjunctions. Therefore, we focused on tree queries and arrived at rather surprising tractability results, as described next.

We developed the first query language (published in [12]) that combines ranking and incompleteness. Our setting is abstract in the sense that queries can be evaluated over general databases, e.g., relational, semistructured, XML, RDF, etc. Moreover, the query language allows for maximal answers but enables the user to specify completeness constraints that control the amount of incompleteness. The highlight is that queries can be evaluated efficiently in ranked order while the ranking function can take into account the following three elements: (1) the quality of the match of each query node, (2) the quality of the relationship between the pair of matched along every edge, and (3) the missing (i.e., null) matches. Practically, this enables the user to naturally combine standard ranking techniques (e.g., IR, uncertainty, editing distance, graph distance, etc.) with the amount of incompleteness.

The flexibility of queries in the above tool is reflected in the abstraction of the notion of a match. For example, the meaning of an edge in the query can be something vague like “the
database objects that correspond to the two endpoints are closely related” rather than “di-
rectly connected.” In the literature, there are other approaches that entail different types of
flexibility. In [30] [71], for instance, the notion of a match of the query in the database ranges
over different levels of specific similarities between the query and the matched fragment. A
different approach is taken in [20] [21] [77] [82], where flexibility is obtained by syntactically
relaxing the user-formulated query. Both of these approaches are limited in the sense that
they consider a fixed set of relaxed alternatives to the interpretation of the query, while our
abstraction allows for arbitrary interpretations and ranking thereof. In addition, in these
approaches it can be highly intricate for the user to understand the relationship between
the answers that she gets and her original query; in our approach it is much easier since
there is a direct relationship between the constraints of the query (nodes and edges) and
the components of an answer. So, in that sense, our approach is more natural and intuitive.

Another approach in the literature is that of [2] [31] [80] where flexibility is obtained
by enabling the user to specify a set of XML tags that correspond to sets of semantically
related XML elements. However, it is up to the system to determine how elements of such
sets are interconnected. Clearly, this specific type of flexibility (under each of the different
specifications of in [2] [31] [80]) can be gracefully incorporated in our tree queries.

1.1.4 Maximal Answers in Uncertain Data

Due to the imprecise nature of modern database applications, there has been a lot of interest,
in the database-research community, in various forms of uncertain data. For example,
in [58] [97], some of the source tuples are annotated with a symbol that indicates that they
are uncertain. The result of evaluating a query over the database consists of the answers in
each possible world. More general annotations are Boolean formulae [67] and polynomials
over commutative semirings [57], which are closed under the query language, namely, the
set of all possible answers itself can be represented as an annotated table.

The notion of probabilistic databases means that data items are associated with prob-
abilistic events and, consequently, the database is a probability distribution over ordinary
(random) instances (often called possible worlds), rather than just a single one. Both re-
ational [22 36 38 51 75 89 93] and XML [18 65 66 86 98 103] models have been studied.

Our first interest in probabilistic databases was in the context of full disjunctions, namely, how this operator can be handled if the database is probabilistic (specifically, we used the model of [36 37 93]). (As far as we know, the natural combination of querying probabilistic data and handling incomplete information has not been studied in the past.)

As an example, consider again the schemata depicted in Figure 1.3. Suppose that a specific answer includes tuples from each of the three relations (i.e., it is actually complete) but the overall probability of this answer is too low to be of any interest. Applying the exact semantics of full disjunctions (while ignoring the probabilities) means that the information contained in this answer is practically lost because a partial answer may have a high probability. For example, we will lose the information in the join of two tuples from Climates and Accommodations even if the probability of this join is high. Thus, there is a tradeoff between complete answers that have rather low probabilities and partial answers that carry less information but have higher probabilities.

So, the first observation is that in probabilistic relations, the full disjunction operator should be applied with respect to some certainty threshold, namely, a probability below which answers are uninteresting. Hence, the goal is to find the maximal of the partial answers that have a probability not lower than the threshold. We found that this problem is inherently different from its two components, i.e., maximal matches in deterministic data and complete matches in probabilistic data. Moreover, though simple to phrase, solving this problem is rather intricate and, in fact, our results were obtained by employing several reductions and a combination of different algorithmic techniques (see Section 1.2). In addition to a thorough complexity analysis, we have developed efficient algorithms (under query-and-data complexity) for maximally joining data in various important tractable cases. (These results were published in [14].) We also investigated the problem of finding the maximal matches of twig patterns in probabilistic XML (specifically, in the model of [86]). We showed that this problem is more tractable than the previous one (in the relational model) by presenting an efficient algorithm for the general case (published in [13]).
1.1.5 Querying Probabilistic XML

As mentioned in the previous section, our work on maximal answers in probabilistic data was the first to study the combination of incompleteness and probabilistic uncertainty. In the case of the relational probabilistic model, the usual semantics of queries (i.e., complete answers) has been extensively studied in recent years (e.g., [22, 36, 38, 51, 75, 89, 93]). The usual notion of querying probabilistic databases means that each answer has a probability, namely, the probability that the answer is obtained when querying a random instance (thus, this probability represents the extent to which the answer is certain). In [36, 37, 93], the focus is on queries with projection. The role of this operator is far more significant in probabilistic data than in deterministic data, as we next illustrate.

Consider again the relations of Figure 1.3. For planning a trip, we look for a site that is located in a city with dry weather and a 5-star hotel. Now, suppose that we phrase the following projection-free query.

**Query 1**

```
SELECT *
FROM Climates C, Accommodations A, Sites S
WHERE C.climate='dry' & C.country=A.country & A.stars='5' & C.city=S.city
```

Suppose that the database contains many 5-star hotels in (which has a dry climate) Jerusalem, but each hotel has a small probability (of being correct). Then, the result of Query 1 contains many tuples for the site of Mount Temple, each of which with a small probability. As a result, we may conclude that this site is not suitable, since none of the results that include Mount Temple is sufficiently certain. But this result is misleading and, actually, Query 1 does not correctly reflect our intention, since we are not inquiring about a specific hotel; we only require that the city has at least one 5-star hotel. The correct query is the following Query 2, which is similar to Query 1 except for projecting on S.site.

**Query 2**

```
SELECT S.site
FROM Climates C, Accommodations A, Sites S
WHERE C.climate='dry' & C.country=A.country & A.stars='5' & C.city=S.city
```
Clearly, the result of Query 2 may indicate a high probability that there is a 5-star hotel in the vicinity of Mount Temple, since there are many low-probability hotels in Jerusalem (and the probability of not seeing any of them in a random instance can be very low). Therefore, though Query 2 is obtained from Query 1 by applying projection, the results of the two queries can lead us to opposite conclusions.

As another indication of the importance of applying projection at the right stage, observe the following. We cannot obtain the result of Query 2 by applying projection to the result of Query 1.

In [36, 37], Dalvi and Suciu studied the complexity of conjunctive queries over probabilistic relational models. They focused on data complexity and showed that the projection operator highly complicates the task of query evaluation and, in fact, the class of tractable queries with projection is very limited. For example, the above Query 1 is tractable but Query 2 is \#P-complete (which is deemed highly intractable [101]). The reason for the difficulty is the fact that applying projection entails the computation of the probability of a disjunction of dependent events.

Probabilistic XML models have also been studied in previous research [18, 65, 66, 86, 98, 103]. Naturally, we were interested in knowing whether the complications introduced by the projection operator carry over to probabilistic XML. However, notwithstanding the thorough understanding that we have on query evaluation in probabilistic relational models, we were surprised to find out that nothing was known about the projection operator in probabilistic XML. So, from that point of the research we have invested a lot of effort in exploring this problem.

The first question that comes to mind is what exactly is probabilistic XML. We observed that the literature contains a plethora of seemingly unrelated models. So, to understand these models, our first step was to devise an abstract model of probabilistic XML documents that we call p-documents. Within this abstraction, concrete families of p-documents include previous models as well as natural extensions and combinations thereof. Then, based on this framework, we analytically compared the expressive power of the concrete families (and in particular, the existing models). Specifically, we found the hierarchy that shows when
an instance of one model can be efficiently translated into an instance of another model.

The next step was to study the complexity of querying. Having in mind the theory of this problem in the relational case, we were amazed by the computational capabilities that exist regarding query evaluation in probabilistic XML. We were able to show that for a model that properly extends most of the existing probabilistic XML models (i.e., [85, 103], the tree models of [65, 66] and the “simple probabilistic trees” of [18]), all twig queries can be evaluated efficiently. These queries include arbitrary node conditions, child and descendant edges, branches and, most importantly, projection. The practicality of the evaluation algorithm is demonstrated in an implementation by Yuri Kosharovsky in his master’s thesis. The importance of this result is that it shows, for the first time, that a general query language with projection is tractable over an interesting model of probabilistic data.

The above model for which query evaluation is tractable is maximal in the sense that for the concrete families that it does not contain, there are practically no tractable queries. More accurately, for these families, we showed that the following dichotomy holds. Every Boolean query is \#P-complete, unless it is extremely (i.e., uninterestingly) trivial. Such a model is the “probabilistic trees” of Abiteboul and Senellart [18, 98]. The extreme intractability lies in the ability of their model to represent probabilistic dependencies among the junctions of uncertainty in the document. It turns out that the tractable models are exactly those which make an inherent assumption of independence. Of course, such an assumption often renders these models too simplistic in practical scenarios. So, we proposed two approaches to circumvent this tradeoff. The first approach is to allow approximate answers, and we showed that approximate query evaluation is tractable in all of the concrete models in our framework. The second approach is that of incorporating constraints.

Constraints are important not only from the traditional database point of view (i.e., for maintaining data integrity), but also from the probabilistic aspect because they capture natural dependencies among data items. A probabilistic XML database (PXDB) is the probability sub-space comprising the instances of a p-document (of a tractable model) that satisfy a set of constraints. Even though complicated dependencies may exist among data
items of a PXDB, we proved that query evaluation (as well as testing well-definedness and sampling) are tractable. Furthermore, queries and constraints can include the aggregate functions count, min, max, and ratio (which captures predicates like “70% of the hotels have 3 stars”). We also showed that this approach can be easily extended to allow a probabilistic interpretation of constraints.

Overall, our results on probabilistic XML show that, unlike the relational case, it is highly tractable. Moreover, compared to existing paradigms for representing probabilistic data (e.g., Bayesian networks [34, 88]), it gives rise to novel database-oriented representations that are intuitive, compact and support efficient querying. Our results on probabilistic XML are published in [4, 6, 13] and are described in detail in Chapters 4 and 5.

1.2 Principal Algorithmic Techniques

Within the studied querying paradigms, various algorithms were devised for the task of query evaluation. These algorithms are, in general, inherently different from one another and each requires some techniques that are relevant solely to the specific task at hand. However, there are also techniques that are more general and were used across different settings and query paradigms. In this section, we give an overview of these techniques.

1.2.1 Partitioning the Output Space and Coroutines

One of the earliest tasks that we explored was that of enumerating all the reduced subtrees of a graph (e.g., answers of keyword proximity search) with polynomial delay [8, 15]. The main technique that was used there (and later in other settings) was that of partitioning the output space. By means of this technique, we solve the enumeration problem at hand by dividing it into multiple sub-problems that form a partition of the set of answers into nonempty subsets. The requirement for non-emptiness is essential for avoiding recursions that take exponential time in the size of the input and the output without producing any answer.

The crux of applying the above technique to a specific task is in generating sub-problems
for which non-emptiness can be tested efficiently. Another problem is that in some tasks (like enumerating reduced subtrees), one needs to combine answers of different sub-problems in order to obtain an answer to the original problem. In these cases, it is impossible to obtain an efficient incremental evaluation if one waits until all the solutions of the sub-problems are generated before constructing the first solution of the original problem, since each sub-problem may have a large number of intermediate answers.

So, partitioning the output space, like other techniques, often provides an evaluation in polynomial total time while it can be quite complicated to transform it into an incremental one, e.g., one that runs with polynomial delay. In some of these cases, polynomial delay is realized by employing coroutines \[83\] which essentially give to an evaluation algorithm the functionality of an iterator. In fact, we used coroutines in order to achieve polynomial delay in all of the problems that we solved by partitioning the output space.

### 1.2.2 Hereditary Graph Properties

A hereditary (resp., connected hereditary) graph property \[78\] is a property that is closed w.r.t. induced (resp., connected induced) subgraphs. Natural enumeration problems can often be described as finding the maximal (connected) induced subgraphs w.r.t. a (connected) hereditary property. One example is the enumeration of all maximal cliques (or independent sets) in a graph (a problem that was studied in \[68\]). Another example is the full-disjunction operator in relational databases. In fact, following our work on this operator, we found that one of the algorithms we devised\(^5\) (and published in \[1\]) can be generalized to arbitrary hereditary properties. Essentially, this algorithm provides a reduction from the original enumeration problem to a restricted version of the problem. Observe that if the input graph itself satisfies the hereditary property, then the above enumeration problem is trivial (namely, the only answer is the whole graph). In the restricted version, the graph almost satisfies the property in the sense that it does so after removing one node. In the case of maximal cliques or full disjunctions, the restricted problem is trivial. However, it is not always the case as we describe next.

\(^5\)In \[3\], we presented various algorithms, including this one, for enumeration problems involving hereditary properties.
The problems of maximally matching twigs in probabilistic XML and maximally joining probabilistic relations (which we studied in [13] and [14], respectively) can also be described as finding maximal induced subgraphs w.r.t. a connected hereditary problem. However, the restricted problem is not at all trivial in these cases. Take, for example, the problem of maximally joining probabilistic relations. In order to solve the restricted problem, we used the technique of the previous section, namely, partitioning the output space (combined with coroutines). Recall that one of the difficulties in applying that technique is to create partitions in a way that testing their non-emptiness is tractable. In the partitions created for this problem, non-emptiness is tractable in important cases, e.g., when the relational schema is $\gamma$-acyclic or a clique. But in general, the non-emptiness problem is NP-complete. One would naturally ask if we could employ a different solution that would not lead to a hard problem. Surprisingly, we found that the answer is negative, i.e., our approach is optimal in the following sense. If the non-emptiness problem (in our way of creating partitions) is intractable (for a given class of schemata), then the original problem (maximally joining probabilistic relations) is intractable as well (formally, unless P=NP, there is no algorithm that solves the original problem in time that is polynomial in the size of the input and the output).

Figure 1.4 illustrates the algorithmic and proof techniques used for maximally joining probabilistic relations as a cycle of reductions. Overall, we showed that the original enumeration problem is computationally equivalent to a natural decision problem. In particular, for distinguishing between the tractable and intractable cases, it is enough to reason about the decision problem (which is the common type of problems for which complexity is analyzed). Based on that, we gave in [14] a thorough analysis of the complexity of maximally joining probabilistic relations. Note that the ability to show intractability of this problem...
is interesting because the usual way of proving that an enumeration problem is intractable (e.g., evaluating conjunctive queries \cite{27}) is to prove that it is intractable to determine whether there is at least one solution. But in our case, determining that is trivial (the answer is “yes” except for extreme cases). As far as we know, no such complexity analysis was previously given in the literature (i.e., an intractable enumeration problem with a tractable decision version).

**Lawler/Yen’s Method**

In \cite{109} \cite{110}, Yen presented an efficient algorithm for finding the \(k\)-shortest simple paths between two given nodes in a graph. Lawler \cite{76} generalized that algorithm to a procedure for computing the top-\(k\) solutions to discrete optimization problems. Essentially, this procedure is a reduction from the problem of finding the top-\(k\) answers to that of finding the top answer under *constraints* (which is an optimization problem in the ordinary sense). Usually, there are two types of constraints. An *inclusion* constraint requires some components to appear in the answer while an *exclusion* constraints requires them to be excluded. This method can be straightforwardly extended to a technique for ranked enumeration of answers with polynomial delay.

Our first interest in this technique was in the context of keyword proximity search, namely, for the task of enumerating the answers by increasing weight. We believed that it can be helpful for obtaining an efficient solution since our problem is a natural generalization of enumerating simple paths by increasing length. However, we found that a naive application of this technique easily leads to intractable problems. In particular, finding the top answer under the generated constraints is NP-hard, even for two keywords. Nevertheless, we devised (in \cite{9}) a method for applying this technique in a way that only tractable constraints are generated. But even then, our solution used (a sequence of) two, rather intricate reductions for finding the top answer under constraints. The algorithm is presented in Chapter \cite{3}.

The algorithm for keyword proximity search described above is efficient only under data complexity. This is necessary since, under query-and-data complexity, finding the top an-
swer is itself an intractable problem, namely, the famous Steiner-tree problem [53]. However, approximate solutions for this problem have been extensively studied (e.g., [28, 61, 112]). Interestingly, we showed (in [9]) that Lawler’s procedure gracefully incorporates approximations in the following sense. If one uses a $\theta$-approximation instead of an optimal solution to the constrained optimization problem, then the enumeration is in a $\theta$-approximate order.

Based on that, we devised efficient approximate algorithms for keyword proximity search. The algorithm we presented in [5] enumerates the answers in an approximate order by height, and our experiments showed that this type of ordering is highly useful in the sense the actual order is very close to increasing weight (while this algorithm is far more efficient than the one enumerates by an approximate increasing weight).

Following the efficient evaluation algorithm that we obtained by using Lawler’s procedure, we were interested in further understanding the potential of this technique for evaluating queries. Surprisingly, we discovered that it is rather simple to adapt it to classic database problems, such as evaluating conjunctive queries with ORDER BY, and ranked evaluation of twig patterns over XML graphs. As a result, we presented (in [10, 11]) the first algorithms that evaluate such queries in ranked order with polynomial delay.

We then went one step further and tried to apply Lawler’s procedure to enumerating maximal answers (in ranked order). Unfortunately, we found that this technique is not easily adapted to this task. Roughly, the main difficulty is that a maximal answer w.r.t. the constraints is not necessarily globally maximal (i.e., maximal regardless of the constraints). The naive application of Lawler’s procedure results in an answer of the first type while we are interested in one of the second type. For example, there does not seem to be any way of adapting this technique to evaluating full disjunctions (regardless of any ranking) without generating intractable constraints (recall that full disjunctions are evaluated efficiently by a technique for hereditary properties). Nevertheless, in [12] we managed to obtain an efficient adaptation in the case of (abstract) tree queries, and for that we applied a set of transformations to the constraints in order to unify the two types of maximality.
Dynamic Programming

Dynamic programming is a well-known technique for efficiently applying a recursion that, when executed naively, takes exponential time. We used this technique (in [4, 6, 13]) for evaluating various types of Boolean queries over different models of probabilistic XML. Consistently, that was done by a bottom-up traversal of the p-document and an evaluation of several logical combinations of branches that were obtained from the query.

The main difficulty in applying this technique is in the transition to the recursive step, namely, evaluating the query at hand based on the previously computed results. For example, evaluating aggregate queries (e.g., count, max and ratio) entails several intricate steps; one of them is a repeated application of eight transformations to the query, where even termination (of these transformations) requires an involved proof. Interestingly, the aggregate operators for which this approach fails (i.e., sum and average) are generally intractable. The details of evaluating queries over probabilistic XML are given in Chapters 4 and 5.

1.3 Thesis Structure

The rest of the thesis is organized as follows. Chapter 2 gives preliminary concepts and definitions. Chapter 3 describes the algorithms for enumerating answers of keyword proximity search by increasing weight and an approximation thereof, where the required Steiner-tree algorithms and background are given in Appendix A. A component of the approximation algorithm is given in Appendix B and Appendix C surveys existing algorithms for keyword proximity search. The p-document model is studied in Chapter 4 along with a comparison of probabilistic XML models in terms of expressive power and query efficiency. In particular, this chapter describes the algorithm for evaluating queries with projection over probabilistic XML. Some of the proofs of the results of this chapter are given in Appendices D and E. Chapter 5 presents the PXDB model that is based on incorporating constraints in p-documents, where the proofs are in Appendices F and G. Chapter 6 is a technical abstract of all the results that were obtained during the doctorate studies. Chapter 7 concludes.
Chapter 2

Preliminaries

This chapter presents the basic concepts and notation that are used throughout the thesis.

2.1 Complexity of Query Evaluation

In this section, we define notions of complexity and efficiency of query evaluation. Traditionally, two measures are used for characterizing the complexity of query evaluation, namely, query-and-data complexity and data complexity.

Query-and-data complexity. Under query-and-data complexity, the input comprises both the query and the database. Under this notion, the size of the output might often be exponential in the input size (e.g., the size of the query is in the exponent). Therefore, polynomial time in the size of the input is not a suitable yardstick of efficiency. In this case, the standard yardstick of an efficiency is polynomial total time \([68]\), that is, the running time is polynomial in the combined size of the input and the output. But even with respect to this yardstick, some elementary database operations are intractable under query-and-data complexity, e.g., evaluation of a join expression over a relational database. This follows from the fact that determining whether this evaluation results in a nonempty relation is an NP-complete problem \([27]\).

Data complexity. While databases are naturally large, queries are typically very small. Therefore, a common measure for the complexity of querying is that of data complexity \([104]\),
namely, the query is fixed and only the database is given as input. Formally, each query is associated with a distinct computational problem—its evaluation over a given database. Thus, polynomial data complexity of a query language means that for each query, its associated problem is solvable in polynomial time. In particular, the query can determine the exponent of the polynomial.

**Parameterized complexity.** In [87], it is argued that the complexity of query evaluation should be studied within the parametric complexity theory [40][41]. This means that the input includes a parameter p and problems are classified according to the dependency of the running time on p. A problem is fixed-parameter tractable (FPT) if the running time is bounded by a function of the form \(g(p) \cdot n^c\), where \(n\) is the size of the input, \(c\) is a constant and \(g\) is any function. In other words, \(p\) may effect the constant of the running-time bound, but not the exponent of the polynomial (i.e., \(c\)). An algorithm with this running time is called an FPT algorithm.

In a query-evaluation problem, \(q\) is typically an integer that represents the size of the query (e.g., the number of variables or terms). So a query language is FPT if a given query can be evaluated over a given database in time \(O(g(q) \cdot d^c)\), where \(q\) and \(d\) and the sizes of the query and the database, respectively. Note that, in contrast, the query language has a polynomial data complexity even if the query evaluation is in time \(O(d^{g(q)})\). Obviously, FPT is stronger than polynomial data complexity. The results of [87] suggest that, in the relational model, the language of conjunctive queries is unlikely to be FPT.

### 2.2 Enumeration and Top-k Answers

An instance of an enumeration problem consists of an input \(x\) and a finite set \(\mathcal{A}(x)\) of answers. An enumeration algorithm \(E\) prints all the answers of \(\mathcal{A}(x)\) without repetitions. We usually assume an underlying scoring function that maps answers to positive real numbers. The score of an answer \(a\) is denoted as \(\text{score}(a)\). Suppose that the algorithm \(E\) enumerates the sequence \(a_1, \ldots, a_n\). If \(\text{score}(a_i) \geq \text{score}(a_j)\) holds for all \(1 \leq i \leq j \leq n\), then the enumeration is in ranked order.
There are two related problems. First, finding an optimal (i.e., top-ranked) answer, that is, printing an answer that has the highest score (or ⊥ if there is no answer at all). Second, finding the top-k answers (or all the answers if they are fewer than k).

Sometimes, for the sake of efficiency, we are interested in approximations. The quality of an approximation is determined by an approximation ratio \( \theta > 1 \) (\( \theta \) may be a function of the input \( x \)). Given an input \( x \), a \( \theta \)-approximation of an optimal answer is any answer \( app \in \mathcal{A}(x) \), such that \( \theta \cdot \text{score}(app) \geq \text{score}(a) \) for all answers \( a \in \mathcal{A}(x) \); if \( \mathcal{A}(x) = \emptyset \), then \( \bot \) is a \( \theta \)-approximation. Following Fagin et al. [44], a \( \theta \)-approximation of the top-k answers is any set \( \text{AppTop} \) consisting of \( \min(k, |\mathcal{A}(x)|) \) answers, such that \( \theta \cdot \text{score}(a) \geq \text{score}(a') \) holds for all \( a \in \text{AppTop} \) and \( a' \in \mathcal{A}(x) \setminus \text{AppTop} \).

Enumerating all the answers in a \( \theta \)-approximate order means that if one answer precedes another, then the first one is at most \( \theta \) times worse than the other one. More formally, the sequence of all answers \( a_1, \ldots, a_n \) is in a \( \theta \)-approximate order if \( \theta \cdot \text{score}(a_i) \geq \text{score}(a_j) \) for all \( 1 \leq i \leq j \leq n \).

### 2.3 Efficiency of Enumeration and Top-k Algorithms

Polynomial time in the size of the input is not a suitable yardstick of efficiency when analyzing an enumeration algorithm, because the output size could be exponential in the input size. In [68], several definitions of efficiency for enumeration algorithms are discussed. The weakest definition is polynomial total time (see Section 2.1). Two stronger definitions consider the time that is needed for generating the \( i \)th element (of the output), after the first \( i-1 \) elements have already been produced. *Incremental polynomial time* means that the \( i \)th element is generated in time that is polynomial in the combined size of the input and the first \( i-1 \) elements. The strongest definition is *polynomial delay*, that is, after the algorithm prints the \((i-1)\)st element, the next one is generated in time that is polynomial only in the input size. In particular, under each of the two definitions, the first answer is generated in polynomial time after the execution of the algorithm begins.

For characterizing space efficiency, we use two definitions. Note that the amount of space
needed for writing the output is ignored—only the space used for storing intermediate results is measured. The usual definition is polynomial space, that is, the amount of space used by the algorithm is polynomial in the input size. Linearly incremental polynomial space means that the space needed for generating the first $i$ elements is bounded by $i$ times a polynomial in the input size. Note that an enumeration algorithm that runs with polynomial delay uses (at most) linearly incremental polynomial space.

We now consider efficiency measures of algorithms for finding the top-$k$ answers within an enumeration problem. An optimal answer is found efficiently if the running time is polynomial in the size of the input $x$. An algorithm for finding the top-$k$ answers is considered efficient if the running time is polynomial in the size of the input $x$ and the value of $k$ (rather than the size of its binary representation). Note that the running time is at least linear in $k$ and, in principle, $k$ may be larger than the size of $x$.

Given an algorithm $E$ that enumerates in ranked order with polynomial delay, we can obtain an efficient algorithm for finding the top-$k$ answers by stopping the execution after generating $k$ answers. In particular, we can use $E$ for finding efficiently an optimal answer. There are two important advantages to such an approach. First, the running time is always linear in $k$ (and polynomial in $x$). Second, $k$ need not be known in advance (hence, the user can decide whether more answers are required, based on the first ones).

Similarly, an algorithm that enumerates in a $\theta$-approximate order with polynomial delay can also find efficiently a $\theta$-approximation of the top-$k$ answers, by stopping the execution after printing $k$ answers. In particular, it can efficiently find a $\theta$-approximation of an optimal answer.

### 2.4 Graphs and Trees

A graph $G$ consists of a set $\mathcal{V}(G)$ of nodes and a set $\mathcal{E}(G)$ of edges. Unless explicitly stated otherwise, edges are directed, that is, an edge is a pair $(n_1, n_2)$ of nodes.

A graph is rooted if it contains some node $r$, such that every node of $G$ is reachable from $r$ through a directed path. The node $r$ is called a root of $G$. (Note that a rooted data graph
may have several roots.) A data graph is connected if its underlying undirected graph is connected.

We say that a data graph $G'$ is a subgraph of the data graph $G$, denoted by $G' \subseteq G$, if $\mathcal{V}(G') \subseteq \mathcal{V}(G)$ and $\mathcal{E}(G') \subseteq \mathcal{E}(G)$.

For a data graph $G$ and a subset $U \subseteq \mathcal{V}(G)$, we denote by $G - U$ the induced subgraph of $G$ that consists of the nodes of $\mathcal{V}(G) \setminus U$ and all the edges between these nodes. If $u \in \mathcal{V}(G)$, then we may write $G - u$ instead of $G - \{u\}$.

If $G_1$ and $G_2$ are subgraphs of $G$, we use $G_1 \cup G_2$ to denote the subgraph that consists of all the nodes and edges that appear in either $G_1$ or $G_2$; that is, $G_1 \cup G_2$ is the subgraph $G'$ such that $\mathcal{V}(G') = \mathcal{V}(G_1) \cup \mathcal{V}(G_2)$ and $\mathcal{E}(G') = \mathcal{E}(G_1) \cup \mathcal{E}(G_2)$. The operation $G_1 \cup (u, v)$ adds the edge $(u, v)$ and the nodes $u$ and $v$ to $G_1$ (if they are not already there).

**Trees, Subtrees and Reduced Subtrees**

We use two types of trees. A directed tree is a rooted graph, such that there is exactly one root and for every node $u$, there is a unique path from the root to $u$. The root of a directed tree $T$ is denoted by $\text{root}(T)$. An undirected tree is a data graph that is connected and has no cycles, when ignoring the directions of the edges.

A leaf of a directed tree is a node without children (i.e., without outgoing edges), whereas in an undirected tree, a leaf is a node with one incident edge (or zero if the tree consists of a single node). We use $\text{leaves}(T)$ to denote the set of all leaves of a tree $T$.

Consider a graph $G$. Subtrees of $G$ are special cases of subgraphs and we use three types of them: directed subtrees (abbr. $d$-subtrees), undirected subtrees (abbr. $u$-subtrees) and root-subtrees (abbr. $r$-subtrees). The third type is defined only if $G$ is rooted and, in that case, it is a d-subtree, such that $\text{root}(T)$ is a root of $G$.

Consider a graph $G$ and subset $U$ of its nodes. A $d$-subtree (resp., $u$-subtree, $r$-subtree) $T$ is reduced w.r.t. $U$ if $T$ contains $U$ but no proper $d$-subtree (resp., $u$-subtree, $r$-subtree) of $T$ contains $U$.  

\footnote{Observe that this is an abuse of notation, since a tree can be both directed and undirected, in which case the definition of a leaf depends on the type that is assigned to the tree. Nevertheless, the type should always be clear from the context.}
Chapter 3

Efficient Engines for Keyword Proximity Search

In this chapter, we introduce the paradigm of keyword proximity search over data graphs within a formal framework. Then, we present efficient algorithms for enumerating answers by the exact and an approximate order of increasing weight. In the first type of enumeration, we necessarily assume that the number of keywords is fixed while in the second, no such assumption is made. In other words, data complexity is used in the first result whereas query-and-data complexity in the second. A survey of the existing algorithms for keyword proximity search in given in Appendix C.

3.1 Formal Framework

This section describes a formal framework for keyword proximity search over data graphs.

3.1.1 Data Graphs

A data graph is a graph with two types of nodes: structural nodes and keyword nodes (or keywords for short). \( \mathcal{S}(G) \) denotes the set of structural nodes of the data graph \( G \) and \( \mathcal{K}(G) \) denotes the set of keywords of \( G \). Keywords have only incoming edges, while structural nodes may have both incoming and outgoing edges. In other words, \( \mathcal{E}(G) \subseteq \)
\( \mathcal{S}(G) \times \mathcal{V}(G) \). In particular, an edge cannot connect two keywords. The edges of a data graph \( G \) may have *weights*. The weight function \( w_G \) assigns a positive weight \( w_G(e) \) to every edge \( e \in \mathcal{E}(G) \). The weight of the data graph \( G \), denoted by \( w(G) \), is the sum of the weights of all the edges of \( G \), that is, \( w(G) = \sum_{e \in \mathcal{E}(G)} w_G(e) \).

Both relational and XML data can be represented as graphs. When representing an XML document as a graph, the XML elements correspond to structural nodes. The edges represent either element nesting or ID references (which must be taken into account when answering queries because they represent semantic relationships). The text is represented by a keyword node for each word \( w \). For all structural nodes \( v \) that correspond to XML elements containing \( w \), there is an edge from \( v \) to \( w \). In the case of relational data, the structural nodes correspond to tuples. The edges represent two types of information. First, there are edges from tuples to the keywords that they contain. Second, there is an edge from each tuple that includes a foreign key to the tuple that has that key value. Note that in our model, each keyword is represented by a single node and all its occurrences point to that node (hence, this is not a restriction).

For example, consider the data graph \( G_1 \) depicted in Figure 3.1. Filled circles represent structural nodes and keywords are written in typewriter font.

### 3.1.2 Queries and Answers

A *query* is simply a finite set \( Q \) of keywords. Given a data graph \( G \), a *directed \( Q \)-fragment* (abbr. DQF) is a \( d \)-subtree of \( G \) that is reduced w.r.t. \( Q \). Similarly, an *undirected \( Q \)-fragment* (abbr. UQF) is a \( u \)-subtree of \( G \) that is reduced w.r.t. \( Q \). A *strong \( Q \)-fragment* (abbr. SQF) is a UQF, such that all the keywords are leaves. Note that a DQF is also an SQF (since keywords have no outgoing edges), and an SQF is also a UQF.

**Example 3.1.1** Consider again the data graph \( G_1 \) of Figure 3.1. This figure also depicts four subtrees \( F_1, \ldots, F_4 \) of \( G_1 \). Let \( Q_1 \) be the query \{Belgium, Brussels, EU\} and \( Q_2 \) be the query \{Belgium, Norway\}. \( F_1 \) and \( F_2 \) are directed \( Q_1 \)-fragments of \( G_1 \). The \( Q_2 \)-fragment \( F_3 \) is strong but not directed. The \( Q_2 \)-fragment \( F_4 \) is undirected but neither
If $Q$ contains only one keyword, then there is exactly one $Q$-fragment, namely, the one consisting of the single keyword of $Q$. Naturally, we are interested in the case where $Q$ has at least two keywords and in the sequel, this assumption is always implicitly made. In that case, the following gives an alternative characterization of $Q$-fragments that is important for understanding the ideas discussed in this paper.

**Observation 3.1.2** Let $G$ be a data graph and $Q$ be a query of two or more keywords.

1. A d-subtree $T$ of $G$ is a $DQF$ if and only if $\text{leaves}(T) = Q$ and the root of $T$ has at least two children.

2. A u-subtree $T$ of $G$ is a $UQF$ if and only if $\text{leaves}(T) \subseteq Q \subseteq \mathcal{X}(T)$.

3. A u-subtree $T$ of $G$ is an $SQF$ if and only if $\mathcal{X}(T) = \text{leaves}(T) = Q$. 
3.2 The Formal Results

For simplification of the presentation, this and the following sections discuss only DQFs. The generalization of the results to SQFs and UQFs is discussed in Section 3.7.

In the sequel, we use the following notation for a data graph $G$ and a query $Q$. The number of nodes and edges of $G$ are denoted by $n$ and $e$, respectively. By $q$ we denote the number of keywords in $Q$ and $n_i$ is the number of nodes in the $i$th generated DQF. Note that there are at most $2^e$ DQFs, i.e., $i \leq 2^e$.

The first theorem that is proved in this chapter is that all the DQFs can be enumerated by the order of increasing weight. The delay is polynomial under data complexity and even under fixed-parameter complexity (where, as usual, the size of $Q$ is the parameter). Note that enumeration by the order of increasing weight cannot be done with polynomial delay under query-and-data complexity, or else P=NP. This holds since, under query-and-data complexity, just finding the top answer is itself an intractable problem, namely, the directed Steiner-tree problem [53].

**Theorem 3.2.1** Given a data graph $G$ and a query $Q$, all DQFs can be enumerated in the order of increasing weight, where the $i$th delay is $O(n_i(4^qn + 3^qe \log n))$.

**Corollary 3.2.2** Under data complexity, all DQFs are be enumerated in ranked order with polynomial delay and the top-$k$ can be efficiently computed.

The next theorem considers approximate enumeration with polynomial delay under query-and-data complexity. In particular, it shows that every approximation algorithm for directed Steiner trees (i.e., an algorithm for approximating the top DQF) can be extended to an enumeration in an approximate order, with almost the same approximation ratio.

Given an approximation algorithm for Steiner trees, $f$ denotes its running time and $\theta$ denotes its approximation ratio. We assume that both $f$ and $\theta$ are functions of the numbers of nodes and edges of the given graph, and the number of keywords in the query. Moreover, we assume that $f$ and $\theta$ are monotonically nondecreasing functions of their arguments; that is, if $n_1 \leq n_2$, $e_1 \leq e_2$ and $q_1 \leq q_2$, then $f(n_1, e_1, q_1) \leq f(n_2, e_2, q_2)$ and $\theta(n_1, e_1, q_1) \leq \theta(n_2, e_2, q_2)$.

28
Enumerating DQFs in a ranked order

Algorithm DQFSearch (Sections 3.3 and 3.4)

Finding a minimal DQF under PF constraints

Algorithm Min-Fragment (Section 3.6)

Finding a minimal $G$-supertree

Algorithm Min-Supertree (Section 3.5)

Finding a directed Steiner tree

Figure 3.2: Ranked enumeration of DQFs: The chain of reductions

**Theorem 3.2.3** If a $\theta$-approximation of the minimal DQF can be found in time $f$, then all DQFs can be enumerated in a $(\theta + 1)$-approximate ranked order, where the $i$th delay is $O(n_i(f + e \log n))$.

**Corollary 3.2.4** If a $\theta$-approximation of the minimal DQF can be found efficiently, then all DQFs are be enumerated in a $(\theta + 1)$-approximate order with polynomial delay and a $(\theta + 1)$-approximation of the top-$k$ can be efficiently computed.

In the remainder of this chapter, we prove Theorems 3.2.1 and 3.2.3 by presenting the corresponding enumeration algorithms. The algorithm for enumerating by increasing weight is actually a chain of three reductions, as illustrated in Figure 3.2. The meaning of the terms inside the dotted rectangles is described in the corresponding sections. As described in the sequel, enumeration in an approximate order is done by a similar chain of reductions, while each of the tasks in the rectangles is solved in an approximate manner (hence different algorithms are used).

### 3.3 The Basic Algorithm

Lawler [76] generalized an algorithm of Yen [109, 110] to a procedure for computing the top-$k$ solutions of discrete optimization problems. The algorithm $\text{EnumDQFs}$ of Figure 3.3...
Algorithm EnumDQFs(G, Q)

1: Queue ← an empty priority queue
2: F ← Fragment(G, Q, ∅, ∅)
3: if F ≠ ⊥ then
4: Queue.insert((∅, ∅, F))
5: while Queue ≠ ∅ do
6: ⟨I, E, F⟩ ← Queue.remove-top()
7: let \{e_1, \ldots, e_k\} = \partial(F) \setminus I
8: for i = 1 to k do
9: I_i ← I ∪ \{e_1, \ldots, e_{i-1}\}
10: E_i ← E ∪ \{e_i\}
11: F_i ← Fragment(G, Q, I_i, E_i)
12: if F_i ≠ ⊥ then
13: Queue.insert((I_i, E_i, F_i))
14: print(F)

Figure 3.3: The basic algorithm for enumerating DQFs

is an adaptation of Lawler’s procedure to enumerating all DQFs, given a query Q and a
data graph G. In this algorithm, a constraint is an edge. A DQF F satisfies a set I of
inclusion constraints and a set E of exclusion constraints if it has all the edges of I and
none of the edges of E. The algorithm employs a subroutine Fragment(G, Q, I, E) that
returns a DQF of G that satisfies both I and E. If no such DQF exists, then ⊥ is returned.
The following sections describe two options for implementing this subroutine. One returns
a minimal (i.e., optimal) DQF satisfying I and E and the other returns an approximation
of such a DQF. We now describe how EnumDQFs works, assuming that Fragment returns a
minimal DQF.

The algorithm EnumDQFs uses a priority queue Queue. An element in Queue is a triplet
⟨I, E, F⟩, where I and E are sets of inclusion and exclusion constraints, respectively, and F
is the top-ranked DQF that satisfies both I and E. Priority of ⟨I, E, F⟩ in Queue is based
on the score of the DQF F; that is, the top element of Queue is a triple ⟨I, E, F⟩, such
that score(F) is maximal. The operation Queue.remove-top() removes the top element and
returns it to the caller. We assume that operations on the priority queue take logarithmic
time in the size of Queue, i.e., polynomial time in the size of the input.

The algorithm EnumDQFs(G, Q) starts by finding, in Line 2, a DQF \( F \) that satisfies
the empty sets of inclusion and exclusion constraints. Thus, \( F \) is the top-ranked DQF. In
Line 4, the triplet \( (\emptyset, \emptyset, F) \) is inserted into Queue. In the main loop of Line 5, the top
triplet \( (I, E, F) \) is removed from Queue in Line 6. In principle, the algorithm should add
to Queue the second DQF, in ranked order, among all those satisfying both \( I \) and \( E \). But
instead of doing that, the algorithm partitions the set comprising all the DQFs satisfying \( I \)
and \( E \), except for the top-ranked DQF \( F \), and adds the top-ranked DQF of each partition
to Queue. The partitioning is done as follows. Let \( e_1, \ldots, e_k \) be the edges of \( F \) that are not
in \( I \). The \( i \)th partition \((1 \leq i \leq k)\) is obtained by adding the first \( i-1 \) edges \( e_1, \ldots, e_{i-1} \) to
\( I \) and adding the \( i \)th edge \( e_i \) to \( E \), thereby creating the new sets of constraints \( I_i \) and \( E_i \),
respectively. In Line 11, these sets are given as arguments to Fragment and the top-ranked
DQF \( F_i \) that satisfies \( I_i \) and \( E_i \) is returned. In Line 13, \( (I_i, E_i, F_i) \) is added to Queue,
provided that \( F_i \neq \bot \). Note that \( F_1, \ldots, F_k \) are all different from \( F \), since they satisfy
constraints that \( F \) itself does not satisfy. The DQF \( F \) is printed at the end of the iteration,
in Line 14.

3.3.1 Correctness and Efficiency

We now show the correctness and efficiency of EnumDQFs(G, Q); that is, we prove that it
enumerates all DQFs of \( G \) and the delay and the order of the enumeration are determined by
the implementation of Fragment. The following theorem, which is proved in the remainder
of this section, gives the formal result. Recall that the parameters \( n, e, q \) and \( n_i \) are defined
in Section 3.2.

**Theorem 3.3.1** Consider a data graph \( G \) and a query \( Q \).

1. If Fragment\((G, Q, I, E)\) returns a \( \theta(n, e, q) \)-approximation of the minimal DQF satis-
fying \( I \) and \( E \), then EnumDQFs\((G, Q)\) enumerates all DQFs in a \( \theta(n, e, q) \)-approximate
ranked order.
2. If $\text{Fragment}(G, Q, I, E)$ terminates in $T(n, e, q)$ time, then $\text{EnumDQFs}(G, Q)$ enumerates with $O(n_i \cdot T(n, e, q) + n_i \cdot \log(n \cdot i))$ delay.

Let $I$ and $E$ be sets of inclusion and exclusion constraints, respectively. By $\mathcal{F}_{I,E}(G, Q)$ we denote the set of all DQFs of $G$ that satisfy both $I$ and $E$ (that is, DQFs $F$ that contain all the edges of $I$ and none of the edges of $E$). In particular, observe that $\mathcal{F}_{\emptyset, \emptyset}(G, Q)$ is the set of all DQFs of $G$ and we simply denote it as $\mathcal{F}(G, Q)$.

Consider an iteration of Lines 6–14 during some execution of $\text{EnumDQFs}(G, Q)$. The following lemma shows that when this iteration begins, the sets $\mathcal{F}_{I,E}(G, Q)$ that correspond to the elements of Queue cover all the DQFs that have not yet been printed and they are mutually disjoint.

**Lemma 3.3.2** Consider the beginning of one of the iterations of Lines 6–14 during an execution of $\text{EnumDQFs}(G, Q)$. Suppose that at that time, $\text{InQueue}$ is the set of all tuples in Queue and $\text{Printed}$ is the set of all DQFs that have already been printed. Then,

1. $\bigcup \{ \mathcal{F}_{I,E}(G, Q) \mid \exists F ((I, E, F) \in \text{InQueue}) \} = \mathcal{F}(G, Q) \setminus \text{Printed}$.

2. If $(I, E, F)$ and $(I', E', F')$ are distinct elements of $\text{InQueue}$, then $\mathcal{F}_{I,E}(G, Q)$ and $\mathcal{F}_{I',E'}(G, Q)$ are disjoint.

**Proof.** Consider the first iteration of Lines 6–14. The lemma clearly holds for that iteration since when it begins, $\text{Printed}$ is empty and $\text{InQueue}$ consists of a single element of the form $(\emptyset, \emptyset, F)$. We complete the proof of the lemma by showing that if it holds in some iteration, then it also holds in the subsequent iteration. So consider an iteration of Lines 6–14 and suppose that both Parts 1 and 2 of the lemma hold when it begins. We will show that they hold when that iteration ends (i.e., relative to the updated content of $\text{Printed}$ and $\text{InQueue}$).

Let $(I, E, F)$ be the tuple that is chosen in Line 6 and $(I_i, E_i, F_i)$ be the tuples that are generated in Lines 8–13 ($i = 1, \ldots, k$). From the way tuples are constructed by the algorithm (and the correctness of $\text{Fragment}$), it follows that $F \in \mathcal{F}_{I,E}(G, Q)$. During the considered iteration of Lines 6–14, the tuple $(I, E, F)$ is replaced, in Queue, with the tuples
that satisfy \( \mathcal{F}_{I, E}(G, Q) \neq \emptyset \) (Lines 12–13). Moreover, \( F \) is printed (Line 14).

Since the lemma holds at the beginning of the iteration, it is enough to prove that the set \( \{ \mathcal{F}_{I, E}(G, Q) \mid 1 \leq i \leq k \} \) forms a cover of \( \mathcal{F}_{I,E}(G, Q) \setminus \{ F \} \) by mutually disjoint sets. This is done next.

First, observe that \( I \subseteq I_i \) and \( E \subseteq E_i \) for all \( 1 \leq i \leq k \). Consequently, each \( \mathcal{F}_{I, E}(G, Q) \) is a subset of \( \mathcal{F}_{I, E}(G, Q) \). Second, if \( 1 \leq i < j \leq k \), then \( \mathcal{F}_{I, E}(G, Q) \) and \( \mathcal{F}_{I, E}(G, Q) \) are disjoint since \( E_i \) and \( I_j \) are contradictory, namely, the first requires the exclusion of \( e_i \) while the second requires its inclusion. So it is left to show that every DQF \( F' \in \mathcal{F}_{I, E}(G, Q) \setminus \{ F \} \) is contained in some \( \mathcal{F}_{I, E}(G, Q) \). So let such \( F' \) be given. Then \( I \subseteq \mathcal{E}(F') \) and \( E \cap \mathcal{E}(F') = \emptyset \). Let \( 0 \leq l \leq k \) be the maximal number, such that \( I \cup \{ e_1, \ldots, e_l \} \subseteq \mathcal{E}(F') \). Note that \( l < k \) must hold, or otherwise \( F' \) is not reduced since it properly contains \( F \) (and we assume that \( F' \neq F \)). Then \( E \cup \{ e_{l+1} \} \cap \mathcal{E}(F') = \emptyset \). So for \( i = l + 1 \), it holds that \( 1 \leq i \leq k \), and \( F' \in \mathcal{F}_{I, E} \), as claimed. \( \square \)

From Part 1 of Lemma 3.3.2 and Lines 5, 6 and 14 of EnumDQFs, we conclude the following.

**Corollary 3.3.3** EnumDQFs\((G, Q)\) enumerates \( \mathcal{F}(G, Q) \).

The next lemma completes the proof of Part 1 of Theorem 3.3.1 by showing that the approximation ratio of the enumeration is the same as that of Fragment.

**Lemma 3.3.4** Suppose that Fragment\((G, Q, I, E)\) returns a \( \theta(n, e, q)\)-approximation of the minimal DQF satisfying \( I \) and \( E \). Then EnumDQFs\((G, Q)\) enumerates in a \( \theta(n, e, q)\)-approximate ranked order.

**Proof.** Let \( F \) and \( \hat{F} \) be two DQFs and suppose that \( F \) is printed before \( \hat{F} \). We need to show that \( w(F) \leq \theta(n, e, q) \cdot w(\hat{F}) \). So consider the iteration of Lines 6–14 in which \( F \) is printed. By Part 1 of Lemma 3.3.2, we conclude that, at the beginning of that iteration, Queue contains a tuple \( \langle I', E', F' \rangle \), such that \( \hat{F} \in \mathcal{F}_{I', E'}(G, Q) \). Since \( F' \) is generated by executing Fragment\((G, Q, I', E')\), it follows that \( w(F') \leq \theta(n, e, q) \cdot w(\hat{F}) \). But then, from the priority that is used in Queue and the fact that \( F \) is chosen in Line 6 (rather than \( F' \)), we conclude that \( w(F) \leq w(F') \). The lemma then immediately follows. \( \square \)
We conclude this section by analyzing the running time of \textsc{EnumDQFs}(G, Q) and, by that, complete the proof of Theorem 3.3.1. Observe that the \(i\)th delay is dominated by the execution cost of the iteration of Lines 6–14 in which the \(i\)th DQF is printed in Line 14, i.e., the \(i\)th iteration. In each iteration of Lines 6–14, at most \(n\) elements are inserted into \texttt{Queue}, so in the \(i\)th iteration, \texttt{Queue} contains at most \(n \cdot i\) elements and each execution of Line 6 or 13 has an \(O(\log(n \cdot i))\) cost. The execution cost of Lines 7 and 14 is \(O(n_i)\). Finally, each of Lines 9 and 10 has a constant execution cost, when implementing the sets of constraints \(E_i\) and \(I_i\) as linked lists (e.g., \(I_2\) is obtained by concatenating \(I_1\) and \(e_1\) and \(E_2\) is obtained by concatenating \(E\) and \(e_2\)). We conclude that the execution cost of each iteration of Lines 9–13 is \(O(T(n, e, q) + \log(n \cdot i))\) and that of the rest of Lines 6–14 is \(O(\log(n \cdot i) + n_i)\). Part 2 of Theorem 3.3.1 then immediately follows.

### 3.4 Generating Tractable Constraints

#### 3.4.1 Intractability of General Constraints

By theorem 3.3.1, the task at hand is to efficiently implement \texttt{Fragment}, namely, to find a minimal (or approximately minimal) DQF satisfying a set of inclusion and exclusion constraints. Moreover, we can obtain an enumeration of the DQFs with polynomial delay by simply devising an algorithm that finds a DQF that satisfies the given constraints. However, the following proposition shows that even the latter task is intractable.

**Proposition 3.4.1** It is NP-complete to determine whether \(F_{I,E}(G, Q) \neq \emptyset\), even if \(|Q| = 2\), \(E = \emptyset\) and \(I\) contains only one edge.

**Proof.** For membership in NP, observe that verifying membership in \(F_{I,E}(G, Q)\) of a subtree of \(G\) is in polynomial time. NP-hardness (under the given assumptions) can be proved straightforwardly by using the following intractable case of the \textit{subgraph-homeomorphism} problem \cite{47}: Given a directed graph \(G\) and three nodes \(s\), \(t\) and \(v\), determine whether \(G\) contains a simple directed path \(p\) from \(s\) to \(t\), such that \(p\) visits \(v\). \(\Box\)
By Proposition 3.4.1, the hardness of finding a DQF that satisfies a set of constraints lies in the inclusion constraints. This is not surprising, since handling a set $E$ of exclusion constraints is easy and can be done by simply deleting the edges of $E$ from the data graph $G$. Formally, this means that $\mathcal{F}_{I,E}(G, Q) = \mathcal{F}_{I,\emptyset}(G', Q)$, where $G'$ is obtained from $G$ by removing all the edges of $E$. Hence, in the reminder of this chapter, we ignore the exclusion constraints and assume that there are only inclusion constraints.

To circumvent the difficulty of handling inclusion constraints, we modify the algorithm EnumDQFs so that instead of generating arbitrary sets of inclusion constraints, it only generates tractable ones, namely, partial fragments.

### 3.4.2 Partial Fragments

Let $G$ be a data graph. Given a set $\mathcal{T}$ of directed subtrees of $G$, a $G$-supertree of $\mathcal{T}$ is a subtree of $G$ that contains all of the subtrees of $\mathcal{T}$. A partial fragment (PF) is any directed subtree $T$ of $G$, such that all the leaves of $T$ are keywords (that is, $\text{leaves}(T) = \mathcal{K}(T)$). A set of PFs $\mathcal{U}$ is called a set of PF constraints if the PFs in $\mathcal{U}$ are pairwise node disjoint. A subtree $T$ of $G$ satisfies $\mathcal{U}$ if it is a $G$-supertree of $\mathcal{U}$.

**Observation 3.4.2** EnumDQFs$(G, Q)$ can be executed so that for every generated set $I$ of inclusion constraints induces a set of two or fewer PF constraints.

As an illustration, consider an iteration of Lines 6–14 of EnumDQFs. In Line 6, the tuple $(I, E, F)$ is chosen and suppose that $F$ is the DQF of Figure 3.4(a) while $I$ consists of the edges that are surrounded by the polygon with the dashed edges. Note that $I$ is a set of 2 PF constraints, namely, $T_1$ and $T_2$. The order of the remaining edges, $e_1, \ldots, e_7$, is such that every generated set of inclusion constraints $I_i = I \cup \{e_1, \ldots, e_{i-1}\}$ is itself a set of two or fewer PF constraints. As $e_i$, we choose an edge of $\mathcal{E}(F) \setminus I_i$ that enters the root of one of the PFs induced by $I \cup \{e_1, \ldots, e_{i-1}\}$; if no such edge exists (that is, $I_i$ induces one PF and the root is that of $F$) then $e_i$ is an arbitrary edge of $\mathcal{E}(F) \setminus I_i$ among those that enter a keyword.
There are two types of PFs. A reduced PF (RPF) has a root with at least two children whereas a nonreduced PF (NPF) has a root with only one child. As an example, in Figure 3.4(a) $T_1$ is an NPF while $T_2$ is an RPF. We emphasize that, as a special case, a single keyword node is considered an RPF. Note that an RPF, but not an NPF, is reduced w.r.t. the set of its leaves (i.e., its keywords). We use $U_{rpf}$ and $U_{npf}$ to denote the set of all the RPFs of $U$ and the set of all the NPFs of $U$, respectively.

### 3.4.3 Rephrasing the Problem

By Observation 3.4.2 (and the fact that exclusion constraints are ignored), the subroutine Fragment should find a DQF that satisfies a given set $U$ of PF constraints. For simplification, we eliminate the query $Q$ by adding to $U$, as a single-node RPF, every keyword of $Q$ that is not already in $U$. Thus, from now on, when looking for a DQF that satisfies a set $U$ of PF constraints, we assume that all the keywords of $Q$ appear in $U$ (hence $Q$ need not be given as part of the input of Fragment). Note that the number of PFs of $U$ is at most the number of keywords in $Q$. By Observation 3.4.2 we can assume that at most two of those PFs are NPFs (and the rest are RPFs).

Suppose that $T$ is a $G$-supertree of $U$. Note that, although $T$ contains all the keywords of $Q$, it is not necessarily a DQF since the root of $T$ may have a single child (see Part I of
Observation 3.1.2. Furthermore, if the root belongs to some NPF $N \in \mathcal{V}_{npf}$, then it cannot be deleted in order to obtain a DQF (since the outgoing edge of the root is a constraint).

As an example, Figure 3.4(b) shows the data graph $G_2$ and two PFs, $T_3$ and $T_4$, that are surrounded by dotted polygons. Let $Q$ be the query \{A, B, C, D\} and $\mathcal{V}$ be the set of PF constraints \{T_3, T_4\}. The minimal $G_2$-supertree of $\mathcal{V}$ comprises the nodes and edges of $T_3$ and $T_4$ as well as the edge $(v_4, v_5)$. Note, however, that this supertree is not a DQF, since its root $v_1$ has only one child. It can be shown that the only DQF that satisfies $\mathcal{V}$ is obtained by choosing $r$ to be the root, connecting $r$ to $T_3$ using the path $u_1 \rightarrow \cdots \rightarrow u_6 \rightarrow v_1$ and connecting $r$ to $T_4$ with the edge $(r, v_5)$. As another example, suppose that $T_5$ is the subtree of $G_2$ that is obtained by adding the edge $(v_0, v_1)$ to $T_3$ and let $\mathcal{V}' = \{T_4, T_5\}$. Although there is a $G_2$-supertree of $\mathcal{V}'$, no DQF of $G$ satisfies $\mathcal{V}'$.

The above example shows that finding a $G$-supertree is different from finding a DQF. Nevertheless, as implied in Figure 3.2, our algorithms for finding DQFs under PF constraints require finding minimal supertrees and approximations thereof. In the next section, we show how it is done.

### 3.5 Computing Supertrees

In this section, we consider a data graph $G$ and a set $\mathcal{T}$ of d-subtrees of $G$. We give algorithms for finding a minimal $G$-supertree of $\mathcal{T}$ and an approximation thereof. Both algorithms are based on an algorithm for finding a reduced $G$-supertree $T$ of $\mathcal{T}$, namely, $T$ is a d-subtree of $G$ that includes all the subtrees of $\mathcal{T}$ and has no proper subtree that also includes these subtrees. It should be emphasized that the algorithms of this section are later applied even when the set $\mathcal{T}$ does not consist of PFs, that is, some of the leaves in the d-subtrees of $\mathcal{T}$ may be structural.

Consider a data graph $G$ and a set $\mathcal{T}$ of d-subtrees of $G$. A necessary (but not sufficient) condition that a $G$-supertree of $\mathcal{T}$ exists is the following. For every two subtrees $T_1$ and $T_2$ of $\mathcal{T}$, either $T_1$ and $T_2$ are node disjoint (i.e., $\mathcal{V}(T_1) \cap \mathcal{V}(T_2) = \emptyset$) or the union $T_1 \cup T_2$ forms a d-subtree of $G$ (and consequently, we can replace $T_1$ and $T_2$, in $\mathcal{T}$, with their
Therefore, we can assume that the given set \( \mathcal{T} \) consists of pairwise node-disjoint (abbr. p.n.d.) d-subtrees of \( G \).

Finding a reduced \( G \)-supertree of \( \mathcal{T} \) is a three-step process. First, \( G \) is transformed into a new data graph \( \hat{G} \) by collapsing each subtree of \( \mathcal{T} \) into a single node. Second, in the new graph \( \hat{G} \), we find a reduced subtree \( \hat{T} \) w.r.t. the nodes that correspond to the subtrees of \( \mathcal{T} \). Third, the subtrees of \( \mathcal{T} \) are restored in \( \hat{T} \), so that the result is a tree. We start by describing the collapse and restore operations.

### 3.5.1 Collapsing and Restoring Subtrees

Consider a data graph \( G \) and a set \( \mathcal{T} \) of p.n.d. d-subtrees of \( G \). We delete from \( G \) all edges \( e \), such that \( e \) enters a non-root node of some subtree \( T \in \mathcal{T} \) and \( e \) is not an edge of \( T \). The reason for deleting these edges is that they cannot be included in any \( G \)-supertree of \( \mathcal{T} \). Thus, from now on we assume that if an edge of \( G \) enters a non-root node of some \( T \in \mathcal{T} \), then that edge belongs to \( T \).

The data graph \( \text{cllps}(G, \mathcal{T}) \) is the result of collapsing all the d-subtrees of \( \mathcal{T} \) in \( G \) and it is obtained as follows.

- Delete from \( G \) all the non-root nodes of subtrees of \( \mathcal{T} \); that is, replace \( G \) with \( G - \bigcup_{T \in \mathcal{T}} (\mathcal{V}(T) \setminus \{\text{root}(T)\}) \). (Note that the roots of the subtrees of \( \mathcal{T} \) remain in the collapsed graph, because these subtrees are pairwise node disjoint.)

- For all deleted edges \((u, v)\), such that \( u \) is a non-root node of a subtree \( T \in \mathcal{T} \) but \((u, v)\) is not an edge of \( T \), add the edge \((\text{root}(T), v)\) to \( \text{cllps}(G, \mathcal{T}) \). (Note that if \( v \) is in another tree \( T' \) of \( \mathcal{T} \), then \( v \) is necessarily the root of \( T' \).) The weight of the edge \((\text{root}(T), v)\) is the minimum among the weights of all the edges of \( G \) from nodes of \( T \) to \( v \). In particular, if \((\text{root}(T), v)\) is already an edge of \( G \), then its weight is decreased if there is another edge with a smaller weight from a non-root node of \( T \) to \( v \).

As an example, the top part of Figure 3.16 shows how two node-disjoint subtrees \( T_1 \) and \( T_2 \) are collapsed. These subtrees are surrounded by dotted polygons. In this figure, weights of edges are not specified and we assume that they are equal. (Note that the edges \((v_3, B)\) and
Algorithm RSupertree($G, \mathcal{T}$)

1: $\hat{G} \leftarrow \text{cllps}(G, \mathcal{T})$
2: $U \leftarrow \{\text{root}(T) \mid T \in \mathcal{T}\}$
3: if $\exists v \in \mathcal{V}(\hat{G})$ s.t. each $r \in U$ is reachable from $v$ then
4: $\hat{T} \leftarrow \text{RSubtree}(\hat{G}, U)$
5: return $rstr(\hat{T}, \mathcal{T}, G)$
6: else
7: return ⊥

Figure 3.5: Finding a reduced $G$-supertree of $\mathcal{T}$

$(v_4, v_6)$ are actually deleted before collapsing $\mathcal{T} = \{T_1, T_2\}$, according to the construction described above.)

Next, we describe the operation of restoring the subtrees of $\mathcal{T}$ in a subgraph $\hat{H}$ of cllps($G, \mathcal{T}$). The data graph $rstr(\hat{H}, \mathcal{T}, G)$ is obtained as follows.

- We add all the edges and nodes of $\mathcal{T}$ to $\hat{H}$.

- Consider each subtree $T$ of $\mathcal{T}$. Each edge $(\text{root}(T), v)$ of $\hat{H}$ is replaced with $(u, v)$, where $(u, v)$ has the minimal weight among all edges of $G$ from nodes of $T$ to $v$. If there are several such nodes $u$, an arbitrary one is chosen. Note that according to this definition, the edge $(\text{root}(T), v)$ is left unchanged if $u = \text{root}(T)$.

As an example, the bottom part of Figure 3.6 shows how the collapsed subtrees $T_1$ and $T_2$ are restored in $\hat{T}$.

3.5.2 Generating Reduced Supertrees

Consider a data graph $G$ and a set $\mathcal{T}$ of p.n.d. d-subtrees of $G$. A reduced $G$-supertree of $\mathcal{T}$ can be obtained as follows. First, collapse all the subtrees of $\mathcal{T}$. Note that only the roots of the subtrees of $\mathcal{T}$ remain in the resulting graph $\hat{G}$. Next, find a subtree $\hat{T}$ of $\hat{G}$ that is reduced w.r.t. the set $U$ comprising the roots of the subtrees of $\mathcal{T}$. Finally, restore the subtrees of $\mathcal{T}$ in $\hat{T}$ to obtain a reduced $G$-supertree of $\mathcal{T}$. The algorithm RSupertree($G, \mathcal{T}$)
of Figure 3.5 formally describes this process. Note that in Line 3, a simple test is performed in order to determine whether $\hat{G}$ contains a reduced d-subtree w.r.t. $U$. If the test is positive, then such a tree $\hat{T}$ is obtained by executing the procedure $\text{RSubtree}(\hat{G}, U)$ in Line 4. Later, in Section 3.5.3 two options for implementing this procedure are discussed.

For illustration, Figure 3.6 depicts the graph $G$ and the d-subtrees $T_1$ and $T_2$. This figure shows an execution of $\text{RSupertree}$ for the input consisting of $G$ and $T = \{T_1, T_2\}$. In the first step, $\hat{G}$ is obtained from $G$ by collapsing the subtrees $T_1$ and $T_2$. The second step constructs the subtree $\hat{S}$ of $\hat{G}$ that is reduced w.r.t. the set of roots $\{v_1, v_5\}$. Finally, in the third step, $T_1$ and $T_2$ are restored in $\hat{T}$ and the result is returned.

The next lemma shows the correctness of $\text{RSupertree}$. We use the following notation. Let $G_1$ and $G_2$ be two data graphs. We use $G_1 \preceq G_2$ to denote that $G_1$ and $G_2$ are identical, except for the weights of the edges that can be larger in $G_2$. Formally, $G_1 \preceq G_2$ if $\mathcal{N}(G_1) = \mathcal{N}(G_2)$, $\mathcal{I}(G_1) = \mathcal{I}(G_2)$, $\mathcal{E}(G_1) = \mathcal{E}(G_2)$ and $w_{G_1}(e) \leq w_{G_2}(e)$ for all edges $e \in \mathcal{E}(G_1)$.

Lemma 3.5.1 Let $G$ be a data graph, $\mathcal{T}$ be a set of p.n.d. d-subtrees of $G$ and $U = \{\text{root}(T) \mid T \in \mathcal{T}\}$. The following hold.

1. If $\hat{T}$ is a d-subtree of $\text{cllps}(G, \mathcal{T})$ that is reduced w.r.t. $U$, then $\text{rstr}(\hat{T}, \mathcal{T}, G)$ is a reduced $G$-supertree of $\mathcal{T}$.

2. If $T$ is a reduced $G$-supertree of $\mathcal{T}$, then $\text{cllps}(G, \mathcal{T})$ contains a d-subtree $\hat{T}$, such that $\hat{T}$ is reduced w.r.t. $U$, and $\hat{T} \preceq \text{cllps}(T, \mathcal{T})$.

Proof. We start with Part 1. Let $G' = \text{rstr}(\hat{T}, \mathcal{T}, G)$ and $\hat{r} = \text{root}(\hat{T})$. The following observations can be proved straightforwardly from the definition of the restore operation. First, in $G'$ all the nodes are reachable from $\hat{r}$. Second, the in-degree of each node of $\hat{T}$ is the same as its in-degree in $G'$, and this is also the case for out-degree of each node of $\mathcal{V}(\hat{T}) \setminus U$. Third, each non-root node of a subtree of $\mathcal{T}$ has exactly one incoming edge in $G'$, namely, the one that appears in $\mathcal{T}$.

From the above it follows that in $G'$, the node $\hat{r}$ has no incoming edges and each of the other nodes is reachable from $\hat{r}$ and has one incoming edge. This proves that $G'$ is a
d-subtree of $G$ with the root $\hat{r}$. To show that $G'$ is a reduced $G$-supertree of $T$, it suffices to prove that (1) Each leaf of $G'$ belongs to a subtree of $T$, and (2) Either $\hat{r} \in U$ or $\hat{r}$ has at least two outgoing edges in $G'$. Both of these claims follow from the second observation above, as follows. For the first claim, if a leaf $v$ of $G'$ does not belong to a subtree of $T$, then $v$ is also a leaf of $\hat{T}$, which contradicts the fact that $\hat{T}$ is reduced w.r.t. $U$. For the second claim, if $\hat{r} \notin U$, then it has more than one child in $\hat{T}$ and, consequently, also in $G'$.

We now prove Part 2. Let $W$ be the set of the nodes of $G$ that do not appear in $T$. Let $\hat{H} = \text{cllp}(T, \mathcal{T})$ and $\hat{G} = \text{cllp}(G, \mathcal{T})$. Consider an edge $e = (v, u)$ of $T$, such that $v$ belongs to a subtree $T'$ of $T$ and $u \in W \cup U$. Then $\hat{H}$ contains the edge $(\text{root}(T'), u)$ with the weight $w_G(e)$. By the definition of the collapse operation, $\hat{G}$ also contains the edge $(\text{root}(T'), u)$ and the weight of this edge in $G$ is at most $w_G(e)$, since $G$ also contains $e$. It follows that each of the edges of $\hat{H}$ appears in $G$ with a similar or smaller weight. So we naturally choose $\hat{T}$ to be the subgraph of $G$ that consists of the edges that correspond to those of $\hat{H}$. To complete the proof, it suffices to prove that $\hat{T}$ is a d-subtree of $\hat{G}$ that is reduced w.r.t. $U$. We omit the details of that proof since it is similar to Part 1; namely, we first show that $\hat{T}$ is a tree with the root $r$ of $T$, then show that $r$ is either in $U$ or has at least two children in $\hat{T}$ and conclude by showing that each leaf of $\hat{T}$ is in $U$ (or otherwise $T$ is not reduced).

3.5.3 Approximate and Minimal Supertrees

We assume that $\text{Approx-DSteiner}(\hat{G}, U)$ is an existing algorithm that accepts as input a data graph $\hat{G}$, with $\hat{n}$ nodes and $\hat{e}$ edges, and a set $U$ of $t$ nodes of $\hat{G}$. It returns a $\theta(\hat{n}, \hat{e}, t)$-approximation of a directed Steiner tree for $U$ in $\hat{G}$ and runs in $f(\hat{n}, \hat{e}, t)$ time. As discussed in Section 3.2, we assume that both $\theta$ and $f$ are monotonically non-decreasing functions.

The algorithm $\text{Approx-Supertree}(G, T)$ is obtained from $\text{RSupertree}(G, T)$ of Figure 3.5 by using $\text{Approx-DSteiner}(\hat{G}, U)$, in Line 4, instead of $\text{RSubtree}(\hat{G}, U)$. The following lemma shows that $\text{Approx-Supertree}(G, T)$ returns a $\theta(n, e, t)$-approximation of the minimal $G$-
Figure 3.6: Execution example of RSupertree

supertree of $\mathcal{I}$. Recall that $\theta$ is assumed to be a monotonically non-decreasing function of its arguments.

**Theorem 3.5.2** Let $G$ be a data graph with $n$ nodes and $e$ edges, and let $\mathcal{I}$ be a set of $t$ p.n.d. $d$-subtrees of $G$. Approx-Supertree$(G, \mathcal{I})$ returns a $G$-supertree that is reduced w.r.t. $\mathcal{I}$ and is a $\theta(n, e, t)$-approximation of the minimal $G$-supertree of $\mathcal{I}$. If no $G$-supertree of $\mathcal{I}$ exists, then $\perp$ is returned. The running time is $O(te + f(n, e, t))$.

**Proof.** From Lemma 3.5.1 it follows immediately that if $G$ does not contain a $G$-supertree of $\mathcal{I}$, then Approx-Supertree$(G, \mathcal{I})$ returns $\perp$. So suppose that $G$ contains a $G$-supertree of $\mathcal{I}$ and let $T_m$ be one with a minimal weight. Let $\hat{G} = \cllsps(G, \mathcal{I})$ and $\hat{T}_m = \cllsps(T_m, \mathcal{I})$. Part 2 of Lemma 3.5.1 shows that $\hat{G}$ contains a subtree $\hat{T}_m'$, such that $\hat{T}_m' \preceq \hat{T}_m$. The numbers of nodes and edges of $\hat{G}$ are at most as those of $G$. It follows from the monotonicity of $\theta$ that the subtree $\hat{T}$ that is returned in Line 4 of Approx-Supertree$(G, \mathcal{I})$ has a weight of at most $\theta(n, e, t)$ times the weight of $\hat{T}_m'$. From Part 1 of Lemma 3.5.1 we conclude that the output of Approx-Supertree$(G, \mathcal{I})$ generated in Line 5, $\rstr(\hat{T}_m', \mathcal{I}, G)$, is a reduced $G$-supertree of $\mathcal{I}$. So let $T = \rstr(\hat{T}, \mathcal{I}, G)$ and we need to show that the weight of $T$ is
at most $\theta(n, e, t)$ times the weight of $T_m$. This is done by the following analysis. Denote by $w(\mathcal{T})$ the sum of the weights of the subtrees of $\mathcal{T}$ (that is, the total weight of the edges that appear in $\mathcal{T}$).

$$w(T) = w_G(T) + w(\mathcal{T}) \leq \theta(n, e, t) \cdot w(\hat{T}_m) + w(\mathcal{T}) = \theta(n, e, t) \cdot (w(T_m) - w(\mathcal{T})) + w(\mathcal{T}) \leq \theta(n, e, t) \cdot w(T_m)$$

as claimed.

To prove the specified running time, note that one can collapse all the subtrees of $\mathcal{T}$ by a single traversal over the edges of $G$, and that the test of Line 3 of \textbf{RSupertree} can be done by $t$ traversals (e.g., depth-first search) over $\hat{G}$.

As a special case of \textbf{Approx-DSteiner}(\hat{G}, U), the existing algorithm \textbf{DSteiner}(\hat{G}, U) returns a directed Steiner tree for $U$ in $G$ (i.e., its approximation ratio is 1) and its running time is $O(3^t \hat{n} + 2^t \hat{e} \log \hat{n})$. This algorithm is described in Appendix A. The algorithm Min-Supertree($G$, $\mathcal{T}$) is obtained by using \textbf{DSteiner}(\hat{G}, U), in Line 4 of Figure 3.5 instead of \textbf{RSubtree}(\hat{G}, U). We then get the following corollary of Theorem 3.5.2.

**Corollary 3.5.3** Let $G$ be a data graph of $n$ nodes and $e$ edges, and $\mathcal{T}$ be a set of $t$ p.n.d. d-subtrees of $G$. Min-Supertree($G$, $\mathcal{T}$) returns a minimal $G$-supertree of $\mathcal{T}$ if one exists, or ⊥ otherwise. The running time is $O(3^t n + 2^t e \log n)$.

### 3.6 Minimal Constrained Answers

In this section, we show how to find efficiently a minimal DQF that satisfies a given set of PF constraints, assuming that the size of the query $Q$ is bounded (recall that, otherwise, the problem is intractable). The algorithm Min-Fragment($G$, $\mathcal{U}$) of Figure 3.7 accepts as input a data graph $G$ and a set $\mathcal{U}$ of PF constraints. As said earlier, we assume that the PFs of $\mathcal{U}$ include all the keywords of $Q$. The algorithm returns a minimal DQF that satisfies $\mathcal{U}$. If Min-Fragment is used instead of Fragment in Figure 3.3 then EnumDQFs enumerates all the DQFs in ranked order with polynomial delay. (Recall that the arguments of Fragment
in Lines 2 and 11 of EnumDQFs are converted to those of Min-Fragment, as described in Section 3.4.3. Next, we describe the algorithm Min-Fragment \((G, \mathcal{U})\).

If \(\mathcal{U}\) contains only one PF, then the problem is easily solved in Lines 1–4. So in the remainder of this section, we assume that \(\mathcal{U}\) contains at least two PFs. Recall that we cannot simply apply the algorithm Min-Supertree of Section 3.5.3 since the result might be a non-reduced \(G\)-supertree of \(\mathcal{U}\). The reason is that the root of the resulting \(G\)-supertree may have only one child if the edge connecting the root to its child is a constraint (that is, an edge of one of the NPFs of \(\mathcal{U}\)). As discussed at the end of Section 3.4.2 even if we find a minimal \(G\)-supertree \(T\) of \(\mathcal{U}\), a minimal DQF that satisfies \(\mathcal{U}\) may be very different from \(T\) or may not exist at all.

One of the key ideas deployed by the algorithm is as follows. When executing Min-Supertree \((G, \mathcal{U})\), if we add at least one of the following two requirements to the generated \(G\)-supertree \(T\), then it is guaranteed that \(T\) is not only a \(G\)-supertree, but actually a DQF:

- **Requirement 1**: The root of \(T\) is the root of a specific RPF \(T_r \in \mathcal{U}_{\text{rpf}}\).
  - This requirement can be handled by simply removing the incoming edges of root\((T_r)\) from \(G\) before calling Min-Supertree \((G, \mathcal{U})\).

- **Requirement 2**: None of the roots of the PFs of \(\mathcal{U}\) is a descendant of a non-root node of an NPF of \(\mathcal{U}\).
  - To handle this requirement, we remove from \(G\) and \(\mathcal{U}\) all the non-root nodes of the NPFs of \(\mathcal{U}\) before calling Min-Supertree \((G, \mathcal{U})\). Finally, the NPFs of \(\mathcal{U}\) are added to the result.

The reason that Requirement 1 guarantees that \(T\) is a DQF is the fact the root of an RPF has at least two outgoing edges in that RPF. As for Requirement 2, we use the fact that \(\mathcal{U}\) has at least two PFs. Consequently, if root\((T)\) is the root of an NPF of \(\mathcal{U}\), then root\((T)\) has at least two children, or otherwise the roots of the rest of the PFs of \(\mathcal{U}\) are all descendants of the child of root\((T)\).

Note that each of the processes described above for handling the requirements results
in a minimal DQF among those satisfying $\mathcal{U}$ and fulfilling the corresponding requirement. However, the minimal DQFs do not necessarily fulfill any of the two requirements, so when applied to given input, both those processes may result in a non-minimal DQF. Moreover, it may be the case that there are DQFs, yet each of them fulfills none of the two requirements. Nevertheless, we next show how a combination of the two solves the problem.

Basically, our approach consists of two main steps. In the first step, we choose a subset $\mathcal{U}_{\text{top}}$ of two or more PFs of $\mathcal{U}$ and construct a minimal $G$-supertree $T_{\text{top}}$ of $\mathcal{U}_{\text{top}}$ w.r.t. Requirement 2. We also require that $T_{\text{top}}$ has none of the nodes of the PFs of $\mathcal{U} \setminus \mathcal{U}_{\text{top}}$, and manage this requirement by first removing these nodes from $G$. Let $Q'$ be the set of keywords that appear in $\mathcal{U}_{\text{top}}$. Then $T_{\text{top}}$ is a directed $Q'$-fragment, hence an RPF. So in the second step, we obtain a new set $\mathcal{U}'$ of PF constraints by replacing in $\mathcal{U}$ the PFs of $\mathcal{U}_{\text{top}}$ with $T_{\text{top}}$. Finally, we produce a minimal $G$-supertree $F$ of $\mathcal{U}'$ w.r.t. Requirement 1, where $T_{\text{top}}$ plays the role of the PRF $T_r$ in that requirement (hence $F$ is a DQF). Later, we prove that there is at least one choice of the set $\mathcal{U}_{\text{top}}$ (if there is at least one DQF), such that the above two steps are guaranteed to generate a minimal DQF satisfying $\mathcal{U}$. But we do not know $\mathcal{U}_{\text{top}}$ in advance, so we repeat the two steps for each possible $\mathcal{U}_{\text{top}}$ (i.e., set of two or more PFs of $\mathcal{U}$). The result of the algorithm is the minimal generated $F$. Recall that the number of PFs of $\mathcal{U}$ is at most the (bounded) number of keywords of $Q$, so our algorithm is efficient.

The loop over the sets $\mathcal{U}_{\text{top}}$ is in Line 6 where, in each iteration, Lines 7–15 are executed. In Lines 7–12, the tree $T_{\text{top}}$ is constructed as follows. First, the graph $G_1$ is obtained from $G$ by removing all the PFs of $\mathcal{U}$ except for those of $\mathcal{U}_{\text{top}}$. All the non-root nodes of the NPFs of $\mathcal{U}_{\text{top}}$ are removed from $G$ as well. The set $\mathcal{T}$ is obtained from $\mathcal{U}_{\text{top}}$ by replacing each NPF $N$ with the one-node subtree that consists of root($N$) only. Having constructed $G_1$ and $\mathcal{T}$, Line 9 generates a minimal $G_1$-supertree of $\mathcal{T}$, namely, the tree $T$. If $T$ does not exist, then the iteration of Line 6 continues. Otherwise, we obtain $T_{\text{top}}$ from $T$ by adding the NPFs of $\mathcal{U}_{\text{top}}$ (since $T$ contains only the roots of these NPFs). As explained above, $T_{\text{top}}$ is an RPF. In Lines 13–14, the DQF $F$ is constructed as a minimal $G_2$-supertree of $\mathcal{U}' = (\mathcal{U} \setminus \mathcal{U}_{\text{top}}) \cup \{T_{\text{top}}\}$, where $G_2$ is obtained from $G$ by removing the incoming edges
Algorithm Min-Fragment\((G, \mathcal{U})\)

1: if \(\mathcal{U}\) consists of a single RPF \(T\) then
2: \hspace{1em} return \(T\)
3: if \(\mathcal{U}\) consists of a single NPF then
4: \hspace{1em} return \(\perp\)
5: \(F_{\text{min}} \leftarrow \perp\)
6: for all subsets \(\mathcal{U}_{\text{top}}\) of \(\mathcal{U}\), such that \(|\mathcal{U}_{\text{top}}| \geq 2\) do
7: \hspace{1em} \(G_1 \leftarrow G - \bigcup_{N \in \mathcal{U}_{\text{npf}}} (\mathcal{V}(N) \setminus \{\text{root}(N)\}) - \bigcup_{T \in \mathcal{U} \setminus \mathcal{U}_{\text{top}}} \mathcal{V}(T)\)
8: \hspace{1em} \(T_{\text{top}} \leftarrow \text{Min-Supertree}(G_1, \mathcal{U}_{\text{top}})\)
9: \hspace{1em} if \(T_{\text{top}} = \perp\) then
10: \hspace{2em} continue
11: \hspace{1em} \(G_2 \leftarrow G\) after removing all the incoming edges of root\((T_{\text{top}})\)
12: \hspace{1em} \(F \leftarrow \text{Min-Supertree}(G_2, (\mathcal{U} \setminus \mathcal{U}_{\text{top}}) \cup \{T_{\text{top}}\})\)
13: \hspace{1em} Update \(F_{\text{min}}\)
14: \hspace{1em} \(F_{\text{min}} \leftarrow \min\{F_{\text{min}}, F\}\)
15: \(\) return \(F_{\text{min}}\)

Figure 3.7: Finding a minimal DQF under a set of PF constraints

of root\((T_{\text{top}})\) (thereby imposing Requirement 1 for \(T_r = T_{\text{top}}\)). The variable \(F_{\text{min}}\), which is returned when the algorithm terminates, stores in Lines 15 the minimal \(F\) among those constructed in the iterations.

Suppose that \(G\) contains at least one DQF that satisfies \(\mathcal{U}\). We now prove that there is at least one choice of \(\mathcal{U}_{\text{top}}\), such that in the corresponding iteration of Lines 7–15, a DQF \(F\) is generated and its weight is minimal. In the remainder of this section, we use the following notation. \(F^m\) denotes an arbitrary DQF of a minimal weight. By \(\mathcal{U}^m_{\text{top}}\) we denote the set of all PFs \(T \in \mathcal{U}\), such that the path of \(F^m\) from root\((F^m)\) to root\((T)\) does not include any non-root node of an NPF of \(\mathcal{U}_{\text{npf}}\). It can easily be shown that \(|\mathcal{U}^m_{\text{top}}| \geq 2\) or otherwise \(F^m\) is not reduced. Next, we prove that \(\mathcal{U}^m_{\text{top}}\) is a “good” choice for \(\mathcal{U}_{\text{top}}\). But first, we need
the following lemma.

**Lemma 3.6.1** Let $G_0$ be a data graph, $\mathcal{T}_0$ be a set of p.n.d. d-subtrees of $G_0$ and $T_0$ be a d-subtree of $G_0$. Denote by $N_0$ the set of all the non-root nodes of the d-subtrees of $\mathcal{T}_0$. If the following conditions hold, then there is a $G_0$-supertree of $\mathcal{T}_0 \cup \{T_0\}$ with the root $\text{root}(T_0)$.

1. All the roots of the subtrees of $\mathcal{T}_0$ are reachable from $\text{root}(T_0)$.
2. $\text{root}(T_0) \notin N_0$.
3. Every node $v \in N_0$ has exactly one incoming edge in $G_0$.

**Proof.** Let $G^m$ be a minimal subgraph $G'$ of $G_0$ among those satisfying the following two conditions. (1) $G'$ contains all subtrees of $\mathcal{T}_0 \cup \{T_0\}$, and (2) all the roots of the subtrees of $\mathcal{T}_0$ are reachable from $\text{root}(T_0)$. We will show that $G^m$ is the required $G_0$-supertree. For that, it is enough to show that $G^m$ is a d-subtree of $G_0$ with the root of $T_0$.

First, observe that $\text{root}(T_0)$ has no incoming edges in $G^m$ since such edges are not in $\mathcal{T}_0$ (by Condition 2) and clearly not in $T_0$, and these edges are not required for reachability from $\text{root}(T_0)$. Moreover, $\text{root}(T_0)$ is a root of $G^m$ since nodes that are not reachable from $\text{root}(T_0)$ are redundant (observe that all the nodes that appear in $\mathcal{T}_0 \cup \{T_0\}$ are reachable from $\text{root}(T_0)$ in $G^m$).

So it is left to show that each node other than $\text{root}(T_0)$ has exactly one incoming edge. So let $v \neq \text{root}(T_0)$ be a node in $G^m$. Suppose, by way of contradiction, that $v$ has two or more incoming edges. We will show that at least one of these edges can be removed from $G^m$, thereby contradict the fact that $G^m$ is minimal. Note that an incoming edge $e$ of $v$ can be removed if $e$ does not belong to any of the d-subtrees of $\mathcal{T}_0 \cup \{T_0\}$ and, in addition, $v$ is reachable from $\text{root}(T)$ even without using $e$. Due to Condition 3 of the lemma, $v$ is not a non-root node of a subtree of $\mathcal{T}_0$, so none of its incoming edges belongs to a subtree of $\mathcal{T}_0$. If $v$ is a node of $T_0$, then only one of the incoming edges of $v$ belongs to $T_0$, and the rest can be removed (since $T_0$ contains a path from $\text{root}(T_0)$ to $v$). Finally, if $v$ is in not in
T\_0, then consider a simple path of G\^m from root(T\_0) to v. Only one incoming edge of G\^m participates in this path, hence all other incoming edges of v can be removed.

\[\text{Lemma 3.6.2} \quad \text{Consider an execution of Min-Fragment}(G, U). In the iteration of Lines 7–15 with } U_{\text{top}} = U_{\text{top}}^m, \text{ Line 14 produces a minimal DQF satisfying } U.\]

\[\text{Proof.} \quad \text{The proof considers only the iteration of Lines 7–15 where } U_{\text{top}} = U_{\text{top}}^m. \text{ The above description of the algorithm contains a proof of the following claim. If Line 14 is reached and } F \neq \bot, \text{ then } F \text{ is a DQF. So it is left to prove the following. First, the tree } T \text{ generated in Line 9 is non-null. Second, } G_2 \text{ contains a } G\text{-supertree } F' \text{ of } U' = (U \setminus U_{\text{top}}) \cup \{T_{\text{top}}\}, \text{ such that } w(F') \leq w(F^m).\]

Consider the set \( \mathcal{T} \) that is constructed in Line 8 and let \( T^m \) be the minimal r-subtree of \( F^m \) that contains \( \mathcal{T} \). By the definition of \( U_{\text{top}}^m \), the d-subtree \( T^m \) does not contain any edge or non-root node of an NPF of \( U_{\text{npf}} \). Moreover, \( U_{\text{top}}^m \) does not contain any root (or any other node) of a PF of \( U \setminus U_{\text{top}}^m \), or otherwise that PF must be in \( U_{\text{top}}^m \). It follows that \( T^m \) is a d-subtree of \( G_1 \) (constructed in Line 7). We conclude that, after Line 9, \( T \) is non-null and, moreover, \( w(T) \leq w(T^m) \).

Let \( G_0 \) the union of \( T \) and all the edges of \( F^m \) that are not in \( T^m \). Note that \( G_0 \) contains all the PFs of \( U' \) (including \( T_{\text{top}} \), since none of the edges of the NPFs of \( U_{\text{top}}^m \) is in \( T^m \)). Since \( w(T) \leq w(T^m) \), we get that \( w(G_0) \leq w(F^m) \). So to conclude the proof, we will show that there is a \( G_0 \)-supertree of \( U' \). For that, we use Lemma 3.6.1 with \( \mathcal{T}_0 = U \setminus U_{\text{top}}^m \) and \( T_0 = T_{\text{top}} \). Since \( T_{\text{top}} \) is a \( G_1 \)-supertree of \( U_{\text{top}}^m \), Condition 2 of the lemma holds. Condition 3 holds since in both \( T \) and \( F^m \), a non-root node of a PF of \( U \) does not have an incoming edge other than that of the PF. It is left to prove that Condition 1 holds, namely, the roots of all the PFs of \( U \setminus U_{\text{top}}^m \) are reachable from root(\( T \)). So let \( T' \in U \setminus U_{\text{top}}^m \) be given. Let \( P \) be the path of \( F^m \) from root(\( F^m \)) to root(\( T' \)). Since \( T' \notin U_{\text{top}}^m \), the path \( P \) contains an edge of an NPF of \( U \). Let \( e = (v, u) \) be the first such edge in the path \( P \). Then \( v \) is necessarily the root of an NPF \( N \). Moreover, \( N \in U_{\text{top}}^m \) and, thus, \( v \) belongs to \( T \). Finally, none of the edges in the suffix of \( P \), from \( v \) to root(\( T' \)), is in \( T^m \). We conclude that this suffix is a part of \( G_0 \), so the latter contains a path from root(\( T \)) to root(\( T' \)), as claimed. \( \square \)
The following theorem concludes this section by showing the correctness of Min-Fragment and stating its running time. Note that Theorem 3.2.1 follows as a direct corollary, since \( p \leq q \).

**Theorem 3.6.3** Consider a data graph \( G \) with \( n \) nodes and \( e \) edges. Let \( Q \) be a query with \( q \) keywords and \( \mathcal{U} \) be a set of \( p \) PF constraints, such that \( \text{leaves}(\mathcal{U}) = Q \). Min-Fragment\((G, \mathcal{U})\) returns either a minimal DQF that satisfies \( \mathcal{U} \), or \( \bot \) if there is no such DQF. The running time is \( \mathcal{O}(4^p n + 3^p e \log n) \).

**Proof.** In the above description of the algorithm, it is proved that if the algorithm returns a value that is not \( \bot \), then that value is a DQF. In other words, if there is no DQF satisfies \( \mathcal{U} \), then \( \bot \) is returned. So suppose otherwise. From Lemma 3.6.2 and Line 15, it follows that the algorithm returns a minimal DQF that satisfies \( \mathcal{U} \).

The running time of Min-Fragment\((G, \mathcal{U})\) is dominated by the two calls to Min-Supertree in Lines 9 and 14. For each subset \( \mathcal{U}_{\text{top}} \subseteq \mathcal{U} \) of size \( k \), Min-Supertree is called once with the set \( \mathcal{T} \) (that is also of size \( k \)) and once with the set \( \mathcal{U} \setminus \mathcal{U}_{\text{top}} \cup \{T_{\text{top}}\} \) which has the size \( p - k + 1 \). By Corollary 3.5.3 an upper bound on the total cost of Lines 9 and 14 is \( \mathcal{O}(g(n, e, p)) \), where

\[
g(n, e, p) = \sum_{k=2}^{p} \binom{p}{k} \left( 3^k n + 2^k e \log n + 3^{p-k+1} n + 2^{p-k+1} e \log n \right) \leq \sum_{k=0}^{p} \binom{p}{k} 3^k + 3e \log n \sum_{k=0}^{p} \binom{p}{k} 2^k = 4n4^p + 3e(\log n)3^p.
\]

The specified runtime then follows immediately. \( \square \)

Similarly to Min-Fragment, the algorithm Approx-Fragment accepts as input the data graph \( G \) and a set of PF constraints \( \mathcal{U} \), such that \( \text{leaves}(\mathcal{U}) = Q \). This algorithm finds a \((\theta + 1)\)-approximation of a minimal DQF that satisfies \( \mathcal{U} \), given that Approx-DSteiner finds a \( \theta \)-approximation of a directed Steiner tree. This algorithm runs in polynomial time under query-and-data complexity, i.e., without assuming that \( Q \) is of a fixed size. Due to a lack of space, this algorithm and the proof of its correctness and efficiency (which imply Theorem 3.2.3) are given in Appendix B.
3.7 Adaptation to UQFs and SQFs

The algorithms described in this chapter are used for enumerating all DQFs in two different orders, namely, the ranked order and an approximation thereof. For enumerating either all UQFs or all SQFs, we use a similar approach. In particular, the basic algorithm of Figure 3.3 does not change for the new tasks. The rest of the components of the algorithms are reused, yet most of them need to be adapted to each of the new types of $Q$-fragments. For instance, new notions of partial fragments, collapse and restore need to be formulated.

As another example, we make use of algorithms for undirected and group Steiner trees instead of directed Steiner trees.

**Theorem 3.7.1** Given a data graph $G$ and a query $Q$, either all UQFs or all SQFs can be enumerated in the order of increasing weight, where the $i$th delay is $O(n_i(4^q n + 3^e \log n))$. Consequently, under data complexity, either all SQFs or all UQFs can be enumerated in ranked order with polynomial delay and the top-$k$ can be efficiently computed.

Interestingly, one can avoid the above adaptation of the components of the algorithm by solving the problem of finding a minimal (or approximate) $Q$-fragment under a set of constraints (that is, Lines 2 and 11 of EnumDQFs) through a rather straightforward reduction to the directed case (i.e., enumerating DQFs as presented in this chapter). This approach is sufficient for proving Theorem 3.7.1. But it has a significant disadvantage when enumerating in an approximate order, namely, it does not enable us to utilize better approximation ratios of algorithms for group and undirected Steiner trees. In particular, we need the adaptation described above to prove the following results, which conclude this section.

**Theorem 3.7.2** If a $\theta$-approximation of the minimal UQF (SQF) can be found in time $f$, then all UQFs (SQFs) can be enumerated in a $(\theta + 1)$-approximate ranked order, where the $i$th delay is $O(n_i(f + e \log n))$. Thus, all UQFs (SQFs) are be enumerated in a $(\theta + 1)$-approximate order with polynomial delay and a $(\theta + 1)$-approximation of the top-$k$ answers can be efficiently computed.
Chapter 4

Probabilistic XML: Expressiveness and Query Evaluation

In this chapter, we present the abstract p-document model of probabilistic XML. Within this model, we explore several families of p-documents which are natural extensions and combinations of previously studied models. We first compare the expressive power of these families, namely, the ability to translate a p-document of one family into another. Then, we study the problem of evaluating queries over these families. In particular, we present an efficient algorithm for evaluating twig queries over one family and show that this family is maximal in the sense that in non-subsumed families every query is intractable (except for extremely trivial ones). Nevertheless, we present efficient randomized algorithms for approximate query evaluation in the maximal family.

4.1 Preliminaries: Trees and Documents

We represent both (probabilistic) data and queries by directed and unordered trees. Throughout the current and the following chapters, tree and subtree implicitly mean directed tree and d-subtree, respectively (see Section 2.4 for our graph and tree notation). Next, we describe some notation about trees.

Let $T$ be a tree. If $(n_1, n_2) \in \mathcal{E}(T)$, then $n_2$ is a child of $n_1$, which in turn is the parent
of $n_2$. Suppose that there is a path from node $n_1$ to node $n_2$. We say that $n_2$ is a descendant of $n_1$, whereas $n_1$ is an ancestor of $n_2$. Note that every node is both a descendant and an ancestor of itself. If $n_1 \neq n_2$, then $n_2$ is a proper descendant of $n_1$, which in turn is a proper ancestor of $n_2$.

A subset $V$ of the nodes of a tree $T$ induces the subgraph $G$ of $T$ that consists of all the nodes of $V$ and the edges connecting those nodes. Note that $G$ is a subtree of $T$ if it is connected. Given a node $n$ of a tree $T$, we denote by $T^n_\Lambda$ the subtree of $T$ that is rooted at $n$ and is induced by all the descendants of $n$. A branch of a tree $T$ consists of the root $r$ of $T$, a child $n$ of $r$ (including the edge $(r, n)$) and the subtree $T^r_\Lambda$. The branch that contains the child $n$ of the root $r$ is denoted by $T^r_\Lambda$. More generally, the branch of $T^m_\Lambda$ that contains the child $n$ of $m$ is denoted by $T^m_n_\Lambda$. Note that a branch has at least two nodes, and the number of branches of $T$ is the same as the number of children of the root.

An XML document (or just document for short) is a tree with a label attached to every node. The label represents either a tag, a value or both. Usually, we use $d$ to denote documents, and $u, v$ and $w$ to denote nodes of documents. The label of a node $v$ is denoted by $\text{lbl}(v)$. As an example, the top right side of Figure 4.1 depicts a document $d$. Each node is represented as $[i]l$, where $i$ is a unique identifier and $l$ is a label. For instance, the label of Node 19 is “manager” while that of Node 14 is “Emma.” Note that a label that corresponds to a value (rather than tag) is written in italic.

Two documents $d_1$ and $d_2$ are isomorphic, denoted by $d_1 \sim d_2$, if there is a one-to-one correspondence between the nodes of the two documents that preserves labels and edges. That is, there exists a bijection $\Phi : \mathcal{V}(d_1) \rightarrow \mathcal{V}(d_2)$, such that (1) for all $v \in \mathcal{V}(d_1)$, $\text{lbl}(v) = \text{lbl}(\Phi(v))$, and (2) for all edges $(v_1, v_2) \in \mathcal{E}(d_1)$, $(\Phi(v_1), \Phi(v_2)) \in \mathcal{E}(d_2)$.

### 4.2 The p-Document Model

In this section, we present an abstract model of probabilistic XML. In Section 4.3, we derive several concrete models from the abstract one, including (or generalizing) various approaches that have been studied in the literature.

---

1It is notationally easier to use a single label rather than a tag and a value.
The notion of probabilistic XML refers to a probability distribution over a space of ordinary documents. A specific model is a mechanism for defining that distribution in terms of a probabilistic process that generates a random document, which is an ordinary XML document.

The abstract model of this section is a common framework for generating random documents. The concrete models of Section 4.3 add the details that are needed to fully define specific probabilistic processes.

A probabilistic document (abbr. p-document) is a tree $\tilde{P}$ that has two types of nodes. Ordinary nodes have labels (i.e., they are regular XML nodes), and they may appear in random documents. Distributional nodes are only used for defining the probabilistic process that generates random documents (but they do not occur in those documents). We denote
by $\mathcal{V}^{\text{ord}}(\hat{P})$ and $\mathcal{V}^{\text{dst}}(\hat{P})$ the disjoint sets of ordinary and distributional nodes of $\hat{P}$, respectively. A distributional node is neither the root nor a leaf of $\hat{P}$.

**Example 4.2.1** Figure 4.1 shows a p-document $\hat{P}$. Distributional nodes are depicted as rounded rectangles that surround words (e.g., ind or mux) denoting the types of those nodes. Each node has a unique identifier of the form $[i]$. The types (and the probabilities attached to edges) are discussed in Section 4.3.

A distributional node $v$ specifies a distribution over the subsets of its children. The probabilistic process that generates a random document chooses a particular subset of the children of $v$ according to the distribution specified for $v$.

A random document of a p-document $\hat{P}$ is generated in two steps. In the first step, one subset of children is randomly chosen for each distributional node. Note that choices made for different nodes could be dependent. All the unchosen children and their descendants (even descendants that have been chosen) are deleted. The result is an r-subtree $s$ of $\hat{P}$.

The second step removes all the distributional nodes. If an ordinary node $u$ no longer has a parent, then the new parent of $u$ is the lowest node among the ordinary proper ancestors of $u$. The resulting document is ordinary and denoted by $\text{doc}(s)$.

In terms of formal probability theory, the probability space that is associated with a p-document consists of all the different combinations of children sets that can be chosen by the distributional nodes. The probability of each such combination depends on the types of the distributional nodes (and the dependencies that exist among them). The above two steps naturally define two random variables that are described next.

We use $\Sigma(\hat{P})$ to denote the set of all r-subtrees $s$ of $\hat{P}$, such that every ordinary node $u$ of $s$ has the same set of children in both $s$ and $\hat{P}$. The random variable $\mathcal{P}^\Sigma$ denotes the r-subtree $s \in \Sigma(\hat{P})$ that is chosen in the first step above. We are usually interested in the random variable $\mathcal{P}$ (i.e., without the tilde sign) which is $\text{doc}(\mathcal{P}^\Sigma)$, i.e., it denotes the result of the second step—the actual random document. $\Pr(\mathcal{P}^\Sigma = s)$, where $s \in \Sigma(\hat{P})$, is the probability that each distributional node of $s$ chooses the exact set of children that it
has in $s$. Thus, the probability of a random document $d$ is given by

$$\Pr (P = d) = \sum_{\substack{s \in \Sigma(\tilde{P}) \\text{doc}(s) = d}} \Pr (P^\Sigma = s).$$

Note that if $d$ cannot be obtained from $\tilde{P}$ by the above two-step process (e.g., $d$ has a node that does not appear in $\tilde{P}$), then $\Pr (P = d) = 0$.

**Example 4.2.2** Consider again Figure 4.1. The p-document $\tilde{P}$ is discussed in Example 4.2.1. The central part of the figure depicts an r-subtree $s \in \Sigma(\tilde{P})$, and the right part is the document $d = \text{doc}(s)$. It can be easily shown that $s$ is the only r-subtree of $\Sigma(\tilde{P})$ that generates $d$ and, consequently, $\Pr (P = d) = \Pr (P^\Sigma = s)$. The computation of the probability on the right-hand side will be explained in Section 4.3.

Note that the operation $\text{doc}(\cdot)$ is not necessarily one-to-one; that is, two different r-subtrees $s_1$ and $s_2$ may yield the same document. This follows from the fact that a distributional node can have a distributional child.

We use $\text{pwd}(\tilde{P})$ to denote the set of all documents $d$ with a non-zero probability, that is, $\Pr (P = d) > 0$. These documents are the possible worlds. From the above description of the process of generating a possible world, it clearly follows that $\sum_{d \in \text{pwd}(\tilde{P})} \Pr (P = d) = 1$.

### 4.2.1 Equivalence of p-Documents

Two p-documents are *equivalent* if they yield the same probability space. Next, we define two types of equivalence—one for the object-based semantics and the other for the value-based semantics.

The p-documents $\tilde{P}_1$ and $\tilde{P}_2$ are *o-equivalent*, denoted by $\tilde{P}_1 \equiv_o \tilde{P}_2$, if $\Pr (P_1 = d) = \Pr (P_2 = d)$ holds for all documents $d$. Similarly, $\tilde{P}_1$ and $\tilde{P}_2$ are *v-equivalent*, denoted by $\tilde{P}_1 \equiv_v \tilde{P}_2$, if $\Pr (P_1 \sim d) = \Pr (P_2 \sim d)$ holds for all documents $d$. In other words, we view a node as an object (with an identifier) and then v-equivalence is similar to o-equivalence, except that we do not distinguish between two documents if one is obtained from the other.
by replacing an object with a different object that has the same label. Clearly, object equivalence implies value equivalence, but not vice versa.

Suppose that $\tilde{P}_1 \equiv_o \tilde{P}_2$. If an ordinary node exists in just one of the two, then it can occur only in documents that have zero probability. Hence, for reasoning about o-equivalence, we can assume that $\tilde{P}_1$ and $\tilde{P}_2$ have the same set of ordinary nodes.

### 4.2.2 Families of p-Documents

The next section describes several families of p-documents. The following definitions are used for comparing the expressive power of those families.

Consider two (infinite) sets $\mathcal{F}_1$ and $\mathcal{F}_2$ of p-documents. We say that $\mathcal{F}_1$ is o-translatable to $\mathcal{F}_2$, denoted by $\mathcal{F}_1 \sqsubseteq_o \mathcal{F}_2$, if each document of $\mathcal{F}_1$ is o-equivalent to some document of $\mathcal{F}_2$. That is, for each document $\tilde{P}_1 \in \mathcal{F}_1$, there exists a document $\tilde{P}_2 \in \mathcal{F}_2$, such that $\tilde{P}_1 \equiv_o \tilde{P}_2$. An o-translator from $\mathcal{F}_1$ to $\mathcal{F}_2$ is an algorithm that receives as input a $\tilde{P}_1 \in \mathcal{F}_1$ and generates an o-equivalent $\tilde{P}_2 \in \mathcal{F}_2$. If there is an efficient o-translator from $\mathcal{F}_1$ to $\mathcal{F}_2$, then $\mathcal{F}_1$ is efficiently o-translatable to $\mathcal{F}_2$, denoted by $\mathcal{F}_1 \sqsubseteq_{o}^{\text{poly}} \mathcal{F}_2$. If $\mathcal{F}_1 \sqsubseteq_o \mathcal{F}_2$ and $\mathcal{F}_2 \sqsubseteq_o \mathcal{F}_1$, then we write $\mathcal{F}_1 \equiv_o \mathcal{F}_2$. Similarly, $\mathcal{F}_1 \equiv_{o}^{\text{poly}} \mathcal{F}_2$ means that there are efficient o-translators in both directions.

We use analogous definitions and notations for the notion of v-translation. For example, $\mathcal{F}_1 \sqsubseteq^{\text{poly}}_v \mathcal{F}_2$ means that there is an efficient v-translator that receives as input a $\tilde{P}_1 \in \mathcal{F}_1$ and generates a $\tilde{P}_2 \in \mathcal{F}_2$, such that $\tilde{P}_1 \equiv_v \tilde{P}_2$.

### 4.3 Specific Models of p-Documents

In this section, we consider several models of p-documents, including some of those that have been studied in the literature. These models are characterized by two properties: (1) The types of distributional nodes that are used in the p-documents, and (2) Whether one can construct hierarchies consisting of only distributional nodes.

We now define the second property. A p-document $\tilde{P}$ is distributional-hierarchy free.
(abbr. DHF) if every distributional node of $\hat{P}$ has only ordinary children. If $F$ is a set of p-documents, then $F|_U$ denotes the restriction of $F$ to its DHF p-documents. Next, we discuss the first property.

4.3.1 Types of Distributional Nodes

To obtain a concrete p-document, we should specify for each distributional node $v$, the probability distribution of choosing a subset of the children of $v$. We define four types of distributional nodes, each with a different way of describing that probability distribution.

**Type ind.** A node $v$ of type ind specifies for each child $w$, the probability $p^v(w)$ of choosing $w$; this probability is independent of any other choice of children (of either $v$ or other distributional nodes). Hence, the probability of choosing a subset $C$ of children of $v$ is $\prod_{w \in C} p^v(w) \prod_{w \in \bar{C}} (1 - p^v(w))$, where $\bar{C}$ is the set of children of $v$ that are not in $C$.

**Type mux.** A node $v$ of type mux specifies probabilities $p^v(w_1), \ldots, p^v(w_k)$ for its children $w_1, \ldots, w_k$, respectively. Node $v$ chooses at most one child $w_i$ with the probability $p^v(w_i)$, independently of the other distributional nodes. We require that $\sum_{i=1}^k p^v(w_i) \leq 1$. The probability that $v$ chooses none of its children is $1 - \sum_{i=1}^k p^v(w_i)$.

**Type exp.** A node $v$ of type exp specifies probabilities $p^v(W_1), \ldots, p^v(W_l)$, where $W_1, \ldots, W_l$ are some (but not necessarily all) distinct subsets of the children of $v$. Node $v$ chooses exactly one subset $W_i$ with the probability $p^v(W_i)$, independently of the other distributional nodes. Note that $W_i$ can be empty. We require that $\sum_{i=1}^l p^v(W_i) = 1$.

**Type cie.** Nodes of this type are associated with independent random Boolean variables $e_1, \ldots, e_m$, called events. For each event $e_i$, the p-document specifies the probability $p(e_i)$ that $e_i$ is true. A node $v$ of type cie specifies for every child $w$, a conjunction $\alpha^v(w) = a_1 \land \cdots \land a_w$, where each $a_j$ is either $e_i$ or $\neg e_i$ for some $1 \leq i \leq m$ (note that different cie nodes can have common events). Before choosing children, values for $e_1, \ldots, e_m$ are randomly determined. A child $w$ is chosen if its corresponding conjunction $\alpha^v(w)$ is true.

**Example 4.3.1** The p-document $\hat{P}$ of Figure $4.1$ has ind and mux nodes. The probability specified for each child is shown next to the edge that leads to that child. We now describe
how to compute the probability $\Pr(\tilde{\Sigma} = s)$ of the document $s \in \Sigma(\tilde{\mathcal{P}})$ that is shown in the middle part of Figure 4.1. Each mux node of $s$ chooses exactly one child with the probability specified for that child. The probabilities of the choices made by the ind nodes are as follows. Node 9 chooses its only child with probability 0.8. Node 10 chooses both children with probability $0.7 \cdot 0.65 = 0.455$. And Node 23 chooses the empty set of children with probability $1 - 0.8 = 0.2$. $\Pr(\tilde{\Sigma} = s)$ is the product of the probabilities of the choices made by all the distributional nodes.

We denote by $\PrXML^{\{\text{type}_1, \text{type}_2, \ldots\}}$ the family of all p-documents that use distributional nodes of only the types listed in the superscript. For example, the p-documents of $\PrXML^{\{\text{ind}, \text{mux}\}}$ use only ind and mux nodes.

Throughout this paper, our complexity analysis assumes that each probability specified in a p-document (e.g., $p(v_i)$ or $p(e_i)$) is a rational number (in $[0, 1]$) that is given as two integers: the numerator and the denominator.

4.3.2 Previously Studied Models

The family $\PrXML^{\{\text{ind, mux}\}}$ comprises the ProTDB documents of [86]. The probabilistic XML model of [103] is a subset of $\PrXML^{\{\text{ind, mux}\}}$, where mux nodes (called “probability nodes”) have only ind nodes (called “possibility nodes”) as children, and ind nodes have only ordinary children (called “XML nodes”) that are all selected with probability one. Alternatively, this model can be viewed as a subset of $\PrXML^{\{\text{exp}\}}_{\mid h'}$.

The model of probabilistic XML that was investigated in [18, 98] is $\PrXML^{\{\text{cie}\}}_{\mid h'}$. In Appendix D we show that adding hierarchies of distributional nodes to $\PrXML^{\{\text{cie}\}}_{\mid h'}$ is not needed, that is, $\PrXML^{\{\text{cie}\}}_{\mid h'} \equiv_{\text{poly}} \PrXML^{\{\text{cie}\}}$. The “simple probabilistic trees” of [18] are actually the family $\PrXML^{\{\text{ind}\}}_{\mid h'}$ (and hierarchies make a difference in this case).

The work of [65] introduced a model of probabilistic XML graphs, where each node explicitly specifies the probability distribution over its possible sets of children. Restricting

---

3In the probabilistic documents of [103] the root is distributional; so we assume that a dummy ordinary node is added for compliance with the p-document model.
the XML graphs of [65] to trees yields a sub-family of \( \text{PrXML}_{\text{exp}} \) (again, lack of hierarchies is significant). The same is true for [66] if we restrict their intervals to points.

4.3.3 Translations among the Families

We now consider the ability to efficiently translate a p-document of one family into another family. Due to a lack of space, the proofs and translation techniques are given in Appendix D while here, we only give the bottom-line results.

The four types of distributional nodes and the restriction of a family to its DHF instances give rise to a huge number of different families of p-documents. We considered the important families which include the existing models and natural extensions, restrictions and combinations thereof. Figure 4.2 shows the efficient o-translations that exist between these families by means of a directed graph. Each edge corresponds to a specific efficient translation algorithm and, consequently, for all families \( \mathcal{F}_1 \) and \( \mathcal{F}_2 \), if there is a path from \( \mathcal{F}_1 \) to \( \mathcal{F}_2 \), then there is an efficient o-translation from \( \mathcal{F}_1 \) and \( \mathcal{F}_2 \) (since translations can be composed). Figure 4.2 describes a complete picture, in the following sense. If there is no path from the family \( \mathcal{F}_1 \) to the family \( \mathcal{F}_2 \), then there is no efficient o-translator from \( \mathcal{F}_1 \) to \( \mathcal{F}_2 \).

As for v-translations, we draw a picture that is almost complete. Clearly, there is an efficient v-translator from the family \( \mathcal{F}_1 \) to the family \( \mathcal{F}_2 \) whenever there is an efficient
o-translator from $\mathcal{F}_1$ to $\mathcal{F}_2$ (since an o-translator is a special case of a v-translator). In particular, there is an efficient v-translator from $\mathcal{F}_1$ to $\mathcal{F}_2$ whenever Figure 4.2 contains a path from $\mathcal{F}_1$ to $\mathcal{F}_2$. We show (in Appendix D) various cases where no efficient v-translator exists. For example, hierarchy is essential in PrXML$^{\{\exp\}}$, namely, PrXML$^{\{\exp\}}$ is not efficiently v-translatable to PrXML$^{\{\exp\}}_{lf}$ (although an inefficient v-translation exists). This shows that PrXML$_{lf}^{\{\exp\}}$ is strictly more expressive than the probabilistic tree model of [65, 66]. As another example, PrXML$_{cie}^{\{\exp\}}$ is v-translatable to PrXML$^{\{\exp\}}$, but there is no efficient v-translator (to prove that, we adapt a proof of [99]). The cases for which the existence of an efficient v-translator is an open problem (and solving each of the open problems will complete the picture) are the ones where the source family is one of PrXML$^{\{\exp\}}$, PrXML$^{\{\exp\}}_{lf}$ and PrXML$^{\{\exp, cie\}}$, and the target family is among PrXML$_{ind, mux}^{\{\exp, cie\}}$ and PrXML$^{\{\exp, cie\}}$.

For the sequel, the following conclusion from Figure 4.2 is important. The family PrXML$^{\{\exp, cie\}}$ is the most general, and the families PrXML$^{\{\exp\}}$ and PrXML$^{\{\exp, cie\}}_{lf}$ are just below PrXML$^{\{\exp, cie\}}$.

### 4.4 Queries

Our queries are essentially twig patterns with projection. These queries extend common fragments of XML queries [16, 25, 46, 84]. We first define twig patterns.

#### 4.4.1 Twig Patterns

A *twig pattern* (or just *pattern* for short) is a tree with child edges and descendant edges. Each node $n$ of the twig is associated with a unary predicate $\text{cond}_n(\cdot)$ over labels. As explained later, $n$ can match a node $v$ of a document only if $\text{cond}_n(lbl(v))$ is satisfied. We assume that $\text{cond}_n(lbl(v))$ can be efficiently decided. Usually, we denote a pattern by $T$, and use $n$ and $m$ to denote the nodes of $T$ (in order to make a clear distinction from nodes of documents).

A *match* of a pattern in a document is a mapping from the nodes of the pattern to those of the document, such that all the constraints of the pattern are satisfied. Formally,
a match of the pattern $T$ in the document $d$ is a mapping $\varphi : \mathcal{V}(T) \rightarrow \mathcal{V}(d)$ that satisfies the following.

1. Roots are matched, i.e., $\varphi(\text{root}(T)) = \text{root}(d)$.
2. For all nodes $n \in \mathcal{V}(T)$, the predicate $\text{cond}_n(lbl(\varphi(n)))$ is satisfied.
3. For all child edges $(n_p, n_c) \in \mathcal{E}(T)$, the node $\varphi(n_p)$ of $d$ is the parent of $\varphi(n_c)$.
4. For all descendant edges $(n_a, n_d) \in \mathcal{E}(T)$, the node $\varphi(n_a)$ of $d$ is a proper ancestor of $\varphi(n_d)$.

By $\mathcal{M}(T, d)$ we denote the set of all matches of a pattern $T$ in a document $d$. The notation $d \models T$ means that $\mathcal{M}(T, d) \neq \emptyset$, that is, there is a match of $T$ in $d$.

**Example 4.4.1** The bottom left side of Figure 4.1 depicts a pattern $T$. The nodes of $T$ have three types of predicates. The predicates $\text{cond}_{n_1}(x)$ and $\text{cond}_{n_3}(x)$ are $x = l$, where $l$ is “company” and “manager,” respectively. The predicate of $n_6$ is $x = "M" \lor x = "L"$. Finally, the predicate of $n_2$, $n_4$ and $n_5$, denoted by $*$, is simply true (that is, every label satisfies it). Consider the document $d$ in that figure. We denote by $v_i$ the node of $d$ with the identifier $i$. There are three matches of $T$ in $d$: (1) $\varphi_1 : n_1, \ldots, n_6 \mapsto v_1, v_2, v_{12}, v_{14}, v_3, v_7$, respectively (i.e., $\varphi_1(n_1) = v_1, \varphi_1(n_2) = v_2$ and so on); (2) $\varphi_2 : n_1, \ldots, n_6 \mapsto v_1, v_{11}, v_{12}, v_{14}, v_{16}, v_{17}$, respectively; and (3) $\varphi_3 : n_1, \ldots, n_6 \mapsto v_1, v_2, v_{19}, v_{22}, v_3, v_7$, respectively. In particular, note that $d \models T$ holds.

### 4.4.2 Queries

We consider two semantics of queries. In the object-based semantics, an answer consists of nodes of documents, while in the value-based semantics, it consists of the labels of those nodes. One case where there is a difference between the two is the following. If a document $d$ has two isomorphic subtrees $d_1$ and $d_2$, then a match of a pattern $T$ in $d_1$ and the isomorphic match in $d_2$ produce two distinct answers under the object-based semantics, but only one answer under the value-based semantics. Next, we define the two semantics.
Consider a pattern $T$ and a document $d$. Under the object-based semantics, the result of applying $T$ to $d$, denoted by $T^o(d)$, is the set comprising all the matches of $T$ in $d$. Given a match $\psi \in T^o(d)$, the function $\text{lbl} \circ \psi$ maps every node $n$ of $T$ to the label of $\psi(n)$. Under the value-based semantics, the result of applying $T$ to $d$, denoted by $T^v(d)$, is the set of all the mappings $\text{lbl} \circ \psi$, where $\psi \in T^o(d)$. Note that $T^o(d) \neq \emptyset$ if and only if $T^v(d) \neq \emptyset$.

Furthermore, $|T^v(d)| \leq |T^o(d)|$, and sometimes it is a strict inequality.

A projection sequence of $T$ is a list $X = (n_1, \ldots, n_k)$ of distinct nodes of $T$. When applying $X$ to a mapping $\varphi$ of either $T^o(d)$ or $T^v(d)$, the result is the tuple $(\varphi(n_1), \ldots, \varphi(n_k))$, denoted by $\varphi[X]$. Thus, $\varphi[X]$ is a tuple of either nodes or labels, depending on whether $\varphi \in T^o(d)$ or $\varphi \in T^v(d)$, respectively.

Naturally, we have two types of queries. Both types are defined by a pattern $T$ and a projection sequence $X$ of $T$. An o-query has the form $\pi_X T^o$. The result of applying $\pi_X T^o$ to a document $d$, denoted by $\pi_X T^o(d)$, is obtained by applying $X$ to each tuple of $T^o(d)$. That is,

$$\pi_X T^o(d) \overset{\text{def}}{=} \{ \varphi[X] \mid \varphi \in T^o(d) \}.$$

Similarly, a v-query has the form $\pi_X T^v$. The result of applying $\pi_X T^v$ to a document $d$, denoted by $\pi_X T^v(d)$, is

$$\pi_X T^v(d) \overset{\text{def}}{=} \{ \varphi[X] \mid \varphi \in T^v(d) \}.$$

Note that if $d_1 \sim d_2$, then $\pi_X T^v(d_1) = \pi_X T^v(d_2)$; of course, this is not the case for $\pi_X T^o$.

**Example 4.4.2** Consider again Example 4.3.1 Let $X$ be the sequence $(n_4, n_6)$ (the two leaves of $T$). The query $\pi_X T^v$ (or $\pi_X T^o$) finds managers of departments and sizes, such that either one of the following two conditions holds. First, the department is either medium or large. Second, the department belongs to a division that has a medium or large amount of sales. Each answer consists of a manager and a size (medium or large).

Now consider the o-query $\pi_X T^o$. The result $\pi_X T^o(d)$ of applying $\pi_X T^o$ to $d$ is the set of tuples $\{(v_{14}, v_7), (v_{14}, v_{17}), (v_{22}, v_7)\}$. For the v-query $\pi_X T^v$, the set of tuples $\pi_X T^v(d)$ is $\{(\text{Emma}, L), (\text{John}, L)\}$. Note that the first two tuples of $\pi_X T^o(d)$ correspond to the same tuple of $\pi_X T^v(d)$, namely, $(\text{Emma}, L)$. \qed
A special case is an empty projection sequence \( X = () \). In this case, both \( \pi_X T^o(d) \) and \( \pi_X T^v(d) \) are either \{()\} or \( \emptyset \), depending on whether \( d \models T \) or not, respectively. Naturally, such a query is called Boolean.

### 4.5 Probabilistic Semantics of Queries

We now define the semantics of querying p-documents. Consider a p-document \( \tilde{P} \) and a query \( Q \). Given a document \( d \in pwd(\tilde{P}) \), the evaluation of \( Q \) over \( d \) results in a set of answers \( Q(d) \) (as defined in Section 4.4.2). However, \( \tilde{P} \) represents a probabilistic space of documents rather than a specific one. In particular, a specific answer can be obtained in one document but not in another. We follow a well known notion of querying probabilistic data [18, 36, 37, 86, 98], which is described as follows. When evaluating \( Q \) over \( \tilde{P} \), each answer \( a \) has a probability that represents its degree of certainty. This is the probability of obtaining \( a \) when querying a random document. Thus, an answer with a high probability is likely to be obtained in a random document.

More formally, consider a p-document \( \tilde{P} \) and a query \( Q = \pi_X T^o \) (or \( Q = \pi_X T^v \)). We use \( Q(\tilde{P}) \) to denote the set of all tuples \( t \), such that \( t \in Q(d) \) for some document \( d \in pwd(\tilde{P}) \). The goal of query evaluation is to determine for every \( t \in Q(\tilde{P}) \), the probability \( \Pr (t \in Q(\tilde{P})) \). Note that the definition of \( pwd(\tilde{P}) \) implies that \( \Pr (t \in Q(\tilde{P})) > 0 \) for all \( t \in Q(\tilde{P}) \). If \( Q \) is a Boolean query (i.e., \( X = \emptyset \)), then evaluating \( Q \) amounts to computing \( \Pr (\tilde{P} \models T) \).

### 4.6 Reducing General Queries to Boolean Queries

In this section, we show how evaluation of general queries can be reduced to that of Boolean queries. Under data complexity, one can efficiently find a superset of \( Q(\tilde{P}) \) by employing an XQuery processor that applies \( Q \) to the ordinary document that is obtained from \( \tilde{P} \) as follows. We remove all the distributional nodes of \( \tilde{P} \), and for every node \( v \) that remains without a parent, we make \( v \) a child of its lowest ordinary ancestor. Consequently, in order to find \( Q(\tilde{P}) \), it is sufficient to have an efficient algorithm that determines whether
the probability of a specific match is nonzero. There is such an algorithm even for p-documents of PrXML\(^{\{\text{exp,\text{cie}}\}}\) (as explained later in Section 4.8). But we want to do more than that, namely, to compute the probability of each tuple of \(Q(\mathcal{P})\). So next, we show how \(\Pr (t \in Q(\mathcal{P}))\) is computed efficiently for each \(t \in Q(\mathcal{P})\). We denote the projection sequence of \(Q\) by \(X = (n_1, \ldots, n_k)\).

We start with the object-based semantics, that is, \(Q = \pi_X T^o\). Given a tuple \(t = (v_1, \ldots, v_k)\), we transform \(\mathcal{P}\) into a new p-document \(\mathcal{P}_t\) as follows. For each \(1 \leq i \leq k\), we replace the label of \(v_i\) with a new unique label \(l_i\) that does not appear anywhere else in \(\mathcal{P}_t\). We also construct a pattern \(T_t\) that is identical to \(T\), except for the following. For each \(1 \leq i \leq k\), we replace \(\text{cond}_{n_i}(\cdot)\) with \(\text{lbl}(\cdot) = l_i\). It is easy to show that there is a one-to-one correspondence between the random documents of \(\mathcal{P}_t\) and those of \(\mathcal{P}\), such that probabilities (of documents) are preserved. (The only difference between two corresponding documents is in the labels of the nodes of \(t\).) Moreover, there is a match \(\varphi\) of \(T\) in a random document \(d\) of \(\mathcal{P}\) that satisfies \(\varphi[X] = t\) (i.e., \(t \in Q(d)\)) if and only if there is a match of \(T_t\) in the random document that corresponds to \(d\). Thus, \(\Pr (t \in \pi_X T^o(\mathcal{P})) = \Pr (\mathcal{P}_t \models T_t)\).

We now consider the value-based semantics and show how \(\Pr (t \in \pi_X T^v(\mathcal{P}))\) is computed efficiently for each \(t \in \pi_X T^v(\mathcal{P})\). In this case, we obtain the pattern \(T_t\) from \(T\) by replacing the predicate of each \(n_i\) with the condition that the label is equal to \(l_i\). (Unlike the previous case, we do not change \(\mathcal{P}_t\).) Now, it is easy to see that a random document \(d\) of \(\mathcal{P}\) satisfies \(t \in Q(\mathcal{P})\) if and only if there is a match of \(T_t\) in \(\mathcal{P}_t\). So, again, \(\Pr (t \in \pi_X T^v(\mathcal{P})) = \Pr (\mathcal{P}_t \models T_t)\). Thus, we obtain the following result.

**Proposition 4.6.1** Under data complexity, the following are equivalent for a family \(\mathcal{F}\) of p-documents.

- \(v\)-queries can be evaluated efficiently.
- \(o\)-queries can be evaluated efficiently.\(^4\)
- Boolean queries can be evaluated efficiently.

\(^4\)In this part, it is assumed that \(\mathcal{F}\) is closed under replacement of labels of ordinary nodes.
4.7 Evaluating Boolean Queries

In this section, we describe the algorithm EvalDP for evaluating Boolean queries over PrXML\(^\{\text{exp}\}\). We first show that this is the most general family, among those of Section 4.3, that has an efficient evaluation algorithm. In particular, the following theorem shows that every Boolean query is \#P-complete in PrXML\(^{\text{cie}}\) (i.e., the probabilistic trees of 18, 98, 99) unless it is extremely trivial. The proof is in Appendix E.

**Theorem 4.7.1** If one of the conditions below holds for a fixed pattern \(T\), then computing \(\Pr(\mathcal{P} \models T)\) over PrXML\(^{\text{cie}}\) is in polynomial time. Otherwise, it is \#P-complete.

- \(T\) has only one node.
- \(\text{cond}_n(\cdot) \equiv \text{false}\) for some \(n \in \mathcal{V}(T)\).

Moreover, this dichotomy holds even under the assumption that \(\tilde{\mathcal{P}}\) does not include negated events.

### 4.7.1 The Algorithm EvalDP

Consider a p-document \(\tilde{\mathcal{P}} \in \text{PrXML}^{\{\text{exp}\}}\) with a root \(r\) and a pattern \(T\) with a root \(n\). The goal is to compute the probability \(\Pr(\mathcal{P} \models T)\) that there is a match of \(T\) in a random document of \(\tilde{\mathcal{P}}\). First, we test whether \(r\) satisfies the predicate of \(n\). If the test fails, then the probability is 0. If it succeeds, then we would like to continue recursively by evaluating the subtrees rooted at the children of \(n\) against the subtrees rooted at the children of \(r\). We may try to do that by computing, for each child \(m\) of \(n\), the probability of the following event: there is some child \(u\) of \(r\), such that there is a match of \(T_m^x\) in a random document of \(\tilde{\mathcal{P}}_u\). But these are not independent events, and it is not clear how their probabilities can be used to compute \(\Pr(\mathcal{P} \models T)\). Furthermore, this approach is inapplicable if the root of \(T\) is connected to some of its children by descendant edges. Consequently, the algorithm EvalDP is more involved and we next give its details.

\(^5\text{Recall that }\#P\text{ is the class of functions that count the number of accepting paths of an NP machine, which is conventionally deemed intractable [101, 102].}\)
Notation and Terminology

We generalize the notation $d \models T$ to Boolean expressions over patterns. For example, $d \models T_1 \land \neg T_2$ means that there is a match of $T_1$ in $d$ and there is no match of $T_2$ in $d$. If $\mathcal{F}$ is a set of patterns, then $\land \mathcal{F}$ denotes the conjunction of the patterns of $\mathcal{F}$, i.e., $\land \mathcal{F} = \land_{T \in \mathcal{F}} T$. Similarly, $\lor \mathcal{F}$ is the disjunction of the patterns of $\mathcal{F}$.

Recall that a branch of a tree $Z$ is denoted as $Z^r_u \Delta$, where $u$ is a child of the root $r$. A sub-branch (abbr. SB) of a pattern $T$ is a branch of a subtree $T^v_\Delta$ (i.e., it comprises $v$ and all the descendants of exactly one child of $v$). Note that a branch is also a sub-branch, and a leaf is not a sub-branch. A relaxed sub-branch (abbr. RSB) of $T$ is obtained from an SB by replacing the root with a new node $*$ (we use $*$ to denote both the predicate true and a node that has that predicate, i.e., every label satisfies the predicate of $*$).

Consider a pattern $\hat{T}$ and a new node $n$. The patterns $n/\hat{T}$ and $n//\hat{T}$ are obtained from $\hat{T}$ by making $n$ the new root and adding a child edge and a descendant edge, respectively, from $n$ to root($\hat{T}$).

Let $\hat{\mathcal{P}}$ be a p-document. For each node $v$ of $\hat{\mathcal{P}}$, we define the p-document $\hat{\mathcal{P}}_v$ as follows. If $v$ is an ordinary node, then $\hat{\mathcal{P}}_v$ is just $\hat{\mathcal{P}}^v_\Delta$ (i.e., the subtree of $\hat{\mathcal{P}}$ rooted at $v$). If $v$ is distributional, then $\hat{\mathcal{P}}^v_\Delta$ is not a p-document (since the root is distributional) and, hence, in this case we introduce a new ordinary node $\circ$ that has the special property of satisfying every predicate. $\hat{\mathcal{P}}_v$ is obtained by adding $\circ$ to $\hat{\mathcal{P}}^v_\Delta$ as a root (and $v$ is the only child of $\circ$).

The Algorithm

The input of the algorithm EvalDP is a p-document $\hat{\mathcal{P}}$ and a pattern $T$, and it computes $\Pr(\mathcal{P} \models T)$. We denote by $\hat{\mathcal{R}}$ the set of all the SBs, RSBs and leaves of $T$. In order to compute $\Pr(\mathcal{P} \models T)$, the algorithm EvalDP solves the more general problem of computing $\Pr(\mathcal{P}_v \models \land \hat{\mathcal{R}})$ (i.e., the probability that there is a match of each pattern of $\hat{\mathcal{R}}$ in $\mathcal{P}_v$) for all nodes $v$ of $\hat{\mathcal{P}}$ and subsets $\hat{\mathcal{R}}$ of $\hat{\mathcal{R}}$. This solves our problem since $\Pr(\mathcal{P} \models T)$ is equal to $\Pr(\mathcal{P}_r \models \land \hat{\mathcal{R}}_T)$, when $r$ is the root of $\hat{\mathcal{P}}$ and $\mathcal{R}_T$ is the set of branches of $T$.

The computation of $\Pr(\mathcal{P}_v \models \land \hat{\mathcal{R}})$ is done by dynamic-programming through a bottom-up traversal of $\hat{\mathcal{P}}$. Thus, when $v \in \mathcal{V}(\hat{\mathcal{P}})$ is reached, then for every child $u$ of $v$ and for all
\( R' \subseteq R \), the probability \( \Pr(P_u \models \land R') \) has already been computed.

**Three Simple Cases**

We first describe how to compute \( \Pr(P_v \models \land \hat{R}) \) when we need not know the probabilities for the children of \( v \).

**Case 1:** Node \( v \) is a leaf (and, hence, ordinary). In this case, \( v \) itself is the only random document. Therefore, \( \Pr(P_v \models \land \hat{R}) = 1 \) if each of the patterns of \( \hat{R} \) has only one node with a predicate satisfied by \( \text{lbl}(v) \); otherwise, \( \Pr(P_v \models \land \hat{R}) = 0 \).

**Case 2:** Node \( v \) is ordinary and it does not satisfy the root predicate of some \( R \in \hat{R} \). In this case, \( \Pr(P_v \models \land \hat{R}) = 0 \).

**Case 3:** Each \( R \in \hat{R} \) consists of a single node. If \( v \) is either a distributional node or an ordinary node with a label that satisfies all the predicates of the nodes of \( \hat{R} \), then \( \Pr(P_v \models \land \hat{R}) = 1 \). Otherwise, it is the previous case.

In the remainder of the algorithm, we assume that none of the above three cases holds. In other words, \( v \) is not leaf of \( \hat{P} \), and if it is ordinary, then its label satisfies the root predicate of each pattern in \( \hat{R} \). In addition, \( \hat{R} \) includes at least one pattern that is not a leaf. Our first action is to eliminate from \( \hat{R} \) all the leaf patterns (we can do that since Case 2 does not hold). So, we actually assume that each pattern of \( \hat{R} \) has two or more nodes. The patterns of \( \hat{R} \) are denoted by \( R_1, \ldots, R_q \). The remainder of the algorithm for computing \( \Pr(P_v \models \land \hat{R}) \) considers the following cases.

**Ordinary Node and One Distributional Child**

Suppose that \( v \) is an ordinary node with a single child \( u_1 \), which is distributional. Since \( \text{lbl}(v) \) satisfies the root predicate of each of the patterns of \( \hat{R} \) (otherwise, Case 2 holds), the definition of the p-document \( \hat{P}_{u_1} \) implies the following.

\[
\Pr\left( P \models \land \hat{R} \right) = \Pr\left( P_{u_1} \models \land \hat{R} \right)
\]
Since $u_1$ is a child of $v$, the right side of the above equation has already been computed in a previous iteration.

**Ordinary Node and One Ordinary Child**

Now, suppose that both $v$ and its only child $u_1$ are ordinary nodes. Let $T_1, \ldots, T_q$ be obtained from $R_1, \ldots, R_q$, respectively, by deleting the root, and let $\mathcal{T} = \{T_1, \ldots, T_q\}$. Then,

$$
\Pr(\mathcal{P}_v \models \land \mathcal{R}) = \Pr(\mathcal{P}_{u_1} \models \land \mathcal{T}).
$$

However, this equation does not necessarily hold if some of the patterns of $\mathcal{R}$ have outgoing descendant edges. To handle descendant edges, we reduce the computation to that of a conjunction of negated branches as follows.

Since the negation of $\land \mathcal{R}$ is $\neg R_1 \lor \cdots \lor \neg R_q$,

$$
\Pr(\mathcal{P}_v \models \land \mathcal{R}) = 1 - \Pr(\mathcal{P}_v \models \lor_{j=1}^q \neg R_j).
$$

(4.1)

Using the principle of inclusion and exclusion, we can formulate $\Pr(\mathcal{P}_v \models \lor_{j=1}^q \neg R_j)$ as a sum of probabilities of the following form, where $J$ is a nonempty subset of $\{1, \ldots, q\}$.

$$
\Pr\left(\mathcal{P}_v \models \land_{j \in J} \neg R_j\right)
$$

(4.2)

Suppose that in some branch $R_j$, a descendant edge connects the root $n$ to its only child. So let $R_j = n//T'$ and we define $R_j^1 = n/T'$ and $R_j^2 = n/*//T'$. Clearly, $\neg R_j$ is equivalent to $\neg R_j^1 \land \neg R_j^2$. So, without loss of generality, we assume that in each branch $R_j$ of Equation (4.2), a child edge emanates from the root, or else we replace $\neg R_j$ with $\neg R_j^1 \land \neg R_j^2$. Since $v$ satisfies the root predicate of every $R_j$ (otherwise, Case 2 holds), we get the following.

$$
\Pr\left(\mathcal{P}_v \models \land_{j \in J} \neg R_j\right) = \Pr\left(\mathcal{P}_{u_1} \models \land_{j \in J} \neg T_j\right).
$$
To compute the right side, we first use the equation:

\[
\Pr \left( \mathcal{P}_{u_1} \models \bigwedge_{j \in J} \neg T_j \right) = 1 - \Pr \left( \mathcal{P}_{u_1} \models \bigvee_{j \in J} T_j \right)
\]  

(4.3)

Then, we apply the principle of inclusion and exclusion in order to convert the probability on the right side to a sum of probabilities of the form \( \Pr(\mathcal{P}_{u_1} \models \bigwedge_{j \in J'} T_j) \), where \( J' \) is a nonempty subset of \( J \). Note that each \( T_j \) is either a leaf or a tree that is equivalent to the conjunction of its branches (which are in \( \mathcal{R} \)). Consequently, each probability in the sum has already been computed in a previous iteration.

**Ordinary Node and Multiple Children**

We now consider the case where \( v \) is ordinary and has multiple children \( u_1, \ldots, u_k \). The difficulty of computing \( \Pr(\mathcal{P}_v \models \bigwedge \mathcal{R}) \) in this case is that two branches of the random document can satisfy the same pattern of \( \mathcal{R} \) (or, more generally, intersecting subsets of \( \mathcal{R} \)). Therefore, we use again the principle of inclusion and exclusion and, as a result, get expressions that are evaluated by considering each branch of the document separately. This is done as follows.

We start by rewriting \( \Pr(\mathcal{P} \models \bigwedge \mathcal{R}) \) as described at the beginning of the previous case (ordinary node and one ordinary child). In other words, we use Equation (4.1), and then apply the principle of inclusion and exclusion to convert the probability on the right side of Equation (4.1) into a sum of probabilities of the form shown in Equation (4.2). The negated branch \( \neg R_j \) (\( j \in J \)) is satisfied in a document \( d \) if and only if it is satisfied in every branch of \( d \). (In contrast to \( R_j \) that only has to be satisfied in one branch of \( d \).) Since we assume that \( v \) is ordinary, distinct branches of \( \mathcal{P}_v \) are probabilistically independent. Recall that \( \mathcal{P}^v_{u_1} \) is the branch of \( \mathcal{P}_v \) that contains the child \( u_i \) of \( v \). Then,

\[
\Pr \left( \mathcal{P}_v \models \bigwedge_{j \in J} \neg R_j \right) = \prod_{i=1}^k \Pr \left( \mathcal{P}^v_{u_i} \models \bigwedge_{j \in J} \neg R_j \right).
\]

Consequently, it suffices to compute the probability for each branch of \( \mathcal{P}_v \). So, consider
the branch $\tilde{v}_\Delta$. If $u_i$ is an ordinary node, then we continue exactly as described in the previous case below Equation (4.2). Otherwise, similarly to the last part of the previous case (starting at Equation (4.3)), we use the equation for the complement probability and the principle of inclusion and exclusion, so that the probabilities to be computed are of the following form (where $J'$ is a nonempty subset of $J$).

$$\Pr\left(\mathcal{P}_{\tilde{v}_\Delta} = \bigwedge_{j \in J'} R_j \right)$$

Finally, we proceed as in the case where $v$ has one distributional child.

**Distributional Node**

The last case is where $v$ is distributional. Let $W_1, \ldots, W_k$ be the sets of children of $v$ for which a probability $p^v(W_i)$ is specified. For $1 \leq i \leq k$, we create the p-document $\tilde{\mathcal{P}}_{W_i}^\circ$ from $\tilde{\mathcal{P}}_v$ by removing $v$ and making each node of $W_i$ a child of $\circ$ (which is the root of $\tilde{\mathcal{P}}_v$); the other children of $v$ and their descendants are deleted. By the law of total probability,

$$\Pr\left(\mathcal{P}_v \models \land \hat{\mathcal{R}}\right) = \sum_{i=1}^{k} \Pr\left(\mathcal{P}_{W_i}^\circ \models \land \hat{\mathcal{R}}\right) \cdot p^v(W_i) .$$

The computation of each probability $\Pr(\mathcal{P}_{W_i}^\circ \models \land \hat{\mathcal{R}})$ is done as described in the previous cases (since $\circ$ is ordinary).

**Running-Time Analysis**

The next proposition shows that EvalDP is FPT (see the complexity measures in Section 2.1) and, moreover, it runs in linear time in $\tilde{\mathcal{P}}$ under data complexity.

**Proposition 4.7.2** EvalDP terminates in $O(6^3|T||\tilde{\mathcal{P}}|)$ time.

**Proof.** The size of a node $v$, denoted by $|v|$, is the out degree of $v$ if $v$ is ordinary; and if $v$ is an exp node, then $|v|$ is the total size of the subsets specifying the distribution.
For each visited node $v$, we compute the probability $\Pr(\tilde{\mathcal{P}}_v \models \land \tilde{\mathcal{R}})$. If $v$ is ordinary, then the most expensive cases are when it has multiple children. In these cases, we repeat the following for each child $u$ of $v$. We first apply the principle of inclusion and exclusion to get expressions of the form $\Pr(\tilde{\mathcal{P}}_v \models \land \tilde{R}_j)$. After this application, we may need to add to each conjunction $\land_{j \in J} \tilde{R}_j$ additional conjuncts in order to deal with descendant edges. Consequently, the second application is done over a set of at most $2|J|$ conjuncts. Then, we again apply the principle of inclusion and exclusion. The cost of the second application is $\Theta(2^{2|J|})$. Let $N$ denote $|\mathcal{R}|$, $\tilde{N}$ denote $|\tilde{\mathcal{R}}|$ and $j$ denote $|J|$. The cost of computing $\Pr(\tilde{\mathcal{P}}_v \models \land \tilde{\mathcal{R}})$ is

$$\Theta \left( |v| \cdot \sum_{j=0}^{\tilde{N}} \binom{\tilde{N}}{j} 2^{2j} \right) = \Theta \left( |v| \cdot \sum_{j=0}^{\tilde{N}} \binom{\tilde{N}}{j} 4^j \right) = \Theta \left( 5^{\tilde{N}}|v| \right),$$

and the total cost of visiting $v$ is

$$\Theta \left( \sum_{N=0}^{N} \binom{N}{\tilde{N}} 5^{\tilde{N}}|v| \right) = \Theta \left( 6^{N}|v| \right).$$

If $v$ is a distributional node, then we repeat the above process for each subset $W_i$ (where in each repetition, $|W_i|$ is the out degree of $v$). Therefore, the total cost is again $\Theta(6^N|v|)$. Then, the proposition immediately follows from the fact that $N = |\mathcal{R}| \leq 3|T|$.

### 4.8 Approximate Evaluation

Query evaluation is intractable over $\PrXML^{\{\text{cie}\}}$ (Theorem 4.7.1), but there are efficient randomized approximations even over $\PrXML^{\{\text{exp.cie}\}}$, as we show next.

We define two notions of approximation, given a fixed error factor $\epsilon \ (0 < \epsilon < 1)$. Let $C : Q(\tilde{\mathcal{P}}) \to [0,1]$ be the mapping from the tuples of $Q(\tilde{\mathcal{P}})$ to their correct probabilities, that is, for all $t \in Q(\tilde{\mathcal{P}})$, $C(t) = \Pr(t \in Q(\tilde{\mathcal{P}}))$. The mapping $A : Q(\tilde{\mathcal{P}}) \to [0,1]$ is an additive approximation of $C$ if for all $t \in Q(\tilde{\mathcal{P}})$, $C(t) - \epsilon \leq A(t) \leq C(t) + \epsilon$. The mapping $A$ is a multiplicative approximation of $C$ if for all $t \in Q(\tilde{\mathcal{P}})$, $(1 - \epsilon)C(t) \leq A(t) \leq (1 + \epsilon)C(t)$.
We only consider randomized algorithms that compute an approximation $A$ of $C$ with probability at least $1 - \delta$, where $\delta (0 < \delta < 1)$ is a fixed unreliability factor\textsuperscript{6}.

As in the case of exact query evaluation, we reduce the evaluation of general queries to that of Boolean ones. This is done as follows. Consider a general query $Q$. The first step is to compute $Q(\tilde{P})$. This is done by first finding a superset of $Q(\tilde{P})$ as described at the beginning of Section 4.6 and then detecting the answers in the superset with a nonzero probability, e.g., by computing the exact probability of each match as we describe later in this section. For each match with a nonzero probability, we apply the projection, and in the case of the value-based semantics, replace the nodes with their labels. By the reduction of Section 4.6, one has to apply a randomized approximation algorithm for Boolean queries $n$ times, where $n$ is the number of tuples of $Q(\tilde{P})$, in order to obtain an approximation of $C$. If an unreliability factor $\delta' = \delta/n$ is used in the algorithm for Boolean queries, then the resulting approximation of $C$ has a probability of at least $1 - \delta$ (by the union bound).

So the goal is to approximate the value $\Pr(\mathcal{P} \models T)$, for a p-document $\tilde{P} \in \mathsf{PrXML}^{\{\exp, \cie\}}$ and a pattern $T$. An efficient additive approximation is computed by applying Bernstein’s inequality. That is, we repeatedly (and independently) sample $\tilde{P}$ and take the ratio of samples $d$, such that $d \models T$. In the next section, we show that, under data complexity, Boolean twig queries can also be efficiently multiplicatively approximated. Thus, we obtain the following formal result.

**Theorem 4.8.1** Both o-queries and v-queries can be efficiently approximated (additively and multiplicatively) over $\mathsf{PrXML}^{\{\exp, \cie\}}$, under data complexity.

### 4.8.1 Proof of Multiplicative Approximation

To obtain a multiplicative approximation of a Boolean query, we adapt the Monte Carlo algorithm of \cite{73}, similarly to \cite{37, 93}.

\textsuperscript{6}The results of this section hold for $\delta = 2^{-\text{poly}(N)}$ and $\epsilon = 1/\text{poly}(N)$, where $N$ is the size of the given p-document.
The Main Idea

The idea behind this algorithm is as follows. Consider a pattern $T$ and let $\varphi_1, \ldots, \varphi_m$ be all the possible mappings from $T$ to documents $d \in pwd(\tilde{\mathcal{P}})$. Denote by $\varphi_i \prec \mathcal{P}$ the event “the whole image of $\varphi_i$ is included in the random instance $\mathcal{P}$.” Then,

$$\Pr (\mathcal{P} | T) = \Pr \left( \bigvee_{i=1}^{m} \varphi_i \prec \mathcal{P} \right) = \sum_{i=1}^{m} \Pr \left( \varphi_i \prec \mathcal{P} \land \bigwedge_{j=1}^{i-1} \neg (\varphi_j \prec \mathcal{P}) \right) = \sum_{i=1}^{m} \Pr (\varphi_i \prec \mathcal{P}) \cdot \Pr \left( \bigwedge_{j=1}^{i-1} \neg (\varphi_j \prec \mathcal{P}) \mid \varphi_i \prec \mathcal{P} \right). \quad (4.4)$$

For $1 \leq i \leq m$, let $p_i$ be the probability $\Pr (\bigwedge_{j=1}^{i-1} \neg (\varphi_j \prec \mathcal{P}) \mid \varphi_i \prec \mathcal{P})$. Later, we show the following for all $1 \leq i \leq m$. First, $\Pr (\varphi_i \prec \mathcal{P})$ can be efficiently computed. Second, we can efficiently sample a document from the subspace that satisfies $\varphi_i \prec \mathcal{P}$ (that is, randomly choose a document $d$ in this subspace with the probability $\Pr (\mathcal{P} = d \mid \varphi_i \prec \mathcal{P})$). Consequently, we can additively approximate $p_i$ (using Bernstein’s inequality) efficiently, for some factors $\delta'$ and $\epsilon'$ (that will be determined later). With a probability of at least $1 - \delta' m$ (using the union bound), we obtain values $a_1, \ldots, a_m$, such that $p_i - \epsilon' \leq a_i \leq p_i + \epsilon'$ for all $1 \leq i \leq m$. So by using $a_i$ instead of the real probability $p_i$ in Equation (4.4), we get the following.

$$\sum_{i=1}^{m} \Pr (\varphi_i \prec \mathcal{P}) \cdot a_i \leq \sum_{i=1}^{m} \Pr (\varphi_i \prec \mathcal{P}) \cdot p_i + \epsilon' \sum_{i=1}^{m} \Pr (\varphi_i \prec \mathcal{P}) = \Pr (\mathcal{P} \models T) + \epsilon' \sum_{i=1}^{m} \Pr (\varphi_i \prec \mathcal{P})$$

We use the fact that for all $1 \leq i \leq m$, the event $\varphi_i \prec \mathcal{P}$ is contained in the event $\mathcal{P} \models T$, and hence, $\Pr (\varphi_i \prec \mathcal{P}) \leq \Pr (\mathcal{P} \models T)$. Consequently,

$$\sum_{i=1}^{m} \Pr (\varphi_i \prec \mathcal{P}) \cdot a_i \leq (1 + \epsilon' m) \cdot \Pr (\mathcal{P} \models T).$$
We similarly show that
\[
\sum_{i=1}^{m} \Pr(\varphi_i \triangleleft \mathcal{P}) \cdot a_i \geq (1 - \epsilon' m) \cdot \Pr(\mathcal{P} \models T).
\]

Therefore, choosing \( \delta' = \delta/m \) and \( \epsilon' = \epsilon/m \) yields the desired approximation. It is left to show how to compute the (exact) probability of a match and how to sample \( \tilde{\mathcal{P}} \) conditioned on the existence of a specific match. Formally, we are given a p-document \( \tilde{\mathcal{P}} \in \text{PrXML}^{\{\text{exp, cie}\}} \) and a match \( \varphi \) of \( T \) in some document in \( \text{pwd}(\tilde{\mathcal{P}}) \). The next two sections show how to efficiently compute \( \Pr(\varphi \triangleleft \mathcal{P}) \), and how to efficiently sample the subspace of \( \tilde{\mathcal{P}} \) that comprises the documents containing the image of \( \varphi \).

We denote by \( s_\varphi \) the minimal r-subtree of \( \tilde{\mathcal{P}} \) that contains the image of \( \varphi \) (that is, \( s \) is obtained from \( \tilde{\mathcal{P}} \) by removing every node that is not an ancestor of a node in the image of \( \varphi \)). For each \( \text{exp} \) node \( v \) of \( s_\varphi \), we denote by \( C^v_\varphi \) the set of children of \( v \) that appear in \( s_\varphi \).

Now suppose that \( v \) specifies a probability for the sets of children \( W_1, \ldots, W_l \). By \( p(C^v_\varphi) \) we denote the probability that the children of \( v \) in \( \mathcal{P}^\Sigma \) include all the nodes of \( C^v_\varphi \), given that \( v \) itself is in \( \mathcal{P}^\Sigma \); that is,
\[
p(C^v_\varphi) \overset{\text{def}}{=} \sum_{i: C^v_\varphi \subseteq W_i} p^v(W_i).
\]

**Computing the Probability of a Match**

Observe that \( \Pr(\varphi \triangleleft \mathcal{P}) \) is the probability that \( s_\varphi \) is an r-subtree of \( \mathcal{P}^\Sigma \) (see Section 4.2 for the definition of the random variable \( \mathcal{P}^\Sigma \)). Let \( v_1, \ldots, v_m \) be all the \( \text{exp} \) nodes that appear in \( s_\varphi \). We denote by \( p^{\text{exp}} \) the product \( \prod_{i=1}^{m} p(C^v_i) \). Now let \( L_\varphi \) be the set of all literals that appear in conjunctions \( \alpha^u(w) \), such that \( u \) is a \( \text{cie} \) node and \( w \) appears in \( s_\varphi \). If \( L_\varphi \) contains an event and its negation, then \( \Pr(\varphi \triangleleft \mathcal{P}) = 0 \). So suppose otherwise. By \( p^{\text{cie}} \) we denote the joint probability of \( L_\varphi \), i.e., the product \( \prod_{a \in L_\varphi} p(a) \), where \( p(a) = p(e) \) if \( a \) is the variable \( e \) and \( 1 - p(e) \) if \( a \) is \( \neg e \). It is easy to show that \( \Pr(\varphi \triangleleft \mathcal{P}) = p^{\text{exp}} \cdot p^{\text{cie}} \).
Sampling Conditioned on a Match

Our goal now is to randomly generate a p-document \( d \), such that \( \varphi \in \mathcal{M}(T, d) \) and the probability of generating \( d \) is \( \Pr(\mathcal{P} = d \mid \varphi \in \mathcal{P}) \). We do that by sampling the p-document \( \tilde{\mathcal{P}}|\varphi \) that is obtained from \( \tilde{\mathcal{P}} \) as follows. We start with \( \tilde{\mathcal{P}}|\varphi = \tilde{\mathcal{P}} \). We traverse the nodes of \( s_\varphi \) and, when we visit a distributional node \( v \), we change the distribution of \( v \) in \( \tilde{\mathcal{P}}|\varphi \) according to the type of \( v \).

**Case 1:** \( v \) is of type \( \text{cie} \). Let \( L \) be the set of all literals that appear in the conjunctions \( \alpha^v(w) \), where \( w \) is a child of \( v \) in \( s_\varphi \). For each literal of \( e_i \) of \( L \), we change the probability of \( e_i \) to 1. Similarly, for each literal of \( \neg e_j \) of \( L \), we change the probability of \( e_j \) to 0.

**Case 2:** \( v \) is of type \( \text{exp} \). For each probability \( p^v(W_i) \) specified by \( v \), if \( C_\varphi \subseteq W_i \), then we change \( p^v(W_i) \) to \( p^v(W_i) / p(C_\varphi) \); otherwise, it is changed to zero.

Having constructed \( \tilde{\mathcal{P}}|\varphi \), we simply sample it by invoking its distributional nodes. The correctness of our sampling method is shown by the following proposition, which is quite straightforward.

**Proposition 4.8.2** \( \Pr(\mathcal{P} = d \mid \varphi \in \mathcal{P}) = \Pr(\tilde{\mathcal{P}}|\varphi = d) \) for all \( d \in \text{pwd}(\mathcal{P}) \).

Note that we only use \( \mathcal{P}|\varphi \) for clearance of the presentation; there is no need to materialize it. It is enough to do the following. First, when invoking a \( \text{cie} \) node, we randomly choose occurrences of the events \( e_i \) that do not appear in \( L \). Second, when invoking an \( \text{exp} \) node, we consider only the children sets \( W_i \) that contain \( C \) (while normalizing their probability).

**4.8.2 Query-and-Data Complexity of Approximations**

We conclude this section by considering the query-and-data complexity of approximate query evaluation. We begin with Boolean queries. The following theorem shows that, while additive approximation is tractable over \( \text{PrXML}^{\{\text{exp,cie}\}} \), multiplicative approximation is intractable even for the restricted family \( \text{PrXML}^{\{\text{max}\}} \). The proof is in Appendix E.

**Theorem 4.8.3** Under query-and-data complexity,
• Additively approximating Boolean queries is tractable over $\text{PrXML}^{\{\exp, \text{cie}\}}$.

• Unless $\text{NP} \subseteq \text{BPP}$\footnote{BPP is the class of decision problems that can be computed by efficient randomized algorithms with \textit{“bounded error”} (which in our terminology means that $\delta = 1/3$ and there is no error factor). $\text{NP} \subseteq \text{BPP}$ is considered unlikely \cite{111}.}, there is no efficient randomized algorithm for multiplicatively approximating Boolean queries over $\text{PrXML}^{\{\text{mux}\}}$.

The first part of the Theorem 4.8.3 does not hold for general queries, because we cannot even generate the set $Q(\tilde{P})$ efficiently (i.e., in polynomial total time \cite{68}). The reason is that Lemma E.2.1 (which is used in Appendix E for proving Theorem 4.8.3) actually shows that over $\text{PrXML}^{\{\text{mux}\}}$, it is NP-hard to determine whether $Q(\tilde{P})$ is nonempty if $Q = \pi_X T^v$ (or $Q = \pi_X T^o$) and $X$ contains all the nodes of $T$. However, we can easily extend the first part to general queries if the running time is measured as a function of $T$, $\tilde{P}$ and $Q(d)$ (rather than $T$, $\tilde{P}$ and $Q(\tilde{P})$), where $d$ is the document that is obtained from $\tilde{P}$ by removing the distributional nodes as done in the operation $\text{doc}(\cdot)$ (defined in Section 4.2 right after Example 4.2.1).
Chapter 5

Incorporating Constraints in Probabilistic XML

In this chapter, we present the probabilistic XML database (PXDB) model which is the probability sub-space comprising the instances of a p-document of $\text{PrXML}^{\text{exp}}$ that satisfy a set of constraints.

5.1 Overview

In Chapter 4, we showed that within the current state-of-art models of probabilistic XML, there is a clear tradeoff between the efficiency of query evaluation and the ability to model probabilistic dependencies. On the one hand, the efficient algorithm, $\text{EvalDP}$ for evaluating twig queries with projection in $\text{PrXML}^{\text{exp}}$ (which generalizes $\text{PrXML}^{\text{cie}}$, the simple probabilistic trees of $\text{PrXML}^{\text{cie}}$, and the tree-structured probabilistic instances of $\text{PrXML}^{\text{cie}}$) was given. But for $\text{PrXML}^{\text{cie}}$ (i.e., the probabilistic-tree model of $\text{PrXML}^{\text{cie}}$), on the other hand, is quite different. Query evaluation for non-trivial Boolean tree queries is #P-complete (Theorem 4.7.1). Thus, typically, either query evaluation is intractable or the model has limited expressiveness.

In this chapter, we present PXDBs, a novel way of modeling probabilistic XML. In this model, a probabilistic database consists of both a p-document $\tilde{\mathcal{D}} \in \text{PrXML}^{\text{exp}}$, as well as a set $\mathcal{C}$ of constraints. Constraints are important, first from the traditional database
point of view, since they are a mechanism for maintaining data integrity. Even more significantly, constraints are important also from the probabilistic aspect because they implicitly capture natural probabilistic dependencies among data items. In addition, constraints can be derived immediately from user-knowledge about real-world requirements, and as such, are expected to be easy to formulate. To the best of our knowledge, this is the first work to consider combining probabilistic data with a rich language of constraints. (In the past, e.g. [94], only key constraints were considered.)

To demonstrate the type of constraints that we can express, consider the p-document of Figure 5.1. Suppose that we have the real-world constraint that a department chair must be a full professor (and not an assistant professor), and that a department has only one chair. In addition, the university may have a policy that an assistant professor guides at most one Ph.D. student. All of these constraints can be expressed in our model. Now, whether or not David is a Ph.D. student of Lisa depends not only on his ancestor nodes, but also on Lisa’s position (which, in turn, depends on the position of Mary). Even more intricate probabilistic dependencies may be expressed by our model, e.g., by using constraints that are in themselves probabilistic, e.g., “with a probability of 95%, at least 80% of the chairs are full professors.” Note that the probabilistic dependencies are in no case explicitly stated, but are simply implied by the constraints.

Although the nodes of our probabilistic XML have complex interdependencies, we show that three pivotal tasks can be performed efficiently. The first is that of testing well-definedness of a PXDB, i.e., non-emptiness of the probability sub-space (of the p-document) comprising the documents that satisfy the constraints. The second task is query evaluation. The third task is sampling, i.e., randomly generating a document in the distribution of a given PXDB, where the probability of generating each document is that of being generated by the PXDB at hand. This task is straightforward in all the models of probabilistic data that are considered in this thesis, but not in PXDBs.

Bayesian networks (cf. [88]) are commonly used for describing probabilistic dependencies. However, it is not clear how natural constraints and queries can be efficiently translated into Bayesian networks. Furthermore, it is unlikely that Bayesian networks have tractability
results similar to those described in this chapter, because determining (or approximating) the probability of very simple events (i.e., inference) in Bayesian networks is intractable \cite{34,35}.

5.2 Constraints

In this section, we define the notion of constraints. For that, we first define selectors and the count operator.

A selector is a query of the form $\pi_X T^o$, where $X$ is a projection sequence of length one. By a slight abuse of notation, we do not distinguish between a sequence of length 1 and its single element. For example, we write $\pi_n T^o$ instead of $\pi(n) T^o$, and we view the result of $\pi_n T^o$ as a set of document nodes. Formally, given a document $d$, the selector $S = \pi_n T^o$ selects the set of nodes $S(d) \overset{\text{def}}{=} \{\varphi(n) \mid \varphi \in \mathcal{M}(T, d)\}$.

Let $U$ be a set of nodes of a document $d$. The aggregate function Cnt returns the size of $U$, namely, $\text{Cnt}(U) = |U|$. In particular, if $S$ is a selector, then $\text{Cnt}(S(d))$ is the number of nodes of $d$ that are selected by $S$.

5.2.1 Constraints

The definition of a constraint is as follows.

**Definition 5.2.1 (Constraints)** A constraint is a Boolean expression of the form $\forall S (\text{Cnt}(S_1) \otimes_1 N_1 \rightarrow \text{Cnt}(S_2) \otimes_2 N_2)$, where

- $S, S_1$ and $S_2$ are selectors,
- $\otimes_1$ and $\otimes_2$ are numerical-comparison operators from $\{=, \neq, <, \leq, >, \geq\}$, and
- $N_1$ and $N_2$ are integers.

Given a document $d$, the quantifier $\forall S$ iterates over all the nodes that are selected by $S$. For each selected node $v$, both $S_1$ and $S_2$ are applied to the subtree $d^v_\Delta$. Formally, a
document $d$ satisfies $\forall S (\text{Cnt}(S_1) \otimes_1 N_1 \rightarrow \text{Cnt}(S_2) \otimes_2 N_2)$ if the following formula is true.

$$\forall v \in S(d) \ (\text{Cnt}(S_1(d^v_\Delta)) \otimes_1 N_1 \rightarrow \text{Cnt}(S_2(d^v_\Delta)) \otimes_2 N_2)$$

We use $\mathcal{C}$ to denote a finite set of constraints. If $d$ satisfies every constraint of $\mathcal{C}$, then we write $d \models \mathcal{C}$.

**Example 5.2.2** The constraints $C_1, \ldots, C_4$ of Figure 5.1 are defined by means of the patterns and selectors shown in the top part of that figure. In this example, we use a shorthand notation as follows. First, a pattern $T$ (e.g., in $C_3$) stands for the comparison $\text{Cnt}(\pi_{\text{root}(T)} T) > 0$, namely, there exists a match of $T$. Second, the symbol $\ast$ represents the pattern that consists of one node with the predicate $\text{true}$.

The constraint $C_1$ says that a department cannot have more than one chair. According to $C_2$, a department with 3 or more professors must have a chair. $C_3$ requires a member to be a full professor in order to be a chair. Finally, $C_4$ says that an assistant professor can supervise at most one Ph.D. student.

When checking satisfaction of $C_2$ and $C_4$, we map the root of $S_{\text{mem}}$ to department nodes and to the root of $d$, respectively. The document $d$ of Figure 5.2 satisfies each of $C_1, \ldots, C_4$. Note that if Mary was not a chair (i.e., the corresponding position node did not have a chair child), then $d$ would violate $C_2$. Similarly, if Lisa was an assistant (rather than full) professor, then $d$ would violate $C_4$.

### 5.3 Probabilistic XML Databases (PXDBs)

In this section, we define the model of *probabilistic XML databases (PXDB)*. A PXDB is defined in terms of a p-document of $\mathbb{P}\text{PrXML}^{(\text{exp})}$ and a set of constraints. Informally, the PXDB is the sub-space comprising all the documents that satisfy the constraints. Next, we give the formal definition.

\footnote{Recall from Section 4.3 that this family generalizes the probabilistic XML models of $[65, 80, 103]$ and other models, as described in Figure 4.2 (Chapter 4).}
A PXDB $\tilde{D} = (\tilde{P}, C)$ describing information obtained by screen-scraping a university Web site

We say that a p-document $\tilde{P}$ is consistent with a set $C$ of constraints if $Pr(\tilde{P} \models C) > 0$. A PXDB is a probability space defined by a pair $(\tilde{P}, C)$, such that $C$ is a set of constraints and $\tilde{P} \in \PrXML\{\text{exp}\}$ is a p-document that is consistent with $C$. The PXDB $\tilde{D} = (\tilde{P}, C)$ comprises all documents $d$ of $\tilde{P}$, such that $d \models C$ and $Pr(\tilde{P} = d) > 0$. We denote the set of these documents by $pwd(\tilde{D})$. The probability distribution of $\tilde{D} = (\tilde{P}, C)$ is that of $\tilde{P}$ conditioned on satisfying $C$. That is, the probability of a document $d \in pwd(\tilde{D})$ is $Pr(\tilde{P} = d | \tilde{P} \models C)$.

Example 5.3.1 Figure 5.1 depicts the PXDB $\tilde{D} = (\tilde{P}, C)$, where $\mathcal{C} = \{C_1, \ldots, C_4\}$. Figure 5.2: A random instance $d$ of the PXDB $\tilde{D}$ of Figure 5.1.
Example 5.2.2 describes the set of constraints $\mathcal{C}$. For graphical convenience, we use ind and mux nodes directly without translating them into exp nodes as described in Theorem D.1.1.

We use $\mathcal{D}$ to denote the random variable that represents a document of the probability space $\tilde{\mathcal{D}}$. For all documents $d$ (by the definition of conditional probability),

$$
\Pr(\mathcal{D} = d) = \begin{cases} 
\frac{\Pr(\mathcal{P} = d)}{\Pr(\mathcal{P} \models \mathcal{C})} & \text{if } d \models \mathcal{C}, \\
0 & \text{otherwise.}
\end{cases}
$$

Example 5.3.2 To illustrate the dependencies that exist among the nodes of a PXDB, we use the p-document $\tilde{\mathcal{P}}$ and the PXDB $\tilde{\mathcal{D}}$ of Figure 5.1. Let us consider the probability of the event “the Ph.D. student Amy appears in a random document,” in both $\tilde{\mathcal{P}}$ and $\tilde{\mathcal{D}}$. In the case of $\tilde{\mathcal{P}}$, this probability is 0.54, and it is easily computed by multiplying the probabilities on the path from the root to Amy.

Now suppose that the probability space is $\tilde{\mathcal{D}}$ rather than $\tilde{\mathcal{P}}$. Due to $C_4$, this event depends on two other events: whether Lisa is an assistant or a full professor, and whether she has other Ph.D. students. In turn, $C_3$ implies that the former event depends on whether Lisa is a chair. But due to $C_1$, Lisa cannot be a chair if Mary is. Whether Mary is a chair depends on her rank (i.e., assistant or full professor) and, due to $C_2$, also on whether Paul exists as a member (otherwise, there are fewer than 3 members, and hence, the department does not necessarily have a chair). So, there does not seem to be any obvious way of computing the probability that Amy appears in $\tilde{\mathcal{D}}$.

The above example shows that the probabilistic dependencies among data items of a PXDB go far beyond the tree structure of the p-document. In particular, they involve nodes that are not related by the ancestor-descendant relationship.

In Section 5.7, we consider additional types and interpretations of constraints that extend the capability to model real-world situations. Two different examples are: “80% of the chairs are full professors,” and “with probability 0.8, all chairs are required to be full professors.”
5.4 Computational Problems

Before defining three computational problems, we explain the relevant complexity measure.

We use the notion of data complexity in order to determine tractability of the problems at hand. This means that queries and constraints are assumed to be fixed while the input consists of the p-document. Note, however, that the numerical operands in the Cnt functions (used in the constraints) may be large. Consequently, “polynomial data complexity” might not be a practical yardstick of efficiency. Therefore, we apply the following twist, as done in [94]. In addition to the p-document, the input contains a numerical specification of the constraints. The numerical specification simply gives the number \( N \) in every comparison \( \text{Cnt}(S) \otimes N \) (while \( S \) and \( \otimes \) remain fixed).

The first problem that we study is deciding whether a PXDB is well-defined, namely, whether a given p-document is consistent with a set of constraints. More generally, the constraint-satisfaction problem (defined below) is computing the probability that a p-document satisfies a set of constraints.

**Problem 1 (Constraint Satisfaction)** For a set \( \mathcal{C} \) of constraints, \( \text{C-Sat}(\mathcal{C}) \) denotes the problem of determining the probability \( \Pr(\mathcal{P} \models \mathcal{C}) \), where the input is a p-document \( \tilde{\mathcal{P}} \in \text{PrXML}^{\{\text{exp}\}} \) and a numerical specification of \( \mathcal{C} \).

The next problem is query evaluation; namely, to compute the probability of each tuple (as done in Section 4.7). Consider a PXDB \( \tilde{\mathcal{D}} = (\tilde{\mathcal{P}}, \mathcal{C}) \) and a query \( Q \) (of the form \( \pi_X T^o \) or \( \pi_X T^v \)). We denote by \( Q(\tilde{\mathcal{D}}) \) the set of all tuples \( t \), such that \( t \in Q(d) \) for some document \( d \in \text{pwd}(\tilde{\mathcal{D}}) \). The result of applying \( Q \) to \( \tilde{\mathcal{D}} \), denoted by \( Q(\tilde{\mathcal{D}}) \), is the mapping \( \Phi : Q(\tilde{\mathcal{D}}) \to [0, 1] \), such that for all \( t \in Q(\tilde{\mathcal{D}}) \), it holds that \( \Phi(t) = \Pr(t \in Q(\tilde{\mathcal{D}})) \).

**Problem 2 (Query Evaluation)** For a query \( Q \) and a set \( \mathcal{C} \) of constraints, \( \text{Eval}(Q, \mathcal{C}) \) denotes the problem of computing \( Q(\tilde{\mathcal{D}}) \), where \( \tilde{\mathcal{D}} = (\tilde{\mathcal{P}}, \mathcal{C}) \). The input is a numerical specification of \( \mathcal{C} \) and a p-document \( \tilde{\mathcal{P}} \in \text{PrXML}^{\{\text{exp}\}} \) that is consistent with \( \mathcal{C} \).

Sampling is “the act, process, or technique of selecting a representative part of a population for the purpose of determining parameters or characteristics of the whole population”
In our setting, given a PXDB \( \tilde{\mathcal{D}} \), sampling is a randomized method that generates a document \( d \) of \( \tilde{\mathcal{D}} \). This method should properly simulate \( \tilde{\mathcal{D}} \) in the sense that for all documents \( d \), the probability of generating \( d \) is equal to \( \Pr(\mathcal{D} = d) \).

**Problem 3 (Sampling)** For a set \( \mathcal{C} \) of constraints, \( \text{Sample}(\mathcal{C}) \) is the problem of randomly generating a document \( d \in \text{pwd}(\tilde{\mathcal{D}}) \), where \( \tilde{\mathcal{D}} = (\tilde{\mathcal{P}}, \mathcal{C}) \). Every \( d \) is generated with probability \( \Pr(D = d) \). The input is a numerical specification of \( \mathcal{C} \) and a \( p \)-document \( \tilde{\mathcal{P}} \in \text{PrXML}^{\exp} \) that is consistent with \( \mathcal{C} \).

### 5.5 The Evaluation Algorithm

In this section, we show that the constraint-satisfaction and query-evaluation problems are tractable. We do so by solving these problems for \( c \)-formulae that are defined below.

Similarly to Section 4.6, we reduce general queries to Boolean ones. So for query evaluation, our goal is to compute the probability \( \Pr(D \models T') \), given a twig pattern \( T' \) and a PXDB \( \tilde{\mathcal{D}} = (\tilde{\mathcal{P}}, \mathcal{C}) \). By the definition of conditional probability,

\[
\Pr(D \models T') = \Pr(D \models T' | D \models \mathcal{C}) = \frac{\Pr(D \models \mathcal{C} \land T')}{\Pr(D \models \mathcal{C})}.
\]

It thus suffices to first compute \( \Pr(D \models \mathcal{C}) \) and then \( \Pr(D \models \mathcal{C} \land T') \) in order to solve the constraint-satisfaction and query-evaluation problems. We do this by means of \( c \)-formulae that generalize constraints and Boolean queries, and can express both \( \mathcal{C} \) and \( \mathcal{C} \land T' \).

Intuitively, a new \( c \)-formula is built from existing ones in three steps. First, we *augment* a pattern \( T \) by attaching a \( c \)-formula to every node. (When mapping a node \( n \) of \( T \) to a node \( v \) of a document \( d \), the subtree \( d^v_\Delta \) must satisfy the \( c \)-formula attached to \( n \).) Second, we convert the augmented pattern to an *s*-formula, which is a generalized selector. Third, we use one or more s-formulae in order to create a new \( c \)-formula, which is a generalized constraint. Next, we give the mutually recursive definition of augmented patterns, s-formulae and c-formulae; we denote them by \( \alpha T, \sigma \) and \( \gamma \), respectively. The term *formula* refers to
either an s-formula or a c-formula.

**Definition 5.5.1 (Formulae)** There are five cases.

1. The basic c-formulae are **true** and **false**.
2. If $\gamma_1, \ldots, \gamma_m$ are c-formulae, then $\gamma_1 \land \cdots \land \gamma_m$ is a c-formula.
3. If $T$ is a pattern and $\alpha$ is a function that maps every node of $T$ to a c-formula, then $\alpha T$ is an augmented pattern.
4. If $\alpha T$ is an augmented pattern and $n \in \mathcal{V}(T)$, then $\pi_n \alpha T$ is an s-formula.
5. If $N \in \mathcal{Z}$, $\odot \in \{=, \neq, <, \leq, >, \geq\}$ and $\sigma_1, \ldots, \sigma_k$ are s-formulae, then $\gamma = \text{CNT}(\sigma_1 \lor \cdots \lor \sigma_k) \odot N$ is a c-formula.

We now define how to evaluate augmented patterns, s-formulae and c-formulae over a document $d$. An augmented pattern has an associated set of matches. The result of an s-formula $\sigma$, denoted by $\sigma(d)$, is a set of nodes of $d$. A c-formula $\gamma$ gets a Boolean value, and $d \models \gamma$ denotes that this value is **true**. In the following definition, the order of items corresponds to that of Definition 5.5.1.

**Definition 5.5.2 (Evaluation)** Let $d$ be a document.

1. If $\gamma \in \{\text{true}, \text{false}\}$, then $d \models \gamma$ iff $\gamma = \text{true}$.
2. If $\gamma = \gamma_1 \land \cdots \land \gamma_m$ is a c-formula, then $d \models \gamma$ iff $d \models \gamma_i$ for all $1 \leq i \leq m$.
3. If $\alpha T$ is an augmented pattern, then $\mathcal{M}(\alpha T, d)$ is the subset of all matches $\varphi \in \mathcal{M}(T, d)$, such that for all $n' \in \mathcal{V}(T)$, it holds that $d^{\varphi(n')}_{(\Delta)} = \alpha(n')$.
4. If $\sigma = \pi_n \alpha T$ is an s-formula, then $\sigma$ selects the set of nodes $\sigma(d) = \{\varphi(n) \mid \varphi \in \mathcal{M}(\alpha T, d)\}$.
5. If $\gamma = \text{CNT}(\sigma_1 \lor \cdots \lor \sigma_k) \odot N$, then $d \models \gamma$ evaluates to $|\sigma_1(d) \cup \cdots \cup \sigma_k(d)| \odot N$. 
5.5.1 Queries and Constraints as c-Formulae

We now show how to express queries and constraints as c-formulae. We use $T_0$ to denote the trivial pattern, which consists of a single node $r$ that has the predicate true.

An augmented pattern $\alpha T$ and a c-formula $\gamma$ are congruent if for all documents $d$, there is a match of $\alpha T$ in $d$ (i.e., $M(\alpha T, d) \neq \emptyset$) if and only if $d \models \gamma$. Given a c-formula $\gamma$, we can construct a congruent $\alpha T$ by taking the trivial pattern $T_0$ and attaching $\gamma$ to its root. Similarly, given an augmented pattern $\alpha T$, a congruent c-formula is $\text{Cnt}(\pi_r \alpha T) = 1$, where $r$ is the root of $\alpha T$. Hence, $\text{Cnt}(\pi_r \alpha T) = 0$ is an anti-congruent of $\alpha T$, namely, $d \models Cnt(\pi_r \alpha T) = 0$ if and only if $M(\alpha T, d) = \emptyset$. So, the negation of a c-formula $\gamma$ is obtained by converting $\gamma$ to a congruent $\alpha T$ and then constructing an anti-congruent of $\alpha T$. It thus follows that c-formulae are closed under conjunction, negation and disjunction.

From now on, we view every pattern $T$ as an augmented one; if $\alpha$ is not explicitly given, then it maps every node of $T$ to the c-formula true.

A pattern $T'$ (that represents a Boolean query) is expressed as a c-formula by taking the congruent of $T'$. Now, consider a constraint $\forall S(Cnt(S_1) \otimes_1 N_1 \rightarrow Cnt(S_2) \otimes_2 N_2)$ and let $S = \pi_n T$. We express this constraint as a c-formula by first converting $T$ into an augmented pattern $\alpha T$, as described next, and then taking an anti-congruent of $\alpha T$. To obtain $\alpha T$, we attach true to all the nodes of $T$, except for $n$ which gets the c-formula $\text{Cnt}(S_1) \otimes_1 N_1 \land \text{Cnt}(S_2) \bar{\otimes}_2 N_2$, where $\bar{\otimes}_2$ is the complement of $\otimes_2$ (e.g., if $\otimes_2$ is $<$ then $\bar{\otimes}_2$ is $\geq$).

It follows from this discussion that if c-formulae can be evaluated efficiently over p-documents, then both constraint satisfaction and query evaluation can be solved in polynomial time. Evaluation of c-formulae is the subject of the next section.

5.5.2 Evaluating c-Formulae

The following theorem shows that c-formulae can be evaluated efficiently over p-documents.

**Theorem 5.5.3** Consider a c-formula $\gamma$. Computing the probability $\Pr(\mathcal{D} \models \gamma)$, given a
p-document $\tilde{\mathcal{P}} \in \PrXML^{\{\exp\}}$ and a numerical specification of $\gamma$, is in polynomial time.

By the discussion above, we derive the following corollary.

**Corollary 5.5.4** For all queries $Q$ and finite sets of constraints $\mathcal{C}$, both $\text{C-Sat}(\mathcal{C})$ and $\text{Eval}(Q, \mathcal{C})$ can be solved in polynomial time.

In the remainder of this section, we describe the algorithm for evaluating c-formulae, i.e., the proof of Theorem 5.5.3. Since the evaluation algorithm is involved, we will only sketch the main ideas. The complete proof is in Appendix F.

Let $\gamma$ be a c-formula and $\tilde{\mathcal{P}}$ be a p-document. The goal is to compute $\Pr(\mathcal{P} \models \gamma)$. Our strategy is to apply a recursion that reduces the problem of computing $\Pr(\mathcal{P} \models \gamma)$ to problems of computing probabilities of the form $\Pr(\mathcal{P}' \models \gamma')$, where $\mathcal{P}'$ is a proper subtree of $\mathcal{P}$ (and $\gamma'$ is a c-formula). Note that this approach was also used in Section 4.7.1 (namely, the algorithm EvalDB) for the task of evaluating Boolean queries. However, evaluating c-formulae is far more intricate.

To make the discussion clearer, we start by considering a restricted case, where $\gamma$ is of the form $\text{Cnt}(\pi_n \alpha T) = N$, namely, $\gamma$ is a c-formula with a single conjunct, the argument of $\text{Cnt}$ has only one disjunct and $\otimes$ is the equality operator. Later on, we briefly discuss the general case (i.e., when these restrictions are not assumed).

In order to straightforwardly apply a recursive reduction, it is helpful if the c-formula $\gamma$ and the p-document $\tilde{\mathcal{P}}$ satisfy the following conditions:

1. $\pi_n \alpha T$ satisfies the following.
   
   (a) No condition is attached to the root of $T$, i.e., $\text{cond}_{\text{root}(T)} = \text{true}$.
   (b) No c-formula is attached to the root of $T$, i.e., $\alpha(\text{root}(T)) = \text{true}$.
   (c) $\gamma$ counts a non-root element, that is, $\text{root}(T) \neq n$.
   (d) $\text{root}(T)$ has exactly one child.
   (e) The single edge that emanates from $\text{root}(T)$ is a child edge.

2. $\text{root}(\tilde{\mathcal{P}})$ has exactly one child which is ordinary.
If the above conditions hold, then we do the following reduction. Let $T'$ and $\tilde{P}'$ be the pattern and p-document that are obtained from $T$ and $\tilde{P}$, respectively, by removing the root. Let $\gamma'$ be the c-formula $\text{Cnt}(\pi_n \alpha T') = N$. Then

$$\Pr (\mathcal{P} \models \gamma) = \Pr (\mathcal{P}' \models \gamma').$$

(5.1)

Clearly, the conditions above may not hold. We first consider the case where Condition 1 is violated, and show how to transform $\Pr(\mathcal{P} \models \gamma)$ into a sum $\sum_{i=1}^{q} \pm \Pr(\mathcal{P} \models \gamma_i)$, such that each $\gamma_i$ satisfies Condition 1. Afterwards, we discuss violations of Condition 2.

Suppose that Condition 1 does not hold. In this case we transform $\Pr(\mathcal{P} \models \gamma)$ into a sum $\sum_{i=1}^{q} \pm \Pr(\mathcal{P} \models \gamma_i)$, such that each $\gamma_i$ satisfies Condition 1. Initially, the sum consists of only the original probability $\Pr(\mathcal{P} \models \gamma)$. Afterwards, we repeatedly apply transformations to an operand $\pm \Pr(\mathcal{P} \models \gamma_i)$ of the sum, in order to replace it with a new sum. We repeat this step until none of the transformations can be applied to any operand. Our transformations are devised so that upon termination, all the c-formulae $\gamma_i$ of $\sum_{i=1}^{q} \pm \Pr(\mathcal{P} \models \gamma_i)$ will satisfy Condition 1.

We need eight different transformations (which are described in Appendix F). Due to a lack of space we discuss only one, called ExCR (extracting c-formulae from roots). Furthermore, we describe this transformation only for the simple case where $\gamma_i$ is of the form $\text{Cnt}(\pi_n \alpha_i T_i) = 0$. This transformation is applicable if $\alpha_i(\text{root}(T_i)) \neq \text{true}$, that is, Condition (b) is violated. Let $\alpha'_i$ be the same mapping as $\alpha_i$, except that it maps root($T_i$) to true. The transformation ExCR replaces the addend $\Pr(\mathcal{P} \models \text{Cnt}(\pi_n \alpha_i T_i) = 0)$ with the following equal sum.

$$\Pr (\mathcal{P} \models \neg \alpha_i(\text{root}(T_i))) + \Pr (\mathcal{P} \models \alpha_i(\text{root}(T_i)) \land \text{Cnt}(\pi_n \alpha'_i T_i) = 0)$$

Note that $\alpha_i(\text{root}(T_i))$ (i.e., the c-formula attached to the root of $T_i$) does not necessarily satisfy Condition (b). So, the transformation ExCR may have to be applied again to the addends of the above sum.
By repeatedly applying the transformations, we will end up with a summation of the form
\[ \sum_{i=0}^{N} \Pr(\mathcal{P} \models \gamma_i) \], such that every \( \gamma_i \) satisfies Condition 1. If Condition 2 holds, then the recursive reduction (described earlier) can be applied. We now consider the case where only Condition 2 is violated.

We consider again a c-formula \( \gamma \) of the (restricted) form \( \text{Cnt}(\pi_\alpha T) = N \), such that \( \gamma \) satisfies Condition 1. If the root of \( \tilde{\mathcal{P}} \) has exactly one child \( v \), such that \( v \) is distributional, then the subtree of \( \tilde{\mathcal{P}} \) rooted at \( v \) is not a well-defined p-document. We handle this case similarly to Section 4.7.1 by introducing a dummy node \( \circ \) as the parent of \( v \) (further details are omitted due to lack of space). Next, suppose that the root of \( \tilde{\mathcal{P}} \) has multiple children \( w_1, \ldots, w_k \). We partition \( \tilde{\mathcal{P}} \) into two p-documents as follows. \( \tilde{\mathcal{P}}_k \) is the subtree of \( \tilde{\mathcal{P}} \) that consists of the root, the child \( w_k \) and all the descendants of \( w_k \). \( \tilde{\mathcal{P}}_{<k} \) comprises the root and the rest of the nodes of \( \tilde{\mathcal{P}} \), i.e., it is obtained by removing \( w_k \) and its descendants. Clearly, a random document of \( \tilde{\mathcal{P}} \) consists of two parts that are obtained from \( \tilde{\mathcal{P}}_k \) and \( \tilde{\mathcal{P}}_{<k} \). These two parts are probabilistically independent. Since \( \gamma \) satisfies Condition 1(c), \( n \neq \text{root}(T) \). So, \( \Pr(\mathcal{P} \models \gamma) \) can be computed as a sum of \( N + 1 \) products, namely,

\[ \sum_{i=0}^{N} \Pr(\mathcal{P}_{<k} \models \text{Cnt}(\pi_\alpha T) = i) \cdot \Pr(\mathcal{P}_k \models \text{Cnt}(\pi_\alpha T) = N - i) \]. \hspace{1cm} (5.2)

The first term in each product is computed recursively by partitioning \( \tilde{\mathcal{P}}_{<k} \) into two p-documents as described above. The second term is computed by applying Equation (5.1).

Our discussion above sketched the main ideas of our algorithm for the restricted form of \( \gamma \) defined above. We conclude the description of the algorithm with a few remarks on the details needed for more general forms of c-formulae. Some transformations introduce conjunctions of c-formulae and disjunctions of s-formulae (which are defined in Parts 2 and 5, respectively, of Definition 5.5.1). Therefore, the above description of the algorithm is an oversimplification due to the assumption about the restricted form of the c-formula \( \gamma \). In particular, the handling of multiple children is more complicated than that described by Equation (5.2), because we need to evaluate a conjunction of c-formulae \( \gamma_1 \land \cdots \land \gamma_l \) rather

\[ \text{Note that we can assume that } N \text{ is at most the number of (ordinary) nodes of } \mathcal{P}, \text{ so this summation is efficient.} \]
than just an atomic constraint $\text{CNT}(\pi_n \alpha T) = N$ (see Appendix F for details). A second remark is that constraints with inequalities are handled by one of the transformations that replaces any $\otimes$ comparison with equality.

To prove correctness of our algorithm, we show that the repeated execution of the transformations terminates. Moreover, for proving efficiency, we show the following two facts. First, there is a polynomial (in the size of $\mathcal{P}$ and the numerical specification) upper bound on the number of repeated applications (when starting with a given c-formula $\gamma$). Second, the result of applying the transformations (at each node of $\mathcal{P}$) has a polynomial size.

The proof of efficiency also requires to show that there is a polynomial upper bound on the overall number of probabilities that need to be evaluated throughout the entire algorithm (i.e., including the recursive steps). This follows from the fact that memoing \cite{memoing} is used to avoid an exponential blowup during the recursion (and from the second fact above).

5.6 Sampling PXDBs

In this section, we present an efficient sampling algorithm for PXDBs. The formal result is the following.

**Theorem 5.6.1** Let $\mathcal{C}$ be a finite set of constraints. There is an efficient randomized algorithm for solving the problem $\text{Sample}(\mathcal{C})$.

Throughout this section, we assume that $\mathcal{C}$ is a fixed set of constraints. The input consists of a p-document $\mathcal{P} \in \text{PrXML}^{\text{exp}}$ and a numerical specification of $\mathcal{C}$. Furthermore, $\mathcal{P}$ is consistent with $\mathcal{C}$, and $\mathcal{P}$ is the PXDB $(\mathcal{P}, \mathcal{C})$. By $\mathcal{W}(\mathcal{P})$ we denote the set of all pairs $(v, W)$, such that $v$ is a distributional node of $\mathcal{P}$ that specifies a probability $\mathcal{P}(v, W)$ for the subset $W$ of its children. We assume that $\mathcal{W}(\mathcal{P}) = \{(v_1, W_1), \ldots, (v_m, W_m)\}$.

In the rest of this section, we describe the algorithm of Figure 5.3, called $\text{Sample}(\mathcal{C})(\mathcal{P})$, that constructs a sample with the appropriate probability. For simplicity of presentation, we omit the numerical specification of $\mathcal{C}$ from the input of $\text{Sample}(\mathcal{C})$. The underlying idea is rather simple. The loop of Line 4 iterates over $\mathcal{W}(\mathcal{P})$. In the $i$th iteration, the task is to randomly determine whether to choose $(v_i, W_i)$; namely, to decide whether or not $v$ chooses
$W_i$ (as its set of children in $\mathcal{P}_\Sigma$) for the construction of the sample. The main problem is to compute the probability of making this choice. Two factors determine this probability: the choices made thus far, and the probability (at this stage) of producing a random document that satisfies the constraints if we indeed choose $(v_i, W_i)$.

To reflect the choices made thus far, we change the p-document as follows. Let $\tilde{P}_{i-1}$ be the p-document just before processing the pair $(v_i, W_i)$. We obtain the p-document $\tilde{P}_i$ by modifying $\tilde{P}_{i-1}$ as follows. We change the probabilities $\tilde{P}_{i-1}(v_i, W_i)$ and $\tilde{P}_{i-1}(v_j, W_j)$, for each $1 \leq j \leq m$, such that $v_i = v_j$ and $i \neq j$ (i.e., $W_i \neq W_j$). The change depends on whether we choose $(v_i, W_i)$ or not. If we choose $(v_i, W_i)$, then we set $\tilde{P}_i(v_i, W_i)$ to 1 and $\tilde{P}_i(v_j, W_j)$ to 0; otherwise, we set $\tilde{P}_i(v_i, W_i)$ to 0 and $\tilde{P}_i(v_j, W_j)$ to $\tilde{P}_{i-1}(v_j, W_j)/(1 - \tilde{P}_{i-1}(v_i, W_i))$. The subroutine Normalize produces a new p-document by modifying $\tilde{P}_{i-1}$ as described above. In Line 10, it is applied under the assumption that $(v_i, W_i)$ is included in the construction of the sample, whereas the opposite is true in Line 18.

At the beginning of the $i$th iteration, $\tilde{P}_{i-1}(v_i, W_i)$ is the prior probability of choosing $(v_i, W_i)$. However, for randomly deciding whether to include $(v_i, W_i)$ in the generation of the sample, we have to use the posterior probability of choosing $(v_i, W_i)$ given that the sample satisfies the constraints. Let $p_i$ be this posterior probability. Consider the p-document $\tilde{P}'$ that is constructed in Line 10 by applying Normalize($\tilde{P}_{i-1}, v_i \rightarrow W_i$). Note that $\Pr(\tilde{P}' \models \mathcal{C})$ is the posterior probability that the sample satisfies the constraints given that $(v_i, W_i)$ is chosen. Hence, by Bayes’ theorem, $p_i = \tilde{P}_{i-1}(v_i, W_i) \cdot \Pr(\tilde{P}' \models \mathcal{C})/q_{i-1}$, where $q_{i-1} = \Pr(\tilde{P}_{i-1} \models \mathcal{C})$ is the prior probability (at the beginning of the $i$th iteration) that the sample satisfies the constraints.

So, Lines 10–12 calculate the probability $p_i$ of choosing $(v_i, W_i)$. The actual random choice is made in Line 13. If $(v_i, W_i)$ is chosen, then Lines 15–16 set the values of $\tilde{P}_i$ and $q_i$. Otherwise, these values are set in Lines 18–19. Note that in the latter case, we calculate $q_i$ from the values that have already been computed in Lines 11 and 12 where it is assumed that $(v_i, W_i)$ is chosen.

---

3 An alternative is to introduce new constraints instead of changing the p-document, but this is inefficient because it produces a set of constraints that is linear in the size of $\tilde{P}$.
Algorithm Sample($C$)($\tilde{P}$)

1: let $\mathcal{W}(\tilde{P}) = \{(v_1, W_1), \ldots, (v_m, W_m)\}$
2: $\tilde{P}_0 \leftarrow \tilde{P}$
3: $q_0 \leftarrow \Pr(\tilde{P}_0 = C)$
4: for $i = 1$ to $m$ do
5: if $\tilde{P}_{i-1}(v_i, W_i) \in \{0, 1\}$ then
6: $p_i \leftarrow \tilde{P}_{i-1}(v_i, W_i)$
7: $\tilde{P}_i \leftarrow \tilde{P}_{i-1}$
8: $q_i \leftarrow q_{i-1}$
9: continue
10: $\tilde{P}'_i \leftarrow \text{Normalize}(\tilde{P}_{i-1}, v_i \rightarrow W_i)$
11: $q' \leftarrow \Pr(\tilde{P}' = C)$
12: $p_i \leftarrow \tilde{P}_{i-1}(v_i, W_i) \cdot q'/q_{i-1}$
13: toss $b_i \in \{\text{true, false}\}$ with prob. $p_i$ for true
14: if $b_i$ then
15: $\tilde{P}_i \leftarrow \tilde{P}'$
16: $q_i \leftarrow q'$
17: else
18: $\tilde{P}_i \leftarrow \text{Normalize}(\tilde{P}_{i-1}, v_i \not\rightarrow W_i)$
19: $q_i \leftarrow (q_{i-1} - q' \cdot \tilde{P}_{i-1}(v_i, W_i))/(1 - \tilde{P}_{i-1}(v_i, W_i))$
20: return the constant random variable $P_m$

Figure 5.3: Sampling $\tilde{D} = (\tilde{P}, C)$

In Lines 5–9, the loop handles two special cases where the prior probability of choosing the edge $(v_i, W_i)$ is either 0 or 1. This is done not just for the sake of efficiency—it is essential for correctness (see Appendix G).

$\tilde{P}_m$ is the last p-document to be created. For all $i$, the probability $\tilde{P}_m(v_i, W_i)$ is either 0 or 1. Consequently, $\tilde{P}_m$ is a distribution with only one document, namely, $P_m$. This document is returned in Line 20.

The following theorem shows that the algorithm indeed samples $\tilde{D}$. Due to lack of space, the proof is given in Appendix G. By Corollary 5.5.4, Sample($C$) is efficient. Thus, Theorem 5.6.1 immediately follows.
Theorem 5.6.2  For all documents $d$, the probability that $\text{Sample}(\mathcal{C}) (\mathcal{D})$ returns $d$ is equal to $\Pr(\mathcal{D} = d)$.

5.7 Generalizations

We now discuss how the results of the previous sections can be generalized in various ways.

5.7.1 Generalizing to c-Formulae

In order to deal with queries and constraints as they were originally defined, we have to translate them into c-formulae. Obviously, the results of Section 5.5 about efficient solutions to the constraint-satisfaction and query-evaluation problems carry over to queries and constraints that are expressed as arbitrary c-formulae. Furthermore, the sampling algorithm and its proof of correctness are oblivious to the type of constraints (as long as we can efficiently compute the probability that the constraints are satisfied by a random instance). Consequently, our results on sampling generalize straightforwardly to the case of constraints expressed as c-formulae.

5.7.2 Additional Aggregate Functions

Next, we consider additional aggregate functions. Generally, an aggregate function $agg$ maps a set of nodes to a rational number. Below, we naturally generalize the notion of c-formulae to a-formulae (that use arbitrary aggregate functions rather than just $\text{Cnt}$). In Definitions 5.5.1 and 5.5.2 $\text{Cnt}(\sigma_1 \lor \cdots \lor \sigma_k) \otimes N$ is replaced with $agg(\sigma_1 \lor \cdots \lor \sigma_k) \otimes R$, where $R$ is a rational number. In addition, every occurrence of the term “c-formula” is replaced with “a-formula.” We denote by $\text{AF}\{agg_1, \ldots, agg_k\}$ the set of all a-formulae that use aggregate functions from $\{agg_1, \ldots, agg_k\}$.

Naturally, every a-formula of $\text{AF}\{agg_1, \ldots, agg_k\}$ is a constraint. We say that a query $Q = \pi_X \alpha T$ belongs to the family $\text{AF}\{agg_1, \ldots, agg_k\}$ if $T$ is a pattern, $X$ is a projection sequence for $T$, and $\alpha$ maps every node of $T$ to an a-formula of $\text{AF}\{agg_1, \ldots, agg_k\}$. Given a document $d$, the result $Q(d)$ consists of all the tuples that are obtained by applying the
projection to the matches of $\mathcal{M}(\alpha T, d)$.

Next, we define the aggregate functions $\text{Min}$, $\text{Max}$, $\text{Sum}$, $\text{Avg}$ and $\text{Ratio}$. For that, we assume that labels of $\Sigma$ may be rational numbers. The Boolean function $\text{num}(l)$ specifies whether a given label $l$ is numeric. Like $\text{Cnt}$, these five aggregate functions are defined over subsets $U$ of document nodes. We denote by $\text{Num}(U)$ the set $\{v \in U \mid \text{num}(\text{lbl}(v))\}$.

The aggregate functions $\text{Max}$ and $\text{Min}$ are defined as follows. If $\text{Num}(U) \neq \emptyset$, then $\text{Max}(U) = \max\{\text{lbl}(v) \mid v \in \text{Num}(U)\}$ and $\text{Min}(U) = \min\{\text{lbl}(v) \mid v \in \text{Num}(U)\}$. If $\text{Num}(U)$ is empty, then $\text{Max}(U)$ is $-\infty$ and $\text{Min}(U)$ is $\infty$. The functions $\text{Sum}$ and $\text{Avg}$ are naturally defined as follows. $\text{Sum}(U) = \sum_{v \in \text{Num}(U)} \text{lbl}(v)$. If $U$ is nonempty, then $\text{Avg}(U) = \text{Sum}(U)/\text{Cnt}(U)$; otherwise, $\text{Avg}(U)$ is 0.

The ratio function, denoted by $\text{Ratio}$, is a special type of an aggregate function that can be used for representing important constraints of the form “at least 40% of all professors (in each department) have an active grant” (such a constraint can be derived, e.g., from published statistics of the university). Formally, this function gets as input a set $U$ of nodes and a subset $U'$ of $U$, and it returns $|U'|/|U|$ (if $U$ is empty, then the result is 0).

An a-formula that uses $\text{Ratio}$ is built from a disjunction $\sigma_1 \lor \cdots \lor \sigma_k$ and another a-formula $\gamma$, and it has the form $\text{Ratio}(\sigma_1 \lor \cdots \lor \sigma_k, \gamma) \otimes R$. Given a document $d$, the a-formula $\text{Ratio}(\sigma_1 \lor \cdots \lor \sigma_k, \gamma) \otimes R$ is true if $r \otimes R$ is satisfied, where $r$ is the fraction of the nodes $n$ from $\sigma_1(d) \cup \cdots \cup \sigma_k(d)$ that satisfy $d^n_{\Delta} = \gamma$.

The following theorem shows that our results can be extended to constraints that use the $\text{Max}$, $\text{Min}$ and $\text{Ratio}$ functions, in addition to $\text{Cnt}$. The numerical specification includes all the rational numbers that appear in the a-formula.

*Theorem 5.7.1* For all sets of constraints $\mathcal{C}$ and queries $Q$ of $\mathcal{AF}^{\{\text{Cnt}, \text{Max}, \text{Min}, \text{Ratio}\}}$, the three problems $\text{C-Sat}(\mathcal{C})$, $\text{Eval}(Q, \mathcal{C})$ and $\text{Sample}(\mathcal{C})$ can be solved efficiently.

Similar results are unlikely to exist for the aggregate functions $\text{Sum}$ and $\text{Avg}$. Consider, for example, the very simple a-formulae of $\mathcal{AF}^{\{\text{Sum}\}}$ and $\mathcal{AF}^{\{\text{Avg}\}}$ that are defined as follows. The a-formula $\xi_{\Sigma_{\text{all}}}$ verifies whether the sum of all the numerical values in the document equals a specific value, i.e., it is the a-formula $\xi_{\Sigma_{\text{all}}}(\text{Sum}(\star \lor /\lor /\lor /\lor) = R$ (where $R$ is given in the

\[4\text{This is an abuse of notation—we use XPath expressions rather than s-formulae.}\]
numerical specification). Similarly, \( \xi_{\text{all}} \) verifies whether the average numerical value in the document equals the specified \( R \), i.e., it is \( \text{Avg}(\ast \vee \ast/\ast) = R \). It can be shown that deciding each of \( \Pr(\mathcal{P} \models \xi_{\Sigma_{\text{all}}}) > 0 \) and \( \Pr(\mathcal{P} \models \xi_{\text{all}}) > 0 \), given a \( \tilde{\mathcal{P}} \in \Pr\text{XML}^{(\text{ind})} \) and a numerical specification, is NP-complete. The proof is by a reduction from the problem Subset-Sum. Note that a direct corollary is that even multiplicatively approximating \( \Pr(\mathcal{P} \models \xi_{\Sigma_{\text{all}}}) \) and \( \Pr(\mathcal{P} \models \xi_{\text{all}}) \) (see Section 4.8) is infeasible (unless \( \text{NP} \subseteq \text{P} \) or \( \text{NP} \subseteq \text{BPP} \), depending on whether the approximation is deterministic or randomized, respectively).

### 5.7.3 Probabilistic Constraints

We introduce *probabilistic constraints* as a natural method to model restrictions with some level of uncertainty. Intuitively, a probabilistic constraint is simply a standard constraint \( C \), associated with a probability \( p_C \), which indicates the likelihood with which the constraint must be satisfied.

As an example, consider constraint \( C_3 \) of Figure 5.1 and Example 5.3.2 which states that a chair must be a full professor. While this may be a rule in most universities, it is likely to be violated by some. Thus, constraint \( C_3 \) may actually be true only with some probability \( p \) (say, 0.95). As another example, we can derive probabilistic constraints from published statistics. Suppose that we learn from a public statement that in 90% of the universities, there are more than a hundred Ph.D. students. This can naturally be modelled as a probabilistic constraint, namely, “there are more than a hundred Ph.D. students” is applied to the specific university described by the p-document, but there is some probability that it is violated.

We consider the probabilistic space defined by a PXDB that consists of a p-document and a set of probabilistic constraints. There are two natural ways to define this probabilistic space—each by a reduction to a PXDB with deterministic constraints. Under *strict negated compliance* (SNC), each probabilistic constraint \( C \) must be satisfied with probability \( p_C \), and the negation of \( C \) must be satisfied with probability \( 1 - p_C \). Under *weak negated compliance* (WNC), \( C \) must be imposed with probability \( p_C \) (and otherwise, it is disregarded and hence, can either be satisfied or not).
While SNC may be a more intuitive semantics, it could lead to a probability space of PXDBs that is not well-defined. For example, consider the following two probabilistic constraints. First, a full professor has at least one Ph.D. student with probability 0.7. Second, a full professor has at most 15 Ph.D. students with probability 0.9. Under SNC, the probability space is not well-defined, because there is a nonzero probability (i.e., 0.03 = (1−0.7)×(1−0.9)) that a random document must satisfy the negation of both constraints, which is impossible. For WNC, the probability space is well-defined, as long as the conjunction of all constraints is satisfiable (similarly to the case of deterministic constraints). In particular, for the above example, the probability space is well-defined with respect to WNC.

Importantly, we can extend all our results to probabilistic interpretations of constraints, for both WNC and SNC (when well-defined). Other semantics, e.g., that allow for probabilistic hierarchies of constraints, can also be defined and manipulated efficiently. Further details are omitted due to lack of space.
Chapter 6

Summary of All the Results

This section lists the results that were obtained in the course of the Ph.D. research. All of these results were published in scientific journals, conference proceedings and workshops.

6.1 Keyword Proximity Search

In [7], we established our framework for keyword proximity search. Within this framework, we developed (and published in [5, 8, 9, 15]) various algorithms for enumerating Q-fragments in keyword proximity search. All of these algorithms have the following properties. First, they enumerate all of the Q-fragments (depending on the required type, i.e., DQFs, UQFs or SQFs). Second, the enumeration is with polynomial delay. In Chapter 3, algorithms that enumerate by increasing weight and in an approximation thereof were presented. Table 6.1 shows the properties of these (in the third and fifth rows) and the other algorithms. This table is described next.

Each row of Table 6.1 describes an algorithm, where some of them have three versions that correspond to the three types of fragments (see the “Comment” column). For all the algorithms, the input consists of a data graph $G$ and a set $Q$ of keywords. In the second and third columns, the following notation is used: $n$ and $e$ denote the number of nodes and edges of $G$, respectively, and $q$ denotes the number of keywords in $Q$. The number of nodes

\footnote{For some of the algorithms, data complexity is necessarily used.}
<table>
<thead>
<tr>
<th>Type of Algorithm</th>
<th>Init.</th>
<th>ith Delay</th>
<th>Space</th>
<th>Comment</th>
<th>Pubs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unssorted</td>
<td>$O(1)$</td>
<td>$O(eq(n_i + n_{i-1}))$</td>
<td>poly.</td>
<td>DQFs, UQFs and SQFs</td>
<td>8, 15</td>
</tr>
<tr>
<td>2-approx. by inc. height</td>
<td>$O(1)$</td>
<td>$O(n_iq(e + n \log n))$</td>
<td>l.i.poly.</td>
<td>DQFs of general data graphs</td>
<td>5</td>
</tr>
<tr>
<td>$(\theta + 1)$-approx. by inc. weight</td>
<td>$O(1)$</td>
<td>$O(n_i(f + e \log n))$</td>
<td>l.i.poly.</td>
<td>DQFs, UQFs and SQFs</td>
<td>9</td>
</tr>
<tr>
<td>Exact order by inc. weight</td>
<td>$O(en^q)$</td>
<td>$O(e)$</td>
<td>l.i.poly.</td>
<td>DQFs of acyclic data graphs</td>
<td>8, 15</td>
</tr>
<tr>
<td>Exact order by inc. weight</td>
<td>$O(1)$</td>
<td>$O(n_i(4^q n + 3^q e \log n))$</td>
<td>l.i.poly.</td>
<td>DQFs, UQFs and SQFs</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 6.1: Algorithms for enumerating Q-fragments with polynomial delay

in the $i$th printed answer is denoted by $n_i$. Given an approximation algorithm for Steiner trees, $f$ denotes its running time and $\theta$ denotes its approximation ratio (see Section 3.2). For each algorithm, the table specifies the type of the algorithm (i.e., the guaranteed order), the time of the initialization phase, the $i$th delay, the space usage and the corresponding publication in which it is described. Note that “poly.” and “l.i.poly.” stand for polynomial space and linearly incremental polynomial space, respectively, as defined in Section 2.3.

### 6.2 Maximal Answers

In [1], three algorithms for computing full disjunctions were presented. The algorithm PDelayFD is the first algorithm that, under query-and-data complexity, computes full disjunctions with polynomial delay and, in particular, in time that is linear in the size of the output (and polynomial in the size of the input). Another algorithm, NLOJ, is even faster but can only be applied when the full disjunction is equivalent to a left-deep sequence of outerjoins (which is not the general case). The algorithm BiComNLOJ combines the two algorithms by decomposing the schema into its biconnected components, thereby improving the efficiency of PDelayFD for the general case. This improvement is proved analytically and is practically confirmed by an experimental study (which also shows the efficiency BiComNLOJ and its superiority over the state-of-art algorithm [33]).
In [3], we studied the complexity of generating all maximal induced subgraphs for hereditary and connected hereditary properties. Among the results of that paper, our algorithm for full disjunctions was generalized to arbitrary (connected) hereditary properties. Essentially, this result relates the problem of enumerating the maximal induced subgraphs to the restricted version of the problem where the input graph is assumed to satisfy the hereditary property if one node is removed from it. (Recall that if the input graph satisfies the property, then the problem is trivial.) We proved that the general problem can be solved with polynomial delay if the restricted version can be solved in polynomial time (which is the case, e.g., for the hereditary property “is a clique”).

6.3 Ranked Querying

In [10], we investigated the query-and-data complexity of incrementally computing SQL queries with the ORDER BY clause. We stated actual tractability results for SQL queries that correspond to acyclic conjunctive queries, since solving efficiently the non-emptiness problem is a necessary condition for incremental computation with polynomial delay (and it cannot be done for cyclic conjunctive queries). The ORDER BY clause is modeled as an order on homomorphisms from the conjunctive query to the database (since the ORDER BY clause may include attributes that do not appear in the SELECT clause). The main result holds for general conjunctive queries and it states that if one can find the top-ranked homomorphism in polynomial time, then all the tuples of the result can be enumerated in sorted order with polynomial delay (subject to the natural assumption that the given algorithm for finding a maximal homomorphism can be applied to any database that is obtained from the given one by deleting some tuples). For acyclic conjunctive queries, we showed that this property is satisfied by a large class of orders that are defined by means of monotonicity (and include, e.g., a sum of attributes and a lexicographic order).

We also studied the problem of querying XML graphs, a task that typically leads one to face a huge number of answers that have varying degrees of semantic strength. The solution that we presented in [11] incorporates several ideas. First, the nodes and edges of XML
graphs have weights, but the user is able to do some fine tuning by overriding these weights. Second, the weights are used not just for ranking, but also for an a priori elimination of matches that map some pairs of adjacent nodes (from the query) to XML nodes that do not satisfy user-specified distance bounds. Third, for a wide range of ranking functions, the answers can be enumerated in ranked order with polynomial delay, under query-and-data complexity, even if projection is used and duplicates are eliminated.

In [12], we have investigated how to combine incompleteness and ranking in tree queries. The result was a highly general and flexible query language for abstract databases (which include the relational and semistructured models, XML, RDF and more). In particular, answers are allowed to be maximal (rather than complete) while the user can specify edge-completeness constraints which have the form “for the edge $e = (n_1, n_2)$ of the query, if $n_1$ is non-null then $n_2$ is non-null as well” (observe that this generalizes the constraint “node $n$ is non-null”). The user can rank answers according to the degree of incompleteness (by penalizing nodes that are assigned the null value) and combine it with known ranking paradigms commonly in use, e.g., relevance of objects to query terms [49] and (semantic) proximity among objects [2, 55] (by rewarding nodes and edges associated with non-null values). We developed an evaluation algorithm that enumerates answers in ranked order with polynomial delay under query-and-data complexity. The main difficulty was to combine Lawler’s procedure with the notion of maximality which necessitated an algorithm for finding the top-1 answer under four different types of constraints.

### 6.4 Maximal Answers in Uncertain Data

We investigated the problem of maximally joining probabilistic relational data in [14]. In our studied model (which generalizes that of [37]), each tuple is associated with an event that has a given probability. In contrast to full disjunctions, there are two different interpretations of maximal answers: (1) maximal tuples that are produced by partial joins, and (2) maximal tuple sets that are join-consistent and connected. In both cases, the query is meaningless unless the user specifies a threshold that determines the minimum amount of
Table 6.2: The query-and-data complexity of ProbableJCC and ProbablePJoin

<table>
<thead>
<tr>
<th></th>
<th>ProbableJCC</th>
<th>ProbablePJoin</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-rdb</td>
<td>general</td>
<td>general</td>
</tr>
<tr>
<td>general</td>
<td>NPc</td>
<td>P\textsuperscript{#P}h</td>
</tr>
<tr>
<td>localized</td>
<td>NPc</td>
<td>P\textsuperscript{#P}h</td>
</tr>
<tr>
<td>equal-prob.</td>
<td>Poly\textsubscript{=}</td>
<td>P\textsuperscript{#P}h</td>
</tr>
<tr>
<td>bounded-prob.</td>
<td>Poly\textsubscript{=}</td>
<td>P\textsuperscript{#P}h</td>
</tr>
</tbody>
</table>

Table 6.2 summarizes (most of) our results regarding query-and-data complexity of maximally joining probabilistic relations. We use the following notation. \( \mathcal{S}_{\text{clique}} \) is the class of all database schemata that have a clique schema graphs, namely, every two relations share an attribute. The class \( \mathcal{S}_{\text{tree}} \) consists of the schemata with a tree schema graph. Finally, \( \mathcal{S}_\gamma \) is the class of all \( \gamma \)-acyclic database schemata \([13]\) and \( \mathcal{S}_{\gamma}^\neq \) is the restriction of this class to schemata without containments among the sets of attributes of the relations. (Note that \( \mathcal{S}_{\text{tree}} \subseteq \mathcal{S}_\gamma \).) Poly\textsubscript{=} and Poly\textsubscript{=} mean that the problem can be solved in incremental polynomial time and with polynomial delay, respectively. NPc (NP-complete) and P\#P\textsubscript{h} (P\#P\textsubscript{h}-hard under Cook reduction) refer to the decision versions of the two problems. Essentially, for each of the two problems, evaluation in polynomial total time implies that the decision problem can be solved in polynomial time. Each of the second, third and forth rows of Table 6.2 corresponds to a restriction over the database. “Localized” means that every event appears in at most one relation. “Equal-prob.” states that there exists a number \( 0 < p < 1 \), such that the probability of each event \( e \) is either 1 or \( p \). Finally, in “bounded-prob.” there exists a fixed number \( 0 < g < 0.5 \), such that the probability of each event is either 1 or in the interval \([g, 1 - g]\).

In [13], we studied the evaluation of twig patterns over the ProTDB model (i.e., the
family $\mathbf{PrXML}^{\{\text{ind,mux}\}}$ of $p$-documents) under the maximal semantics. We showed that, unlike the above problem, this one is tractable in the general case. In particular, all the maximal matches of a twig in a $p$-document can be enumerated in incremental polynomial time under query-and-data complexity.

### 6.5 Querying Probabilistic XML

We first studied the evaluation of twig queries with projection over the ProTDB model and presented in [13] an evaluation algorithm that is efficient under data complexity. This was the first result showing that projection is generally tractable for an interesting query language and model of probabilistic databases. We also showed that, under query-and-data complexity, evaluating a Boolean query is $\#P$-hard.

In [6], we presented the p-document abstract model and compared concrete families by means of expressive power and query efficiency. We also showed there that $\mathbf{PrXML}^{\{\text{exp}\}}$ is the maximal family in our framework for which twig queries (with projection) can be evaluated efficiently. For the families that are not contained in $\mathbf{PrXML}^{\{\text{exp}\}}$, we showed that all but the trivial Boolean queries are $\#P$-complete, and for those, we presented efficient (additive and multiplicative) approximation algorithms. Finally, we showed that under query-and-data complexity, even approximate evaluation is intractable, and already in the restricted family $\mathbf{PrXML}^{\{\text{mux}\}}$.

The PXDB concept (namely, incorporating constraints in probabilistic XML) was presented and studied in [4]. There, we showed that testing well definedness, querying and sampling are all tractable under data complexity.

Our results on probabilistic XML are described in detail in Chapters [4] and [5].
Chapter 7

Conclusions

Our research is motivated by the belief that the chaotic nature of Web data can be understood by suitable formal (mathematical) models and, moreover, that identifying and analyzing the correct problems in these models can lead to practical tools for effective (e.g., correct and efficient) querying. So, we studied various specific querying paradigms which are aimed at extracting information from data sources that are characterized by being inexact, incomplete and with unknown schemata. Our research was conducted by applying the following uniform methodology. For each of the studied paradigms, we established a formal framework that was used for translating query evaluation in that paradigm into particular, well-defined computational problems. We then analyzed these problems and, in particular, devised efficient algorithmic solutions. When developing algorithms for query evaluation, our focus was not merely on achieving the needed solution for the specific problem at hand, but also on identifying the core algorithmic techniques that are required for solving that problem as well as the challenges that are entailed in the specific adaptation of the techniques. We discovered that only a few general algorithmic techniques constitute key components in solving many of the problems that we studied. However, different problems require different adaptations of these techniques and, in fact, these adaptations were often intricate.

Specifically, in the paradigm of keyword proximity search, we devised the first algorithms that enumerate all the $Q$-fragment with polynomial delay. These algorithms range over
the scale of efficiency vs. desirability of the order of answers, and include enumerations in arbitrary and heuristic order \([8, 15]\), exact and approximate order by weight \([9]\) and approximate order by height \([5]\). The algorithms that were published in \([9]\) were described in detail in Chapter \([9]\). Essentially, these algorithms show that, when weight is used for ranking, keyword proximity search is not harder than finding a Steiner tree (i.e., the top-1 answer or an approximation thereof) in the sense that every Steiner-tree algorithm can be extended to enumerating all the \(Q\)-fragments while having a similar performance (i.e., running time and approximation ratio). The algorithms that enumerate with arbitrary and heuristic order use our technique of partitioning the output space (combined with coroutines) while the rest of the algorithms are based on Lawler’s procedure (which is adapted by different sequences of reductions).

Within the paradigm of maximal answers, we studied in \([11]\) the problem of computing full disjunctions. Our best (optimized) algorithm is based on a core algorithm which, for the sake of enhanced efficiency, is combined with the technique of partitioning the output space (this combination is obtained by decomposing the schema graph into biconnected components). The core algorithm is the first one to compute full disjunctions with polynomial delay and linear time in the size of the output. But the importance of the core algorithm is beyond the full-disjunction operator. We generalized this algorithm into a technique for enumerating maximal induced subgraphs w.r.t. hereditary graph properties. This technique, combined with that of partitioning the output space, was later used for different tasks of finding maximal answers in uncertain data. In particular, by using these techniques as key components, we presented in \([13]\) efficient algorithms for finding maximal matches of twig queries in p-documents of \(\text{PrXML}^{\{\text{ind,mux}\}}\). For maximally joining probabilistic relations, these techniques were used in \([14]\) for efficient evaluation algorithms for important classes of schemata, as well as for proving the hardness of the problem in the general case.

We showed that Lawler’s procedure can be used for developing efficient incremental algorithms for ranked querying in traditional settings of database querying, including acyclic conjunctive queries (with ORDER BY) over relational databases \([10]\) and twig queries over XML graphs \([11]\). Formally, we proved that for rather general ranking functions, answers
can be enumerated in ranked order with polynomial delay under query-and-data complexity. We also showed in [12] that in the case of tree queries (over general databases), Lawler’s procedure can be used for combining maximal answers with ranked querying in the sense that the ranking function is aware of the degree to which the answer is incomplete (in addition to traditional ranking measures). The result was a language that provides the user with natural means of controlling the effect of both missing and inexact matches on the scoring function (in addition to having efficient incremental evaluation).

In the paradigm of querying probabilistic XML, the family $\text{PrXML}^{\{\text{exp, cie}\}}$ of p-documents was shown (in Chapter 4 and in [6]) to be a proper generalization of all other probabilistic XML models in the literature. Right below this family are the families $\text{PrXML}^{\{\text{exp}\}}$ and $\text{PrXML}^{\{\text{cie}\}}$ [18, 98]. For $\text{PrXML}^{\{\text{exp, cie}\}}$, efficient approximate evaluations of twig queries with projection were presented. These evaluations are randomized algorithms, the approximation is either multiplicative or additive and their efficiency is in terms of data complexity. As for exact evaluation, we proved a dichotomy theorem showing that every Boolean query is $\#P$-complete (even in $\text{PrXML}^{\{\text{cie}\}}$), unless it is uninterestingly trivial. In $\text{PrXML}^{\{\text{exp}\}}$, however, there is an efficient exact evaluation of queries and, moreover, of c-formulae which significantly extend Boolean twigs by nesting patterns in count operators and vice versa. One can express natural constraints by c-formulae, which leads to the concept of PXDBs (studied in Chapter 5 and in [1]). There, probabilistic dependencies among nodes of a $\text{PrXML}^{\{\text{exp}\}}$ document are described by a fixed set of constraints rather than many specific relationships among distributional nodes as done in $\text{PrXML}^{\{\text{cie}\}}$. Although highly complicated dependencies can be represented in a PXDB, one can efficiently evaluate twig queries with projection and the aggregate functions count, ratio, min and max (and, moreover, test whether a given PXDB is well defined and sample it). For comparison, $\text{PrXML}^{\{\text{cie}\}}$ can describe cases that cannot be handled by PXDBs, but in the former, even queries with negation (which is a restricted form of count) do not admit an efficient multiplicative approximation (let alone exact evaluation).

The ultimate goal underlying this dissertation (and the subject of future work) is to transform the proposed theoretical frameworks and analytical solutions into practical tools.
The challenges that are involved in achieving this goal are of a practical nature (e.g., software engineering, optimization, etc.) in some cases, and more foundational (possibly, requiring further theoretical tools) in others. These challenges are briefly discussed below.

Based on the work of [12] that combines maximal answers and ranked querying, we plan to develop a highly flexible querying system that will enable the user to interactively explore complicated data sources while querying. We expect techniques from our earlier work on interconnection semantics for XML [2] to be very useful in such a system. An important yet open question is how such a system can employ the paradigm of combining maximal answers and uncertainty [13, 14] and, at the same time, preserving the high degree of flexibility.

Our algorithms for keyword proximity search can be used as key components in a system for keyword search over databases. However, as we show in [5], there are still a few challenges that need to be addressed on the way to obtaining a such a system, especially when the task is to search in complex data graphs (like the Mondial database). One problem is that of repeated information, namely, the user might be overwhelmed with different answers that provide similar information (for example, many answers with the same schema or many answers that are obtained by combining a few smaller answers). Various alternatives for solving this problem were presented in [5], and an experimental study (described in the thesis of Konstantin Golenberg) showed that some of them are indeed effective. We are currently continuing the study of keyword proximity search by developing an implemented system.

Our results on probabilistic XML show that it is a highly tractable and rich paradigm. We believe that a well-developed system for querying probabilistic XML can be used for solving complicated, real-life problems involving uncertain data. As part of the master’s thesis of Yuri Koscharovsky, an experimental study (published in [6]) showed the practicality of our algorithmic approach (combined with elementary optimization techniques, e.g., memoing [106]) for query evaluation over probabilistic XML. Recall that the condition for the applicability of PXDBs to a specific scenario is the ability to represent all the probabilistic dependencies by a small set of constraints. In such scenarios, PXDBs are much more
succinct than other data models that support probabilistic dependencies (e.g., the XML models of [18,98] or Bayesian networks [34,88]). Moreover, they enable highly expressive queries (e.g., queries with aggregate operators) to be evaluated efficiently.

In conclusion, we believe that the theory that has been developed in this dissertation can lead to practical tools that satisfy the information needs of Web users by effectively extracting the relevant data therein.
List of Publications


### Additional References


S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava, and D. Toman. Structure and


Querying Paradigms for the Web

Appendix

Benny Kimelfeld

Submitted to the Senate of The Hebrew University
August, 2008
Querying Paradigms for the Web

Appendix

Benny Kimelfeld

Submitted to the Senate of The Hebrew University
August, 2008
# Table of Contents

A Finding Steiner Trees

A.1 Steiner Trees .............................................. 1

A.2 Steiner-Tree Approximations .............................. 2

A.3 Computing Steiner Trees ................................. 3

B Approximating the Minimal Constrained DQF

B.1 The Algorithm ........................................... 10

B.2 Correctness and Efficiency ............................... 12

C Engines for Keyword Proximity Search in the Literature 16

D Translations Between Families of p-Documents

D.1 Most General Families ................................. 19

D.2 Expressive Power of exp Nodes ....................... 21

D.3 Expressive Power of cie Nodes ....................... 23

E Proofs of Hardness Results from Chapter 4

E.1 Proof of Theorem 4.7.1 ................................. 25

E.2 Proof of Theorem 4.8.3 ................................. 27
F Evaluating c-Formulae over p-Documents

F.1 The Main Steps .................................................. 29
F.2 Constructing the Set Γ ............................................. 31
F.3 First Case: Ordinary Node with at Most One Child .......... 33
F.4 Transformations ..................................................... 36
F.5 Second Case: Ordinary Node with Multiple Children ........ 41
F.6 Third Case: Distributional Node .................................. 43
F.7 The Final Computation and Total Execution Time ............ 43

G Sampling PXDBs: Proof of Correctness

G.1 Notation and Conventions ......................................... 45
G.2 Analysis of a Specific Execution ................................. 46
Appendix A

Finding Steiner Trees

In this chapter, we consider the Steiner-tree optimization problem. We first describe the popular variants of this problem, namely, the directed, undirected and group Steiner-tree problem. We then review the literature on approximate solutions for this NP-hard problem. Finally, we present exact solutions, which are efficient under fixed-parameter complexity where the parameter is the number of terminals (or node sets in the case of group variant).

A.1 Steiner Trees

Let $G$ be a directed and weighted graph and $U$ be a set of $k$ nodes of $G$. A directed Steiner tree of $U$ in $G$ is a d-subtree of $G$ that contains $U$ and has a minimal weight; that is, the weight of every d-subtree that contains $U$ is at least as large. Similarly, an undirected Steiner tree of $U$ in $G$ is a minimal-weight subtree among the u-subtrees of $G$ that contain $U$. The third type of Steiner trees is defined as follows. For subsets $U_1, \ldots, U_k$ of the nodes of $G$, a group Steiner tree of $U_1, \ldots, U_k$ in $G$ is a minimal-weight subtree among the u-subtrees $T$ of $G$ with the following property: $T$ contains at least one node from each $U_i$.

We consider the problem of finding Steiner trees. Thus, there are three optimization problems that correspond to the three types of Steiner trees. Under each variant, however, it is intractable to find Steiner trees. In particular, given a graph $G$, a set $U$ of $k$ nodes of $G$ (or $k$ sets of nodes of $G$) and an integer $j$, the following decision problem is NP-complete.
Determine whether \( G \) contains a Steiner tree that has at most \( j \) nodes. In the next section, we review some of the approximation results on these problems. The following sections show that, by using \( k \) as the parameter, all three problems are fixed-parameter tractable.

### A.2 Steiner-Tree Approximations

The vast literature on Steiner-tree approximation offers a wide range of algorithms with approximation-ratio vs. efficiency tradeoff. In this section, we survey some of the existing approximations.

The undirected Steiner-tree problem can be approximated within a constant ratio. For example, one can achieve a 2-approximation in \( \Theta(n \log n + m) \) time \[74] and an \( \frac{11}{6} \)-approximation in \( \Theta(n^3) \) time \[112\]. The best known approximation ratio is \( (1 + \frac{1}{\log_2 j})(1 + \frac{\ln 3}{2}) \) in \( \Theta(n^j) \) time for any fixed integer \( j > 1 \) \[95, 96\]. Thus, we can get arbitrarily close to the approximation ratio \( 1 + \frac{\ln 3}{2} \approx 1.55 \) in polynomial time. On the other hand, it was shown \[23, 29\] that this problem cannot be approximated within a constant ratio that is arbitrarily close to 1, unless \( P=NP \) (in particular, \( \frac{96}{95} \) is a known lower bound \[29\]).

For the directed and the group Steiner-tree problem, a \( j(j-1)k^{\frac{1}{7}} \)-approximation can be achieved in \( \Theta(n^j k^{2j}) \) time for any fixed integer \( j > 1 \) \[28\]. Thus, for any fixed \( \epsilon > 0 \), an \( \Theta(k^\epsilon) \)-approximation can be obtained in polynomial time. A randomized \( \Theta(\log^3 n \log k) \)-approximation is proposed in \[54\]. For the case where \( G \) is an undirected tree, they propose a randomized algorithm that achieves an \( \Theta(\log k \log N) \) approximation ratio, where \( N \) is the maximal size of a set \( U_i \). Note that this result is particularly important for the cases where the data graph is obtained from an XML tree and either DQFs or SQFs are enumerated. In \[60\], it is shown that it is unlikely to obtain, in polynomial time, an approximation ratio that is better than a polylogarithmic function of \( k \). Therefore, the undirected Steiner-tree problem is inherently easier than either the directed or the group version.
A.3 Computing Steiner Trees

In this section, we show that the Steiner-tree problems we consider are fixed-parameter tractable (with the parameter $k$) by presenting algorithms for computing them. We first present an algorithm for directed Steiner trees and then show how to adapt it for finding undirected and group Steiner trees.

A.3.1 Directed Steiner Trees

Dreyfus and Wagner [42] presented an algorithm for finding undirected Steiner trees. Our algorithm is basically an adaptation of their algorithm to the directed case. In addition, we improve the running time by avoiding the computation of all shortest paths between pairs of nodes. The solution of Feldman and Ruhl [45] for a generalized problem is in polynomial time under the assumption that $k$ is fixed. But in their algorithm, $k$ is used as the exponent of the polynomial. In contrast, we present below an FPT algorithm with the parameter $k$ (Section 2.1 defines the relevant complexity measures).

The input of the algorithm $\text{DSteiner}$ of Figure A.1 consists of a graph $G$ and a set $U$ of nodes in $G$. This algorithm returns an array $M$ of weights. An index of $M$ consists of a node $v$ of $G$ and a subset $W$ of $U$. The entry $M[v, W]$ stores the weight of the minimal d-subtree $T$, such that (1) $T$ contains all nodes of $W$ and (2) $T$ is rooted at $v$. If no such subtree $T$ exists, then $M[v, W]$ is $\infty$. Throughout this section, we use $\text{MinWeight}(v, W)$ to denote this minimal weight. In particular, the weight of a directed Steiner tree of $U$ in $G$ is the minimal value $\text{MinWeight}(v, U)$ among all nodes $u$. As we later show, the array $M$ can easily be used for constructing a directed Steiner tree of $U$.

The array $M$ is initialized in Line 1–6 as follows. $M[v, W] = 0$ if either $W = \emptyset$ or $W = \{v\}$ (hence $v \in U$); otherwise, $M[v, W] = \infty$. Lines 8–16 are executed for every subset $W$ of $U$. During this part of the computation, the correct value of $M[v, W]$ is determined for all nodes $v$. We use a priority queue $Q_W$ that stores nodes of $G$. The top element in this queue is a node $v$ for which $M[v, W]$ is minimal. Lines 9–16 consist of two main parts. In Lines 9–11, an initial value $M[v, W]$ is determined for all nodes $v$ of $G$ and $v$
Algorithm DSteiner($G, U$)

1: for all nodes $v \in V(G)$ do
2:   for all subsets $W \subseteq U$ do
3:     $M[v, W] := \infty$
4:     $M[v, \emptyset] := 0$
5: for all nodes $u \in U$ do
6:   $M[u, \{u\}] := 0$
7: for all $W \subseteq U$ in increasing size do
   // — determine $M[v, W]$ for all $v \in V(G)$ — //
8:   $Q_W \leftarrow$ an empty queue of nodes $v$ with priority $M[v, W]$
9:   for all nodes $v \in V(G)$ do
10:     $M[v, W] := \min \{ M[v, Y] + M[v, W \setminus Y] \mid Y \subseteq W \}$
11:   $Q_W$.insert($v$)
12: while $Q_W \neq \emptyset$ do
13:     $v_m := Q_W$.remove-top()
14:     for all edges $e = (v, v_m) \in E(G)$ do
15:       if $M[v, W] > M[v_m, W] + w_G(e)$ then
16:         $M[v, W] := M[v_m, W] + w_G(e)$
17: return $M$

Figure A.1: Finding a directed Steiner tree of $U$ in $G$

is inserted into $Q_W$. The initial value is the minimal value among $M[v, Y] + M[v, W \setminus Y]$, where $Y$ is some subset of $W$. Note that the position in $Q_W$ is determined by the initial value. In Lines 12–16, every node $v_m$ in $Q_W$ is removed. When $v_m$ is removed, Line 14 iterates over all the incoming edges of $v_m$. For each such an edge $e = (v, v_m)$, the entry $M[v, W]$ is set to be the minimal between its current value and $M[v_m, W] + w_G(e)$. The algorithm terminates by returning the array $M$ in Line 17. Next, we prove the correctness of DSteiner($G, U$).

The proof of the following proposition is omitted, since it is straightforward by an induction on the time in which $M[v, W]$ is updated.

**Proposition A.3.1** During the execution of DSteiner($G, U$), the value of $M[v, W]$ is not increasing. Furthermore, $M[v, W] \geq \text{MinWeight}(v, W)$ always holds.
Let $W$ be a subset of $U$ and $v$ be a node of $G$. We denote by $\mathcal{M}_r[v, W]$ the value of $\mathcal{M}[v, W]$ when $v$ is removed from $\mathcal{D}_W$ in Line 13 of DSteiner. The following lemma shows that in Lines 12–16, the elements of $\mathcal{D}_W$ are removed in a non-decreasing order of $\mathcal{M}_r[v, W]$.

**Lemma A.3.2** Consider an execution of DSteiner$(G, U)$. Let $W$ be a subset of $U$ of $v_1$ and $v_2$ be two nodes of $G$. If $\mathcal{M}_r[v_1, W] < \mathcal{M}_r[v_2, W]$, then $v_1$ is removed from $\mathcal{D}_W$ before $v_2$.

**Proof.** Let $W$ be given and suppose, by way of contradiction, that the lemma does not hold for $W$, that is, the nodes of $V$ are not removed from $\mathcal{D}_W$ in a non-decreasing order of $W$. So there are necessarily a number $i$ and two nodes $v_i$ and $v_{i+1}$, such that (1) $\mathcal{M}_r[v_i, W] > \mathcal{M}_r[i + 1, W]$, and (2) $v_i$ and $v_{i+1}$ are removed from $\mathcal{D}_W$ in the $i$th and $(i + 1)$st iterations, respectively, of Lines 13–16. Since $v_i$ is chosen in the $i$th iteration, the priority of $\mathcal{D}$ implies that, at that time, $\mathcal{M}_r[v_i, W] \leq \mathcal{M}_r[v_{i+1}, W]$ holds. But this means that $\mathcal{M}[v_{i+1}, W]$ changes before the next iteration of Lines 13–16. This change must be in Line 16 of the $i$th iteration. However, the new value must be of the form $\mathcal{M}[v_i, W] + w_G(e) \geq \mathcal{M}_r[v_i, W]$. Thus, we get a contradiction. □

Correctness of the algorithm follows from the next lemma.

**Lemma A.3.3** Consider an execution of DSteiner$(G, U)$. For all $W \subseteq U$ and $v \in \mathcal{V}(G)$, $\mathcal{M}_r[v, W] = \text{MinWeight}(v, W)$ holds.

**Proof.** If $G$ does not contain any d-subtree $T$, such that $W \subseteq \mathcal{V}(T)$ and $\text{root}(T) = v$, then $\text{MinWeight}(v, W) = \infty$ and correctness of the lemma follows directly from Lines 1–3 and Proposition A.3.1. So we prove the lemma for $v$ and $W$, such that $\text{MinWeight}(v, W) < \infty$. We denote by $\text{MinSteiner}(v, W)$ the minimal natural number $N$, such that $G$ contains a d-subtree $T$ with the following properties: (1) $T$ contains $W$, (2) $T$ is rooted at $v$, (3) $w(T) = \text{MinWeight}(v, W)$, and (4) $T$ has exactly $N$ nodes. We also denote by $\Delta^v_W$ an arbitrary $T$ that satisfies these four properties. The proof of the lemma is by induction, first on the size of $W$ and then on $\text{MinSteiner}(v, W)$.

The basis of the induction is the case $\text{MinSteiner}(v, W) = 1$. Note that $|W| = 0$ is included in this case. Then $W$ is either $\emptyset$ or $\{v\}$. Thus, correctness of the lemma for this
case is due to the initialization of Lines 1–6 and Proposition A.3.1. For the inductive step, suppose that MinSteiner\((v,W)\) > 1, i.e., \(\Delta^v_W\) has at least one edge. We consider three cases.

**Case 1:** \(v \in W\). If \(W = \{v\}\), then MinSteiner\((v,W)\) = 1 and this case is handled above at the basis of the induction. So suppose that \(W\) contains at least two nodes. Let \(W' = W \setminus \{w\}\). Observe that MinWeight\((v,W)\) = MinWeight\((v,W')\). Consider the time when \(M[v,W]\) is set a value in Line 10. By the order of the traversal of Line 7, both \(W'\) and \(\{v\}\) have already been considered in previous iterations. Moreover, by the induction hypothesis, \(M_r[v,W'] = \text{MinWeight}(v,W')\) and \(M_r[v,\{v\}] = 0\). Thus by Proposition A.3.1 and the definition of the set in the right-hand side of Line 10, we conclude that after that execution of Line 10, \(M[v,W] \leq M_r[v,W'] + M_r[v,\{v\}] = \text{MinWeight}(v,W)\) holds. Observe that \(Q_W\) is constructed only after that execution of Line 10, hence Proposition A.3.1 implies that \(M_r[v,W] = \text{MinWeight}(v,W)\), as claimed.

**Case 2:** \(v \notin W\) and \(\Delta^v_W\) is reduced w.r.t. \(W\). In this case, \(v\) has at least two children in \(\Delta^v_W\). Then it can easily be shown that \(\Delta^v_W\) has two d-subtrees \(T_1\) and \(T_2\) of \(\Delta^v_W\) that satisfy the following. First, \(T_1 \cup T_2 = \Delta^v_W\). Second, \(v\) is the root of both \(T_1\) and \(T_2\) and it is the only node that is in both subtrees. Third, \(T_1\) and \(T_2\) contain nonempty subsets \(W_1\) and \(W_2\) of \(W\) (and possibly other nodes of \(W\)), respectively, such that \(W_1 = W \setminus W_2\). Clearly, MinWeight\((v,W)\) \(\leq \text{MinWeight}(v,W_1) + \text{MinWeight}(v,W_2)\). It follows from the order of the iteration of Line 7, the induction hypothesis and Proposition A.3.1 that both \(M[v,W_1] = \text{MinWeight}(v,W_1)\) and \(M[v,W_2] = \text{MinWeight}(v,W_2)\) hold when \(M[v,W]\) is first considered. We conclude that, after Line 10 is executed for \(W\) and \(v\), \(M[v,W] \leq M[v,W_1] + M[v,W_2]\) holds. Thus, \(M[v,W] = \text{MinWeight}(v,W)\) follows from Proposition A.3.1 as claimed.

**Case 3:** \(v \notin W\) and \(\Delta^v_W\) is not reduced w.r.t. \(W\). Since \(\Delta^v_W\) is minimal, \(v\) has exactly one child in \(\Delta^v_W\) (otherwise, a redundant leaf can be removed). Let \(u\) be that child and \(T_u\) be the subtree of \(\Delta^v_W\) that is obtained by removing \(v\) from \(\Delta^v_W\). Since \(\Delta^v_W\) is minimal, \(T_u\) must have the weight \(\text{MinWeight}(u,W)\). Thus, \(\text{MinWeight}(v,W) = w_G(e) + \text{MinWeight}(u,W)\), where \(e\) is the edge \((v,u)\). Moreover, MinSteiner\((u,W)\) \(\leq |V(T_u)| < \text{MinSteiner}(v,W)\). By the
induction hypothesis, $\mathcal{M}_r[u, W] = \text{MinWeight}(u, W)$ holds. If $u$ is removed from $\mathcal{Q}_W$ before $v$, then the execution of Lines 14–16 that follows the removal of $u$ implies that $\mathcal{M}[v, W] \leq \mathcal{M}_r[v, W] + w_G(e)$ holds. By Proposition [A.3.1], $\mathcal{M}[v, W] = \text{MinWeight}(v, W)$ when $v$ is removed from $\mathcal{Q}_W$, as claimed. Otherwise, if $u$ is removed after $v$, then Lemma [A.3.2] implies that $\mathcal{M}_r[v, W] \leq \mathcal{M}_r[u, W]$. In this case, the edge $e$ has a zero weight and $\mathcal{M}_r[u, W] = \text{MinWeight}(v, W)$. Again, the claim follows from Proposition [A.3.1].

Next, we analyze the running time of $\text{DSteiner}(G, U)$. We use $n$ and $e$ to denote the number of nodes and edges of $G$, respectively. We assume that $G$ is a connected graph (otherwise, we consider only the connected component that contains $U$). Therefore, $e \geq n - 1$. We use $k$ to denote the number of nodes in $U$.

**Lemma A.3.4** $\text{DSteiner}(G, U)$ terminates in $O\left(3^k n + 2^k e \log n\right)$ time.

**Proof.** We first analyze the execution cost of Lines 8–16 for a set $W$ of size $j$. The execution cost of Line 10 is $O(2^j)$. The execution cost of Line 11 is $O(\log n)$. Therefore, Lines 9–11 cost $O(2^j n + n \log n)$ time. In Lines 12–16 we remove $n$ nodes from $\mathcal{Q}_W$ and update $\mathcal{Q}_W$ (in Line 16) $e$ times. Therefore, the execution cost of Lines 12–16 is $O(e \log n)$ and that of Lines 8–16 is $O(2^j n + e \log n)$.

Clearly, the running time is dominated by the execution cost of Lines 7–16. Let $f(G, U)$ denote this cost. Then,

$$f(G, U) = O\left(\sum_{j=0}^{k} \binom{k}{j} (2^j n + e \log n)\right) =$$

$$= O\left(n \sum_{j=0}^{k} \binom{k}{j} 2^j + e \log n \sum_{j=0}^{k} \binom{k}{j}\right) = O\left(3^k n + 2^k e \log n\right),$$

as claimed. □

In [39] it is observed that by implementing $\mathcal{Q}_W$ as a Fibonacci heap [48], we can improve the running time. In particular, the amortized execution cost of the operations **insert** and **decrease-key** (Lines 11 and 16, respectively) is constant while that of **remove-top**

\footnote{Note that [39] was published after our paper [9].}
is logarithmic. By modifying the above proof, the running time becomes

\[ O \left( 3^k n + 2^k n \log n + 2^k e \right). \]

Next, we describe how we can obtain from \( \mathcal{M} \) a minimal subtree \( \Delta^v_W \) of \( G \) that contains a set \( W \subseteq U \) and is rooted at a node \( v \). For each outgoing edge \( e = (v, u) \), we test whether \( \mathcal{M}[v, W] = w_G(e) + \mathcal{M}[u, W] \). If so, we recursively construct the tree \( \Delta^u_W \) and add the edge \( e \). Otherwise, we find a subset \( \emptyset \neq Y \subset W \), such that \( \mathcal{M}[v, W] = \mathcal{M}[v, Y] + \mathcal{M}[v, W \setminus Y] \). We then recursively construct \( \Delta^v_Y \) and \( \Delta^v_{W \setminus Y} \) and concatenate the two subtrees. Finally, to obtain a directed Steiner tree of \( U \) in \( G \), we choose the node \( v \) for which \( \mathcal{M}[v, W] \) is minimal and construct the subtree \( \Delta^v_U \). Note that zero-weight edges may lead to a case where the above process results in a graph rather than a tree. In this case, we need to remove redundant nodes and edges from the result (e.g., by removing redundant leaves from the tree that is obtained by a DFS traversal from \( v \)). An alternative approach to the above tree construction is to store, during the execution of \( \text{DSteiner} \), information about the decisions made when updating \( \mathcal{M} \).

We conclude this section with the following theorem which is a direct corollary of Proposition \[\text{A.3.1}\] Lemma \[\text{A.3.3}\] and Lemma \[\text{A.3.4}\].

**Theorem A.3.5** Let \( G \) be a graph and \( U \) be a set of \( k \) nodes of \( G \). A directed Steiner tree of \( U \) in \( G \) can be found in \( O \left( 3^k n + 2^k n \log n + 2^k e \right) \) time.

### A.3.2 Undirected and Group Steiner Trees

Using the algorithm \( \text{DSteiner} \), we can find undirected and group Steiner trees by reducing these two problems to that of finding directed Steiner trees. For both problems, we add a back edge \( (u, v) \) for every edge \( (v, u) \) in the graph (the weight of \( (u, v) \) is that of \( (v, u) \)). Then, in order to find an undirected Steiner tree of \( U \), we find a directed Steiner tree of \( U \) in the new graph. For finding a group Steiner tree, we also add a node \( u_i \) for every set \( U_i \) and a zero-weight edge from every \( v \in U_i \) to the node \( u_i \). We then find a directed Steiner tree of \( \{u_1, \ldots, u_k\} \) in the new graph. It can also be shown that the algorithm \( \text{DSteiner} \)
can be adapted to find undirected and group Steiner trees without adding new edges and nodes. This way, Steiner trees are found more efficiently than the above reductions. The following theorem summarizes the above.

**Theorem A.3.6** Let $G$ be a graph, $U$ be a set of $k$ nodes of $G$ and $U_1, \ldots, U_k$ be $k$ sets of nodes of $G$. The following can be found in $O(3^k n + 2^k n \log n + 2^k e)$ time.

- An undirected Steiner tree of $U$ in $G$.
- A group Steiner tree of $U_1, \ldots, U_k$ in $G$. 
Appendix B

Approximating the Minimal Constrained DQF

In this chapter, we describe and prove the correctness of the algorithm Approx-Fragment that finds an approximation of a minimal DQF that satisfies the given set of PF constraints. This algorithm runs in polynomial time under query-and-data complexity, i.e., without assuming that the query is of a fixed size.

Note that if, in the algorithm Min-Fragment of Figure 3.7, we simply replaced Min-Supertree with the algorithm Approx-Supertree of Section 3.5.3, the running time would still be exponential in the number of PFs of \( \mathcal{U} \).

B.1 The Algorithm

The algorithm Approx-Fragment of Figure B.1 accepts as input the data graph \( G \) and a set of PF constraints \( \mathcal{V} \), such that leaves(\( \mathcal{V} \)) = \( Q \). This algorithm finds a \((\theta + 1)\)-approximation of a minimal DQF that satisfies \( \mathcal{V} \), given that Approx-DSteiner finds a \( \theta \)-approximation of a directed Steiner tree. By Observation 3.4.2, we can assume that \( \mathcal{V} \) has at most two NPFs. This fact is important in showing that Approx-Fragment has a polynomial runtime even if the size of the query \( Q \) is unbounded.

Approx-Fragment starts by checking whether \( \mathcal{V} \) has no RPFs and, if so, Min-Fragment
Algorithm Approx-Fragment\((G, \mathcal{U})\)

1: if \(\mathcal{U}_{rpf} = \emptyset\) then
2: \hspace{1em} return Min-Fragment\((G, \mathcal{U})\)
3: \hspace{1em} // — Construct T\(_{app}\) — //
4: \hspace{1em} T\(_{app}\) \leftarrow\text{Approx-Supertree}\((G, \mathcal{U})\)
5: \hspace{1em} if T\(_{app}\) = \bot or T\(_{app}\) is a DQF then
6: \hspace{1em} \hspace{1em} return T\(_{app}\)
7: \hspace{1em} \hspace{1em} // — Construct T\(_{min}\) — //
8: \hspace{1em} \hspace{1em} G' \leftarrow G
9: \hspace{1em} \hspace{1em} for all \(T \in \mathcal{U}\) and nodes \(u \in \mathcal{V}(T) \setminus \{\text{root}(T)\}\) do
10: \hspace{1em} \hspace{1em} remove from \(G'\) all edges \((v, u)\), such that \((v, u) \notin \mathcal{E}(T)\)
11: \hspace{1em} \hspace{1em} T\(_{min}\) \leftarrow \bot // — note that the weight of \(\bot\) is \(\infty\) — //
12: \hspace{1em} \hspace{1em} for all \(R \in \mathcal{U}_{rpf}\) do
13: \hspace{1em} \hspace{1em} \hspace{1em} T \leftarrow\text{Min-Fragment}\((G', \mathcal{U}_{npf} \cup \{R\})\)
14: \hspace{1em} \hspace{1em} \hspace{1em} T\(_{min}\) \leftarrow \min\{T\(_{min}\), T\}
15: \hspace{1em} \hspace{1em} if T\(_{min}\) = \bot then
16: \hspace{1em} \hspace{1em} \hspace{1em} return \bot
17: \hspace{1em} \hspace{1em} // — Determine the root of the result — //
18: \hspace{1em} \hspace{1em} r \leftarrow \text{root}(T\(_{min}\))
19: \hspace{1em} \hspace{1em} if \(r\) belongs to a subtree \(R \in \mathcal{U}_{rpf}\) then
20: \hspace{1em} \hspace{1em} \hspace{1em} r \leftarrow \text{root}(R)
21: \hspace{1em} \hspace{1em} // — Remove redundant nodes and edges from T\(_{min}\) ∪ T\(_{app}\) — //
22: \hspace{1em} \hspace{1em} G\(_{app}\) \leftarrow T\(_{min}\) ∪ T\(_{app}\)
23: \hspace{1em} \hspace{1em} remove from G\(_{app}\) all incoming edges of \(r\)
24: \hspace{1em} \hspace{1em} for all nodes \(v \in \mathcal{V}(G\(_{app}\))\) with two incoming edges do
25: \hspace{1em} \hspace{1em} \hspace{1em} remove from G\(_{app}\) the incoming edge of \(v\) that belongs to T\(_{app}\)
26: \hspace{1em} \hspace{1em} \hspace{1em} G\(_{app}\) \leftarrow G\(_{app}\) \setminus \{v \in \mathcal{S}(G\(_{app}\)) \mid \text{no } k \in Q \text{ is reachable from } v \text{ in } G\(_{app}\})\}
27: \hspace{1em} \hspace{1em} return G\(_{app}\)

Figure B.1: Finding an approximation of the minimal DQF under a set of PF constraints is applied to the input. Otherwise, a G-supertree T\(_{app}\) that satisfies \(\mathcal{U}\) is constructed by the algorithm Approx-Supertree. If T\(_{app}\) is either a DQF or \(\bot\), then it is returned by the algorithm. If not, the root of T\(_{app}\) has only one child (and, hence, root(T\(_{app}\)) must also be the root of some NPF of \(\mathcal{U}\), since T\(_{app}\) is reduced w.r.t. \(\mathcal{U}\)). We fix this problem by
modifying $T_{\text{app}}$ at the cost of increasing the approximation ratio to $\theta + 1$. This is done as follows.

In the loop of Line 7, $G'$ is obtained from $G$ by deleting all edges $e$, such that $e$ is not in $\mathcal{U}$ and $e$ enters a non-root node of some PF of $\mathcal{U}$ (note that the deleted edges cannot be included in any subtree of $G$ that satisfies $\mathcal{U}$). The loop of Line 10 iterates over all RPFs $R \in \mathcal{U}$ and applies Min-Fragment to $G'$ and $\mathcal{U}_{\text{npf}} \cup \{R\}$. Later, we show the following. The minimal subtree $T_{\text{min}}$ among all those returned by Min-Fragment, during the iterations of the loop, has a root with at least two children and a weight that is not larger than that of the minimal DQF $F$ that satisfies $\mathcal{U}$.

The resulting DQF is obtained from the subgraph $T_{\text{app}} \cup T_{\text{min}}$. Clearly, the weight of $T_{\text{app}} \cup T_{\text{min}}$ is at most $\theta + 1$ times the weight of $F$. But $T_{\text{app}} \cup T_{\text{min}}$ is not necessarily a tree (it might even contain directed cycles). However, we later show that $T_{\text{app}} \cup T_{\text{min}}$ contains a DQF that satisfies $\mathcal{U}$. Specifically, such a DQF $F$ is obtained in Lines 15–22, as follows.

Recall that the root of $T_{\text{app}}$ is reachable from the root of $T_{\text{min}}$, since root($T_{\text{app}}$) is also the root of some NPF of $\mathcal{U}$ (or else the algorithm terminates in Line 5). Therefore, root($T_{\text{min}}$) becomes the root of $F$ unless root($T_{\text{min}}$) is in some RPF $R \in \mathcal{U}$ and if so, root($R$) becomes the root of $F$. Lines 19–22 remove all redundant nodes and edges of $G_{\text{app}}$. The loop of Lines 20–21 iterates over all nodes $v$ of $G_{\text{app}}$ that have more than one incoming edge. Note that such a node $v$ has exactly two incoming edges: one belongs to $T_{\text{app}}$ and the other is from $T_{\text{min}}$. We remove the first and leave the second. Line 22 deletes all structural nodes that have no descendant keywords.

B.2 Correctness and Efficiency

In this section, we prove the correctness of the algorithm Approx-Fragment and analyze its running time. As in the previous section, $F^m$ denotes an arbitrary DQF of a minimal weight. Note that if the algorithm terminates in Line 2 or 5, then the correctness is trivial.

Lemma B.2.1 Suppose that $G$ contains at least one DQF that satisfies $\mathcal{U}$ and consider an execution of Approx-Fragment($G, \mathcal{U}$). If Line 6 is reached, then in Line 13 $T_{\text{min}}$ is an
RPF and \( w(T_{\text{min}}) \leq w(F^m) \).

Proof. We claim that there is an RPF \( R \in \mathcal{U} \) and a subtree \( T \) of \( F^m \), such that \( T \) is a reduced supertree of \( \mathcal{U}_{\text{npf}} \cup \{R\} \) and its root has at least two children. This claim implies that in Line 13, \( T_{\text{min}} \) is an RPF and the weight of \( T_{\text{min}} \) is not larger than that of \( F^m \). To prove the claim, consider a minimal subtree \( T' \) of \( F^m \) that has the same root as \( F^m \) and includes all the NPFs of \( \mathcal{U} \). We choose the RPF \( R \) as follows. If \( \text{root}(F^m) \) has more than one child in \( T' \), then \( R \) can be any RPF of \( \mathcal{U} \). Otherwise, let \( v \) be the single child of the root in \( T' \). Since \( F^m \) is a DQF, the root of at least one RPF is not a descendant of \( v \) in \( F^m \). So \( R \) is an RPF with such a root. Now, we choose \( T \) to be the minimal subtree of \( F^m \) that satisfies \( \mathcal{U}_{\text{npf}} \cup \{R\} \).

Lemma B.2.2 Consider an execution of Approx-Fragment\((G, \mathcal{U})\). If Line 15 is reached, then Line 23 returns a DQF \( F \), such that \( F \) satisfies \( \mathcal{U} \) and \( F \subseteq T_{\text{min}} \cup T_{\text{app}} \).

Proof. Let \( F \) denote the graph \( G_{\text{app}} \) as returned in Line 23. We first prove that \( F \) is a \( d \)-subtree of \( G \) with the root \( r \) (which gets its final value in either Line 15 or Line 17). First, Line 19 removes all the incoming edges of \( r \). Second, observe that each node of \( T_{\text{min}} \cup T_{\text{app}} \) can have at most two incoming edges, namely, one from each \( d \)-subtree. Consequently, due to Lines 20–21, all the nodes of \( F \) have at most one incoming edge.

It is left to prove that every node of \( F \) is reachable from \( r \) through a directed path. We first show that this is true in \( T_{\text{min}} \cup T_{\text{app}} \). Regardless of the result of the test of Line 16, \( \text{root}(T_{\text{min}}) \) is reachable from \( r \). Consequently, every node of \( T_{\text{min}} \) is reachable from \( r \). Theorem 3.5.2 shows that \( T_{\text{app}} \) is reduced w.r.t. \( \mathcal{U} \) and, since Line 6 is reached, the root of \( T_{\text{app}} \) is the root of some NPF of \( \mathcal{U} \) (otherwise, \( T_{\text{app}} \) is an RPF). But \( T_{\text{min}} \) contains all the NPFs of \( \mathcal{U} \), so the nodes of \( T_{\text{app}} \) are reachable from \( r \) as well. We conclude that \( r \) is a root of \( G_{\text{app}} \) in Line 18. Next, we show that none of the actions performed in Lines 19–22 impairs this property. For Line 19 (removal of the incoming edges of \( r \)), it is clear. Consider an edge \((u, v)\) that is removed in Line 21. From the test of Line 21 it follows that \( v \) must be a node of both \( T_{\text{min}} \) and \( T_{\text{app}} \) and, moreover, none of the edges of \( T_{\text{min}} \) is removed in Lines 19–21 (except, possibly, for an incoming edge of \( r \)). Therefore, \( v \) is still reachable from \( r \) after
the removal of the edge \((u, v)\) and, consequently, \(r\) remains a root. Finally, observe that for each node \(v\) that is removed in Line 22, all the nodes of \(G_{\text{app}}\) that are reachable from \(v\) in Line 22 are removed as well. Therefore, reachability from \(r\) of the remaining nodes is not impaired. This completes the proof that \(F\) is a d-subtree of \(G\).

Next, we prove that \(F\) satisfies \(\mathcal{U}\). Clearly, \(T_{\text{min}} \cup T_{\text{app}}\) contains all the PFs of \(\mathcal{U}\). Line 19 does not remove an edge of a PF of \(\mathcal{U}\) for the following reason. If the test of Line 16 is \textit{true}, then \(r\) is the root of a PF of \(\mathcal{U}\) and, since the PFs of \(\mathcal{U}\) are pairwise node disjoint, none of the PFs of \(\mathcal{U}\) contains an incoming edge of \(r\). Otherwise, if the test of Line 19 is \textit{false}, then \(r\) is the root of \(T_{\text{min}}\) and it does not appear in any RPF of \(\mathcal{U}\). Since all the NPFs of \(\mathcal{U}\) are contained in \(T_{\text{min}}\), none of them contains an incoming edge of \(r\).

To see that Line 21 does not remove an edge of a PF of \(\mathcal{U}\), observe the following. Since in both \(T_{\text{app}}\) and \(T_{\text{min}}\) the non-root nodes of the PFs of \(\mathcal{U}\) have no incoming edges other than those of \(\mathcal{U}\) (note the construction of \(G'\) in Lines 6–8), a non-root node of \(\mathcal{U}\) has one incoming edge in \(T_{\text{min}} \cup T_{\text{app}}\) and hence is never chosen in Line 22. Finally, observe that none of the nodes of the PFs of \(\mathcal{U}\) is removed from \(G_{\text{app}}\) in Line 22 since the leaves of each PF are keywords of \(Q\).

It is left to show that \(F\) is reduced. First, Line 22 and the fact that \(F\) satisfies \(\mathcal{U}\) imply that \(\text{leaves}(F) = Q\). So we need to show that \(r\) has at least two edges in \(F\). If \(r\) is the root of an RPF \(R \in \mathcal{U}_{\text{rpf}}\), then \(R\) already contains more than one outgoing edges of \(r\). If \(r\) is the root of \(T_{\text{min}}\), then due to Line 11 and the correctness of Min-Fragment (Theorem 3.6.3), it has at least two outgoing edges in \(T_{\text{min}}\). Moreover, in this case it can easily be observed that none of the edges of \(T_{\text{min}}\) is removed in Lines 19–22 (in particular, observe that none of the nodes of \(T_{\text{min}}\) is removed in Line 22 since the leaves of \(T_{\text{min}}\) are all keywords of \(Q\)), hence it belongs to \(F\). \(\square\)

Interestingly, the loop of Line 10 can be replaced with a single invocation of Min-Fragment by applying the following transformation to \(G'\). First, collapse each RPF of \(\mathcal{V}\) into a single node. Second, add a new keyword \(v\) and edges from the collapsed RPFs to \(v\). Min-Fragment is then called with the set \(\mathcal{V}_{\text{npf}} \cup \{v\}\).
We conclude this section by proving the following theorem, namely, the correctness and efficiency of \textit{Approx-Fragment}. The running-time analysis in this theorem assumes that the above optimization is used. As explained in Section 3.5.3, \textit{Approx-Supertree} uses \textit{Approx-DSteiner} and the latter finds a $\theta$-approximation of a directed Steiner tree. The approximation ratio $\theta$ and the running time $f$ are monotonically nondecreasing functions and, hence, $n$, $e$ and $q$ can be used as their arguments. The constant $c$ denotes the number of NPFs in $\mathcal{U}$. Recall that $c \leq 2$ whenever \textit{Approx-Fragment} is invoked in Line 11 of \textit{EnumDQFs} (Figure 3.3), hence Theorem 3.2.3 immediately follows.

\textbf{Theorem B.2.3} Consider a data graph $G$ with $n$ nodes and $e$ edges. Let $Q$ be a query with $q$ keywords and $\mathcal{U}$ be a set of PF constraints, such that leaves($\mathcal{U}$) = $Q$ and $\mathcal{U}$ has at most $c$ NPFs. The algorithm \textit{Approx-Fragment}(G, $\mathcal{U}$) finds a $(\theta + 1)$-approximation of the minimal DQF that satisfies $\mathcal{U}$ in $O(f n + 3e \log n)$ time, where $\theta$ and $f$ are the approximation ratio and runtime, respectively, of \textit{Approx-DSteiner}.

\textbf{Proof.} The running time easily follows from that of \textit{MIN-FRAGMENT}, so we only prove correctness. Suppose that the algorithm is applied to $G$ and $\mathcal{U}$. We first prove the following claim. If the return value of the algorithm is not $\bot$, then it is a $(\theta + 1)$-approximation of the minimal DQF among those satisfying $\mathcal{U}$. Let $F^m$ be a minimal DQF that satisfies $\mathcal{U}$. If the execution terminates in Line 2, then a minimal DQF is returned. If the execution terminates in Line 5, then the claim follows straightforwardly from the correctness of \textit{Approx-Supertree}, i.e., Theorem 3.5.2 (in this case, we get a $\theta$ approximation). Otherwise, the result is returned in Line 23. Lemma B.2.2 shows that the result is a DQF. This DQF is a subtree of $T_{\text{min}} \cup T_{\text{app}}$. Lemma B.2.1 shows that $w(T_{\text{min}}) \leq w(F^m)$. Since $F^m$ is a $G$-supertree of $\mathcal{U}$, Theorem 3.5.2 shows that $w(T_{\text{app}}) \leq \theta \cdot w(F^m)$. The claim then immediately follows.

To complete the proof, we assume that $G$ contains a DQF $F^m$ that satisfies $\mathcal{U}$ and show that the algorithm does not return $\bot$. Observe that the algorithm cannot return $\bot$ in Line 5 since, by Theorem 3.5.2, the execution of \textit{Approx-Supertree}(G, $\mathcal{U}$) in Line 2 does not return $\bot$ since $F^m$ is a $G$-supertree of $\mathcal{U}$. So the only line that can return $\bot$ is Line 14. But Lemma B.2.1 shows that $T_{\text{min}}$ is not $\bot$ at Line 13, so Line 14 is not executed. \square
Appendix C

Engines for Keyword Proximity Search in the Literature

In recent years, several systems [19, 24, 39, 62, 64, 69, 81, 105] have implemented some form of keyword proximity search. These systems use an engine for generating answers. Some systems [19, 39, 63, 64, 105] present the answers in the order of generating them, whereas other systems [24, 62, 69, 81] apply functions (e.g., based on IR techniques or link analysis) to determine the final ranking order.

The algorithms used for implementing the engines of [19, 62, 64, 81] are based on a relational approach. That is, they store the data in a relational database and use the schema to extract all join expressions that can potentially generate answers (which correspond to $Q$-fragments when representing the database by a data graph as described in Section 3.1.1). Since these expressions are produced (and evaluated) in the order of increasing size, so are the answers. This approach, however, has two major drawbacks. The first is inefficiency. A join expression does not necessarily produce answers for the specific data at hand. Therefore, this approach may evaluate a huge number (i.e., exponential in the size of the data) of expressions without generating even one answer. So, it cannot enumerate with polynomial delay or even polynomial total time. The inefficiency is intensified when queries have relatively many keywords or the schema is complicated (e.g., highly cyclic). The second drawback is that this approach cannot incorporate weights on the nodes and edges of the
data graph. Conceivably, one may define weights on the schema so that the system can take into account the size of each join expression. But it is impossible to differentiate between answers that are generated by the same expression or to use ranking measures that depend on the particular data. Hence, this approach has a very limited ability to correlate between the actual order of generating answers and the desired ranking.

The engines of [24 39 69 105] are based on a graph approach, that is, they apply algorithms that operate directly on the data graph and generate Q-fragments. Overall, this approach leads to more efficient engines than those mentioned earlier. However, it is a subtle task to find all the answers and avoid the generation of redundant subtrees. In fact, none of the algorithms of [24 39 69 105] manages to do that. The work of [39 105] uses a different algorithm from those employed by [24 69], but in both cases the underlying approach is similar. First, they take an algorithm that finds the top-ranked answer, which is either a minimal-height [24 69] or a minimal-weight [39 105] Q-fragment. Then they repeatedly apply that algorithm to generate more answers. Each application is guaranteed to produce a new subtree, because that subtree is forced to have a new node.

The major flaw of the engines of [24 39 69 105] is that they may miss very relevant (i.e., small or short) answers while generating less relevant ones. As we show below, there could be more than one answer that contains the same occurrences of the keywords and the same new node. But the algorithms of [24 39 69 105] can find only one of those answers. Next, we use examples to illustrate this problem.

Given the data graph $G_1$ of Figure 3.1 the DQFs for the query $Q = \{\text{Brussels}, \text{EU}\}$ are $F$ and $F'$ of Figure C.1. The algorithms of [24 69] attempt to find the answers $F$ and $F'$, but may fail to do so. These algorithms apply backward Dijkstra-style (i.e., shortest-path) iterators that start from the keywords and meet at their ancestors. So, given the keywords Brussels and EU, these iterators meet at the nodes $t_3$ and $t_9$. For each ancestor $v$, the algorithms of [24 69] produce only one (minimal-height) answer by combining shortest paths that connect the keywords to $v$. So, these algorithms might compute the subtree $T_1$ of Figure C.1 for $t_9$. But the root of $T_1$ has a single child and so it is deleted in order to obtain a smaller (and, hence, more meaningful) subtree that contains all the given keywords.
The end result is that only one answer, $F$, is found and the second answer, $F'$, is completely missed.

In [39][105], answers are undirected subtrees. For each node $v$ of the graph, the algorithm of [39][105] generates (at most) one answer by finding a minimal-weight subtree that includes $v$ and the keywords. So, the first answer is $F$. Then the algorithm produces two subtrees that have a weight of 4, namely, $T_1$ and $T_2$ (which should be viewed in Figure C.1 as undirected). Neither $T_1$ nor $T_2$ is reduced, but the algorithm of [39][105] does not check whether the generated subtrees are redundant. The answer $F'$ is not produced because for all nodes $v$ of $F'$, there are smaller subtrees that contain $v$ and the keywords (namely, $F$, $T_1$ and $T_2$). The algorithm of [39][105] continues to generate larger subtrees until it prints $k$ of them. But in our example, all these subtrees are redundant except for the first. Moreover, this algorithm misses the answer $F'$. 

Figure C.1: Subtrees of $G_1$
Appendix D

Translations Between Families of p-Documents

In this chapter, we investigate translations among the families of p-documents that are defined in Section 4.3.

D.1 Most General Families

First, we show that $\PrXML^{\{\text{exp, cie}\}}$ is the most general, and the families $\PrXML^{\{\text{exp}\}}$ and $\PrXML^{\{\text{cie}\}}$ are just below $\PrXML^{\{\text{exp, cie}\}}$. (The second part of the next theorem has been independently observed in [99].)

Theorem D.1.1 The following hold.

1. $\PrXML^{\{\text{ind, mux, exp}\}} \equiv_0 \text{poly} \PrXML^{\{\text{exp}\}}$.

2. $\PrXML^{\{\text{ind, mux, cie}\}} \equiv_0 \text{poly} \PrXML^{\{\text{cie}\}}$.

In particular, for every combination $C$ of the above four types, $\PrXML^{\{C\}} \subseteq_0 \text{poly} \PrXML^{\{\text{exp, cie}\}}$.

Proof. In the proof, $v$ denotes a distributional node with children $w_1, \ldots, w_k$. We first prove Part 1. If $v$ is a mux node, then we efficiently translate it into an exp node as follows. We change the type of $v$ to exp and specify the probabilities $p^v(\{w_i\}) = p^v(w_i)$ ($1 \leq i \leq k$)...
Figure D.1: Transforming an ind node into a hierarchy of exp nodes

and \( p^v(\emptyset) = 1 - \sum_{i=1}^{k} p^v(w_i) \). If \( v \) is an ind node, then we replace \( v \) and its children with the subtree shown in Figure D.1. That is, \( v \) is replaced with a new exp node \( v' \) that has \( k \) new exp nodes \( u_1, \ldots, u_k \) as children. The only probability specified by \( v' \) is \( p^{v'}(\{u_1, \ldots, u_k\}) = 1 \); that is, \( v' \) always chooses all of its children. Each \( u_i \) has \( w_i \) as its only child, and it specifies the probabilities \( p^{u_i}(\{w_i\}) = p^v(w_i) \) and \( p^{u_i}(\emptyset) = 1 - p^v(w_i) \).

We now prove Part 2. If the type of \( v \) is ind or mux, we change the type to cie and introduce \( k \) new events \( e_1, \ldots, e_k \). If \( v \) is an ind node, then for all \( 1 \leq i \leq k \), we define \( p(e_i) = p^v(w_i) \) and \( \alpha^v(w_i) = e_i \). Next, suppose that \( v \) is a mux node. We assume that no \( w_i \) satisfies \( p^v(w_i) = 0 \) (otherwise, we remove \( w_i \)). For all \( 1 \leq i \leq k \), we define \( \alpha^v(w_i) = e_i \land \neg e_{i-1} \land \cdots \land \neg e_1 \) and specify the probabilities \( p(e_1) = p^v(w_1), p(e_2) = p^v(w_2)/(1-p^v(w_1)) \) and, in general, \( p(e_i) = p^v(w_i) \cdot \prod_{j=1}^{i-1}(1-p^v(w_j))^{-1} \). Hence, the probability that \( \alpha^v(w_i) \) is true is \( p^v(w_i) \). To show that the probabilities are well defined, we prove that \( p(e_i) < 1 \) for \( 1 \leq i < k \). (We also have to show that \( p(e_k) \leq 1 \) and this is proved similarly.)

Suppose otherwise and consider the smallest \( l \), such that \( p(e_i) \geq 1 \) or equivalently \( p^v(w_i) \geq \prod_{j=1}^{i-1}(1-p^v(w_j)) \cdot (1-\prod_{j=1}^{i-1}(1-p^v(w_j))) \) is the probability that at least one of the events \( e_1, \ldots, e_{l-1} \) is true or, equivalently, exactly one of \( \alpha^v(w_1), \ldots, \alpha^v(w_{l-1}) \) is true. The \( \alpha^v(w_i) \) are disjoint and therefore \( 1 - \prod_{j=1}^{i-1}(1-p^v(w_j)) = \sum_{j=1}^{i-1} p^v(w_j) \). Since we assumed that \( p^v(w_i) \geq \prod_{j=1}^{i-1}(1-p^v(w_j)) \), it follows that \( \sum_{j=1}^{i} p^v(w_j) \geq 1 \), in contradiction to \( p^v(w_{l+1}) > 0 \).

We have shown that \( \PrXML^{\{ind,mux,cie\}} \subseteq_{g} \PrXML^{\{cie\}} \). For proving \( \PrXML^{\{cie\}} \subseteq_{g} \PrXML\{cie\}_w \), we use the following transformation. If \( u \) is a cie node that has a cie child \( v \), then we remove \( v \), connect each child \( w_i \) of \( v \) directly to \( u \) and define \( \alpha^u(w_i) = \alpha^v(u) \land \alpha^v(w_i) \).
Figure D.2: Transforming a hierarchy of \( \text{exp} \) nodes into a single \( \text{exp} \) node

### D.2 Expressive Power of \( \text{exp} \) Nodes

The next theorem shows that \( \PrXML^{\{\text{exp}\}} \) is o- and v-translatable to \( \PrXML^{\{\text{exp}\}}_{|\text{h}} \), but only at a cost of an exponential blowup. Hence, the models of [65, 66], restricted to trees with point probabilities, are not as general as \( \PrXML^{\{\text{exp}\}} \).

**Theorem D.2.1** The following hold.

1. \( \PrXML^{\{\text{ind, mux, exp}\}} \equiv_o \PrXML^{\{\text{exp}\}}_{|\text{h}} \).

2. \( \PrXML^{\{\text{ind, mux, exp}\}}_{|\text{h}} \equiv_{\text{poly}} \PrXML^{\{\text{exp}\}}_{|\text{h}} \).

3. \( \PrXML^{\{\text{ind}\}} \not\subseteq_{\text{poly}} \PrXML^{\{\text{exp}\}}_{|\text{h}} \).

**Proof.** We first prove Part 1. By Theorem [D.1.1] it is sufficient to show that \( \PrXML^{\{\text{exp}\}} \) is o-translatable to \( \PrXML^{\{\text{exp}\}}_{|\text{h}} \), and that can be done by repeatedly applying the following transformation to a \( \mathcal{P} \in \PrXML^{\{\text{exp}\}} \). Consider an ordinary node \( u \) of \( \mathcal{P} \) that has some \( \text{exp} \) children as well as \( \text{exp} \) grandchildren. Let \( T(u) \) be the maximal subtree of \( \mathcal{P} \), such that the root is \( u \), all the interior nodes are distributional and all the leaves are ordinary. We replace \( T(u) \) with \( T'(u) \), as shown in Figure [D.2]. That is, we remove all the interior nodes of \( T(u) \), add a new \( \text{exp} \) node \( v \) as the only child of \( u \), and each leaf of \( T(u) \) becomes a child of \( v \). For each subset \( W \) of the children of \( v \), we define

\[
p^v(W) = \Pr(W \text{ is the child set of } u \text{ in } \mathcal{P} \mid u \in V(\mathcal{P})).
\]

\(^1\text{Therefore, } \PrXML^{\{\text{ind}\}} \not\subseteq_{\text{poly}} \PrXML^{\{\text{exp}\}}_{|\text{h}} \).
Figure D.3: Transforming an ind node (without a hierarchy of distributional nodes) into exp nodes

For Part 2, the proof of Theorem D.1.1 shows how to transform a mux node to an exp node without creating a hierarchy. So, let v be an ind node with the children $w_1, \ldots, w_k$ and the parent u. The transformation\footnote{Note that this transformation (as opposed to the one of Figure D.1) is correct only if there is no hierarchy of distributional nodes.} is illustrated in Figure D.3. We replace v with k new exp nodes $v_1, \ldots, v_k$. Each $v_i$ is a child of u and the parent of $w_i$. For $1 \leq i \leq k$, we define $p^{v_i}(\{w_i\}) = p^v(w_i)$ and $p^{v_i}(\emptyset) = 1 - p^v(w_i)$.

To prove Part 3, consider the p-document $\tilde{\mathcal{P}} \in \PrXML^{\{\text{ind}\}}$ of Figure D.4(a). Nodes v and u of $\tilde{\mathcal{P}}$ choose each one of their children with probability 1/2. The ordinary nodes $w_1, \ldots, w_n$ have n distinct labels $l_1, \ldots, l_n$, respectively.

Suppose that $\tilde{\mathcal{P}}' \in \PrXML^{\{\text{exp}\}} |_{\mathcal{P}}$ satisfies $\tilde{\mathcal{P}}' \equiv_v \tilde{\mathcal{P}}$. It is easy to show that the children of root($\tilde{\mathcal{P}}'$) are exp nodes and the grandchildren are ordinary nodes. We assume that each child of an exp node v of $\tilde{\mathcal{P}}'$ belongs to some subset with a non-zero probability (otherwise, we eliminate that child).

If children of distinct exp nodes have the same label $l_i$, then there is a document $d \in \pwd(\tilde{\mathcal{P}}')$ that has two occurrences of $l_i$, which cannot happen in any document of $\pwd(\tilde{\mathcal{P}})$, in contradiction to $\tilde{\mathcal{P}}' \equiv_v \tilde{\mathcal{P}}$. Therefore, each label occurs under exactly one exp child of root($\tilde{\mathcal{P}}'$).

Now, suppose that the labels $l_i$ and $l_j$ ($i \neq j$) occur below two distinct exp nodes of $\tilde{\mathcal{P}}'$. It means that the occurrence of the label $l_i$ in a document of $\pwd(\tilde{\mathcal{P}}')$ is probabilistically independent of the occurrence of $l_j$ in the same document. But this is not the case in documents of $\pwd(\tilde{\mathcal{P}})$, because if $l_i$ appears in a document $d \in \pwd(\tilde{\mathcal{P}})$, it means that node u of $\tilde{\mathcal{P}}$ is chosen, and hence, the probability that $l_j$ also appears in $d$ is 1/2 and not
1/4. Consequently, $\hat{P}'$ has only one $\text{exp}$ node.

Since every subset of the labels may occur in a random document of $pwd(\hat{P})$, it means the $2^n$ probabilities are specified by the $\text{exp}$ node of $\hat{P}'$. Therefore, the size of this specification is exponential in the size of $\hat{P}$. \hfill $\Box$

### D.3 Expressive Power of cie Nodes

We now discuss the expressive power of $\text{PrXML}^{\{\text{cie}\}}$. A construction of [18] shows that every distribution over XML documents (that have the same label for the root) can be represented by a $\tilde{P} \in \text{PrXML}^{\{\text{cie}\}}$, under the value-based semantics. Hence, $\text{PrXML}^{\{\text{exp,cie}\}}$ is $v$-translatable to $\text{PrXML}^{\{\text{cie}\}}$. But the construction of [18] does not yield an efficient $v$-translation. Whether $\text{PrXML}^{\{\text{exp}\}}$ can be efficiently $v$-translated to $\text{PrXML}^{\{\text{cie}\}}$ is still an open problem, and we conjecture that the answer is negative. However, the next proposition shows that under the object-based semantics, $\text{PrXML}^{\{\text{cie}\}}$ does not even generalize $\text{PrXML}^{\{\text{exp}\}}$. That is, $\text{PrXML}^{\{\text{exp}\}}$ is not $o$-translatable to $\text{PrXML}^{\{\text{cie}\}}$ (and, by Theorem D.1.1 neither to $\text{PrXML}^{\{\text{ind,mux,cie}\}}$).

**Proposition D.3.1** $\text{PrXML}^{\{\text{exp}\}} \not\subseteq_o \text{PrXML}^{\{\text{ind,mux,cie}\}}$.

**Proof.** By Theorem D.1.1 it suffices to show that there is a p-document $\hat{P}' \in \text{PrXML}^{\{\text{exp}\}} \mid_h$, such that no $\hat{P}$ in $\text{PrXML}^{\{\text{cie}\}}$ satisfies $\hat{P}' \equiv_o \hat{P}$. The existence of $\hat{P}'$ is a consequence of the following inequality that holds for all p-documents $\hat{P} \in \text{PrXML}^{\{\text{cie}\}}$ and all ordinary
nodes $w_1$ and $w_2$ of $\tilde{P}$, such that both $w_1$ and $w_2$ appear in at least one document $d$ of $pwd(\tilde{P})$.

$$\Pr(w_1, w_2 \in \mathcal{V}(\tilde{P})) \geq \Pr(w_1 \in \mathcal{V}(\tilde{P})) \cdot \Pr(w_2 \in \mathcal{V}(\tilde{P}))$$

That is, the probability that both nodes exist in a random document is at least as high as the product of the probabilities that each one exists. Clearly, there is a p-document in $\text{PrXML}_{\{\text{exp}\}}$ that violates this inequality, e.g., the one depicted in Figure D.4(b). We prove the inequality as follows.

Let $U$ be a set of ordinary nodes of $\tilde{P}$, such that all the nodes of $U$ appear together in at least one document of $pwd(\tilde{P})$. Consider the minimal r-subtree $p(U)$ of $\tilde{P}$ that contains all the nodes of $U$. Let $A(U)$ be the set of all the literals (i.e., events or negated events) that appear in the conjunctions $\alpha^v(w)$, where $w$ is a node of $p(U)$. $A(U)$ does not contain both an event $e$ and its negation $\neg e$, because $pwd(\tilde{P})$ has a document that contains all the nodes of $U$. Therefore, the probability that all the nodes of $U$ appear in a random document is the product of the probabilities that the literals of $A(U)$ are true. Hence, the inequality follows because $A(\{w_1, w_2\}) = A(\{w_1\}) \cup A(\{w_2\})$. 

Finally, we discuss the expressive power of $\text{PrXML}_{\{\text{exp}\}}$ compared to that of $\text{PrXML}_{\{\text{cie}\}}$. It is easy to prove that $\text{PrXML}_{\{\text{cie}\}} \not\subseteq_o \text{PrXML}_{\{\text{exp}\}}$. It is also easy to show that every distribution over XML documents (that have the same label for the root) can be modeled as a p-document of $\text{PrXML}_{\{\text{exp}\}}$, under the value-based semantics. Hence, $\text{PrXML}_{\{\text{cie,exp}\}}$ is v-translatable to $\text{PrXML}_{\{\text{exp}\}}$. But this is not an efficient v-translation, and an efficient one cannot exist for the following reason. In [99], it is shown that for all $N > 0$, there is a p-document $\tilde{P}_N \in \text{PrXML}_{\{\text{cie}\}}$ with $O(N)$ nodes, such that every v-equivalent p-document $\tilde{P}_N' \in \text{PrXML}_{\{\text{ind,mux}\}}$ has $\Theta(2^N)$ nodes. By slightly modifying the proof of [99], we can show that this result holds even if we allow exp nodes in $\tilde{P}_N'$. To conclude, we get the following result.

**Proposition D.3.2** The following hold.

- $\text{PrXML}_{\{\text{cie}\}} \not\subseteq_o \text{PrXML}_{\{\text{ind,mux,exp}\}}$.

- $\text{PrXML}_{\{\text{cie}\}} \not\subseteq^\text{poly} \text{PrXML}_{\{\text{ind,mux,exp}\}}$. 

24
Appendix E

Proofs of Hardness Results from Chapter 4

E.1 Proof of Theorem 4.7.1

Clearly, a Boolean query can be efficiently evaluated if it satisfies at least one of the conditions of the theorem. To prove the other direction, let $T$ be a fixed pattern that satisfies neither one of the two conditions, namely, $T$ has more than one node, and none of its node predicates is equivalent to false. We consider the problem of computing $\Pr(\mathcal{P} \models T)$, where the input is a p-document, and prove that it is #P-complete. Showing membership in #P is done by adapting the techniques of [56], using the fact that probabilities of p-documents are specified in terms of rational numbers. We prove that the problem is #P-hard as follows.

The results of [90] easily imply that the following problem is #P-complete. Given a Boolean 2-DNF formula $\psi = C_1 \lor \cdots \lor C_m$, where each $C_j$ is a conjunction of two variables $x_{j1} \land x_{j2}$, determine the number $\#\psi$ of satisfying assignments for $\psi$. So, we show a reduction from this problem to the computation of $\Pr(\mathcal{P} \models T)$, where the input is a p-document.

Let $T_1, \ldots, T_k$ be the branches of $T$. We assume that there are no containments among $T_1, \ldots, T_k$, that is, there are no branches $T_i$ and $T_j$ ($i \neq j$), such that $T_i$ is contained in $T_j$.

\footnote{See [16] [46] [84] for details on redundancy and containment among branches of patterns in a fragment of XPath.}
We can always satisfy this assumption by eliminating some branches (namely, we repeatedly remove a branch that contains another one until this assumption is satisfied). So there are \( k - 1 \) documents \( d_1, \ldots, d_{k-1} \), such that for all \( 1 \leq i < k \), (1) \( d_i \models T_i \), and (2) \( d_i \not\models T_k \). Since \( T_1, \ldots, T_k \) have the same root, we can assume that the roots of \( d_1, \ldots, d_{k-1} \) have the same label.

Let \( d_k^- \) be a document such that \( d_k^- \models T_k^c \), where \( c \) is the child of \( \text{root}(T) \) that appears in \( T_k \). Observe that \( d_k^- \) exists since each node predicate of \( T_k \) is satisfied by some label.

Given a 2-DNF formula \( \psi \), we construct the p-document \( \tilde{P}_\psi \) as follows. If \( k > 1 \), then we join the roots of \( d_1, \ldots, d_{k-1} \) and denote by \( r \) the root of the resulting document. Otherwise, if \( k = 1 \), then \( r \) is a new node with a label that satisfies the predicate of \( \text{root}(T) \). We add to \( r \) a new child \( v \). Node \( v \) has \( m \) ordinary children \( w_1, \ldots, w_m \), where each \( w_j \) is the root of a tree that is isomorphic to \( d_k^- \). The conjunctions \( \alpha_v(w_j) \) are defined as follows.

There is a distinct event \( e_i \), with probability \( 1/2 \), for each variable \( x_i \). For all \( 1 \leq j \leq m \), if \( C_j = x_{j_1} \land x_{j_2} \), then \( \alpha_v(w_j) = e_{j_1} \land e_{j_2} \). As an example, Figure E.1 shows \( \tilde{P}_\psi \) for \( \psi = (a \land f) \lor (a \land c) \lor (b \land c) \).

From the construction of \( \tilde{P}_\psi \), it follows that there is a match of \( T \) in a document \( d \in \text{pwd}(\tilde{P}_\psi) \) if and only if \( d \) contains one (or more) of the copies of \( d_k^- \). This happens if and only if \( v \) chooses at least one child. For a subset \( I \subseteq \{1, \ldots, n\} \), let \( \text{occ}(I) \) denote the event “every \( e_i \), where \( i \in I \), is \( \text{true} \) and the rest are \( \text{false} \).” Note that the probability of each \( \text{occ}(I) \) is \( 2^{-n} \). Let \( a_I \) denote the assignment for \( x_1, \ldots, x_n \), such that \( x_i \) is \( \text{true} \) if \( i \in I \) and \( \text{false} \) otherwise. Then, given the event \( \text{occ}(I) \), the random document \( \mathcal{P} \) satisfies \( T \) if and only if \( \psi(a_I) = \text{true} \). Therefore,

\[
\Pr(\mathcal{P} \models T) = \sum_I \Pr(\mathcal{P} \models T \mid \text{occ}(I)) \cdot \Pr(\text{occ}(I)) = \sum_I 2^{-n} = 2^{-n} \#\psi.
\]

Thus, we can compute \( \#\psi \) by simply multiplying \( \Pr(\mathcal{P} \models T) \) by \( 2^n \).

\( ^2 \)Observe that the proof only requires the existence of \( d_1, \ldots, d_{k-1}, d_k^- \) and it is not necessary to show that they can be efficiently constructed, because \( T \) is fixed.
E.2 Proof of Theorem 4.8.3

We first prove that, under query-and-data complexity measure, it is NP-hard to determine whether a query (without projection) evaluates to a nonempty result in PrXML\(^{\text{mux}}\).

**Lemma E.2.1** The following problem is NP-complete under query-and-data complexity. Given a p-document \(\mathcal{P}\) and a pattern \(T\), determine whether there exists a document \(d \in \text{pwd}(\mathcal{P})\), such that \(d \models T\).

**Proof.** Membership in NP follows from the following fact. Testing whether a mapping \(\varphi : \mathcal{V}(T) \rightarrow \mathcal{V}_{\text{ord}}(\mathcal{P})\) is a match of \(T\) in some document of \(\text{pwd}(\mathcal{P})\) can be done in polynomial time (in the size of \(T\) and \(\mathcal{P}\)), as described in the proof of Theorem 4.8.1 (that is, by computing the probability that the minimal r-subtree \(s_\varphi\) of \(\mathcal{P}\) that contains the image of \(\varphi\) is an r-subtree of \(\mathcal{P}^\Sigma\)).

To prove NP-hardness, we show a reduction from 3-SAT that, given a 3-CNF formula \(\psi\), constructs a p-document \(\mathcal{P} \in \text{PrXML}^{\text{mux}}\) and a pattern \(T\), such that \(\psi\) is satisfiable if and only if there is some \(d \in \text{pwd}(\mathcal{P})\), such that \(d \models T\). So, consider a 3-CNF formula \(\psi\). Let \(x_1, \ldots, x_n\) be the variables of \(\psi\). Suppose that \(\psi = C_1 \land \cdots \land C_m\), where each \(C_i\) has the form \(a_i^1 \lor a_i^2 \lor a_i^3\), such that \(a_i^j\) is either a variable or a negated variable.

\(\mathcal{P}\) is constructed as follows. The root \(r\) has \(n\) \(\text{mux}\) children \(v_{x_1}, \ldots, v_{x_n}\). Every node \(v_{x_i}\) has two children \(w_{i}^+\) and \(w_{i}^-\), such that each one is chosen with probability 1/2. There are \(m\) labels \(l_1, \ldots, l_m\) (that correspond to \(C_1, \ldots, C_m\), respectively). There are also additional nodes as follows. If conjunct \(C_j\) includes the variable \(x_i\), then node \(w_{i}^+\) has a child that is
The document $\tilde{\mathcal{P}}$ and pattern $T$ obtained from the 3-CNF formula $\psi = (a \lor b \lor c) \land (a \lor \neg b \lor \neg d) \land (\neg b \lor c \lor d)$

labeled with $l_j$. Similarly, if $C_j$ includes the negated variable $\neg x_i$, then $w_i^-$ has a child that is labeled with $l_j$.

The root of the pattern $T$ has $m$ children $n_1, \ldots, n_m$. Both the root and its children have the predicate $\text{true}$. Each $n_i$ has a single child with the predicate $x = l_i$ (i.e., the label is $l_i$). As an example, Figure E.2 shows the document $\tilde{\mathcal{P}}$ and the pattern $T$ obtained from the 3-CNF formula $\psi = (a \lor b \lor c) \land (a \lor \neg b \lor \neg d) \land (\neg b \lor c \lor d)$.

A document $d \in \text{pwd}(\tilde{\mathcal{P}})$ satisfies $d \models T$ if and only if it includes all the labels $l_1, \ldots, l_n$. Such a document exists if and only if there is a truth assignment for $x_1, \ldots, x_n$ that satisfies all the conjuncts $C_1, \ldots, C_m$.

Proof of Theorem 4.8.3: The first part follows from the fact that Bernstein’s inequality yields an efficient algorithm (under query-and-data complexity) that additively approximates Boolean queries over $\text{PrXML}_{\{\exp, \cie\}}$.

For proving the second part, observe that a multiplicative approximation of $\text{Pr}(\mathcal{P} \models T)$ is greater than 0 if and only if $\text{Pr}(\mathcal{P} \models T) > 0$. Moreover, the latter inequality holds if and only if there exists a document $d \in \text{pwd}(\mathcal{P})$, such that $d \models T$. Hence, Lemma E.2.1 shows that if there is an efficient (under query-and-data complexity) randomized algorithm for multiplicatively approximating $\text{Pr}(\mathcal{P} \models T)$ with probability at least $2/3$, then $\text{NP} \subseteq \text{BPP}$.
Appendix F

Evaluating c-Formulae over p-Documents

In this chapter, we present an efficient algorithm for evaluating a c-formula over a given p-document. We assume that \( \hat{\gamma} \) is a fixed c-formula and the input consists of a p-document \( \tilde{\mathcal{P}} \in \text{PrXML}^{(\text{exp})} \) and a numerical specification of \( \hat{\gamma} \). The goal is to compute the probability \( \Pr(\mathcal{P} \models \hat{\gamma}) \). In the next section, we describe the main steps taken by the algorithm.

F.1 The Main Steps

Our approach is as follows. Firstly, we obtain from \( \hat{\gamma} \) a set \( \Gamma \) of c-formulae. Then, we apply dynamic programming to p-documents that are subtrees of \( \tilde{\mathcal{P}} \) (including \( \tilde{\mathcal{P}} \) itself); for each of those p-documents, we evaluate all the c-formulae \( \gamma \in \Gamma \). Finally, we determine \( \Pr(\mathcal{P} \models \hat{\gamma}) \) by using the results that were computed for \( \tilde{\mathcal{P}} \).

The dynamic-programming algorithm operates in a bottom-up manner over the p-document \( \tilde{\mathcal{P}} \). The set \( \Gamma \) has several crucial properties. First, it can be efficiently constructed. Second, we can efficiently compute \( \Pr(\mathcal{P} \models \hat{\gamma}) \) from probabilities of the form \( \Pr(\mathcal{P} \models \gamma) \) where \( \gamma \in \Gamma \). Third, given a formula \( \gamma \in \Gamma \), it is quite simple to evaluate \( \Pr(\mathcal{P} \models \gamma) \) by using probabilities \( p' \) that involve only the subtrees \( \tilde{\mathcal{P}}' \) of \( \tilde{\mathcal{P}} \) that evolve from the children of the root (which, in principle, facilitates a bottom-up computa-
tion approach); in turn, the probabilities \( p' \) can be computed efficiently from probabilities \( \Pr (\mathcal{P}' \models \gamma) \) where \( \gamma \in \Gamma \), and so on. The exact definition of the set \( \Gamma \) is given in the next section, and in the remainder of this section we simply assume that it has been constructed.

More formally, the algorithm operates on p-documents that are obtained from \( \hat{\mathcal{P}} \) as follows. For an ordinary node \( v \) of \( \hat{\mathcal{P}} \), the p-document \( \hat{\mathcal{P}}^v \) is simply \( \hat{\mathcal{P}}^v \Delta \), i.e., that subtree of \( \hat{\mathcal{P}} \) that is induced by \( v \) and its descendants. In addition, we determine an arbitrary order \( w_1, \ldots, w_m \) over the children of \( v \) and for all \( 0 \leq k \leq m \), the subtree \( \hat{\mathcal{P}}^v_k \) of \( \hat{\mathcal{P}} \) is obtained from \( \hat{\mathcal{P}}^v \) by removing the nodes \( w_{k+1}, \ldots, w_m \) and all their descendants (thus, \( \hat{\mathcal{P}}^v \) is the same as \( \hat{\mathcal{P}}^v_m \) and \( \hat{\mathcal{P}}^v_0 \) consists of only \( v \) itself).

If \( v \) is a distributional node, then we consider each of the subsets \( W \) of the children of \( v \) for which \( v \) specifies a probability \( \hat{\mathcal{P}}(v, W) \), and the p-document \( \hat{\mathcal{P}}^W \) is obtained from \( \hat{\mathcal{P}}^v \Delta \) as follows. First, all the children of \( v \) that are not in \( W \) are removed. Second, \( v \) is replaced with a new ordinary node that is denoted by \( \circ \). The node \( \circ \) has the special property that it only satisfies the predicate \text{true}. We define an order \( w_1, \ldots, w_m \) over the nodes of \( W \), and for all \( 0 \leq k \leq m \), the p-document \( \hat{\mathcal{P}}^W_k \) is obtained from \( \hat{\mathcal{P}}^W \) by removing the nodes \( w_{k+1}, \ldots, w_m \) and all their descendants (similarly to the way \( \hat{\mathcal{P}}^v_k \) is obtained from \( \hat{\mathcal{P}}^v \) when \( v \) is ordinary).

The algorithm \textbf{Eval}(\hat{\gamma})(\hat{\mathcal{P}})\) is shown in Figure F.1. For simplicity of presentation, we omit the numerical specification of \( \hat{\gamma} \) from the input. After constructing the set \( \Gamma \) (Line 1), the algorithm \textit{processes} a p-document \( \hat{\mathcal{P}}' \) by computing \( \Pr (\mathcal{P}' \models \gamma) \) for all \( \gamma \in \Gamma \). So, the algorithm runs as follows. The nodes \( v \) of \( \hat{\mathcal{P}} \) are traversed in bottom-up order. When a node \( v \) is visited, the algorithm processes the p-documents \( \hat{\mathcal{P}}^v \) in ascending order of \( k \) if \( v \) is ordinary; otherwise, if \( v \) is distributional, then the algorithm processes the p-documents \( \hat{\mathcal{P}}^W_k \), in ascending order of \( k \), for each subset \( W \) that has a probability specified by \( v \). In particular, all the p-documents of the form \( \hat{\mathcal{P}}^v \) and \( \hat{\mathcal{P}}^W \) are processed. Finally, \( \Pr (\mathcal{P} \models \hat{\gamma}) \) is computed from the probabilities that were stored when \( \hat{\mathcal{P}} \) itself was processed. In the next sections, we show how to implement Lines 1, 7, 11 and 12 of this algorithm (namely, the construction of \( \Gamma \) and the computation of the probabilities).
Algorithm Eval(\(\hat{\gamma}\))(\(\hat{\mathcal{P}}\))

1: construct the set \(\Gamma\)
2: for all \(v \in \mathcal{V}(\hat{\mathcal{P}})\) in bottom-up order do
3:   for all \(\gamma \in \Gamma\) do
4:     if \(v\) is ordinary then
5:       let \(w_1, \ldots, w_m\) be the children of \(v\)
6:       for all \(k = 0, \ldots, m\) do
7:         compute and store \(\Pr(\mathcal{P}_k^v | = \gamma)\)
8:     else
9:       for all sets \(W\) of nodes, such that \(\mathcal{P}(v, W)\) is specified do
10:         let \(W = \{w_1, \ldots, w_m\}\)
11:         for all \(k = 0, \ldots, m\) do
12:           compute and store \(\Pr(\mathcal{P}_k^W | = \gamma)\)
13:     compute \(\Pr(\mathcal{P} | = \hat{\gamma})\)

Figure F.1: Evaluating \(\Pr(\mathcal{P} | = \hat{\gamma})\)

F.2 Constructing the Set \(\Gamma\)

To describe the set \(\Gamma\), we introduce some definitions and notation. Consider a pattern \(T\) and let \(n\) be a node that does not appear in \(T\). We use \(n/T\) and \(n//T\) to denote the patterns consisting of \(n\) and \(T\) together with a child edge and a descendant edge, respectively, that connect the new root \(n\) to the root of \(T\).

We use a special node \(\ast\) that does not appear in \(\hat{\gamma}\). The node predicate and the attached c-formula of \(\ast\) are \textbf{true}. Two occurrences of \(\ast\) in the same pattern are distinct nodes, and we write them as \(\ast\) and \(\ast'\).

Let \(T\) be a pattern. An r-subtree of \(T\) that comprises the root, exactly one child of the root and all the descendants of that child is called a \textit{branch}. Each edge \((m, n)\) of \(T\) determines exactly one \textit{sub-branch} which is the subtree induced by the nodes \(m, n\) and all the descendants of \(n\). Note that the root of a sub-branch has exactly one child.

Consider an s-formula \(\sigma = \pi_n \alpha T\). For each sub-branch \(\bar{T}\) of \(T\), there is one \textit{quasi sub-formula} \(\pi_{n'} \alpha'T'\) of \(\sigma\) that is defined as follows. We replace the root of \(\bar{T}\) with \(\ast\) to obtain
The function $\alpha'$ extends $\alpha$ by mapping $\star$ (as mentioned above) to the c-formula \texttt{true}. If $n$ is a node of $T'$, then $n' = n$; otherwise, $n' = \star$.

The letter $\delta$ denotes a disjunction of s-formulae of the form $\sigma_1 \lor \cdots \lor \sigma_q$, where $q \geq 1$. (Note that $\sigma$ continues to denote a single s-formula.) An \textit{atomic} c-formula, or just an \textit{atom}, is of the form $\text{Cnt}(\delta)\theta N$. We use the letters $a$ and $b$ exclusively for denoting atoms. In addition, $a^N(\sigma)$ is a shorthand notation for $\text{Cnt}(\sigma) = N$. Similarly, $b^N(\delta)$ is $\text{Cnt}(\delta) = N$.

Note, however, that we may also denote an atom as $a(\sigma)$ or $b(\delta)$ without explicitly writing $N$ as a superscript.

Let $\hat{\gamma}_1, \ldots, \hat{\gamma}_M$ be all the atomic c-formulae that appear in $\hat{\gamma}$ (including the ones that are nested inside s-formulae). For $1 \leq i \leq M$, let $\hat{\gamma}_i$ be $\text{Cnt}(\hat{\delta}_i)\theta_i \hat{N}_i$. Let $1 \leq i \leq M$ be given, and suppose that $\hat{\delta}_i = \hat{\sigma}_1 \lor \cdots \lor \hat{\sigma}_q$ and each $\hat{\sigma}_j$ (where $1 \leq j \leq q$) has the form $\pi_{n_j} \alpha_j T_j$.

The set $\Sigma^*_i$ consists of all the s-formulae $\pi_{n'} \alpha' T'$, where $\pi_{n'} \alpha' T'$ is a quasi sub-formulae of $\hat{\sigma}_j$ for some $1 \leq j \leq q$, and $n' = \star$ (i.e., $n'$ is the root of $T'$). The set $\Sigma^0_i$ comprises all the other quasi sub-formulae of the $\hat{\sigma}_j$ (i.e., quasi sub-formulae $\pi_{n'} \alpha' T'$ of some $\hat{\sigma}_j$, where $n' = n_j$). The \textit{edge count} of $\hat{\gamma}_i$ is the total number of edges in the forest $T_1, \ldots, T_q$, and the \textit{accumulated depth} of $\hat{\gamma}_i$ is the total sum of depths (i.e., number edges on the path from the root) of the output nodes $n_j$. Observe that the number of quasi sub-formulae in $\Sigma^0_i \cup \Sigma^*_i$ is the edge count of $\hat{\gamma}_i$, and the number of quasi sub-formulae in $\Sigma^0_i$ is the accumulated depth of $\hat{\gamma}_i$.

We assume that $0 \leq \hat{N}_i \leq |\mathcal{V}^{\text{ord}}(\hat{\mathcal{P}})|$, because if this is not the case, then we can easily replace $\text{Cnt}(\hat{\delta}_i)\theta_i \hat{N}_i$ with either \texttt{true} or \texttt{false}.

A \textit{companion} of $\hat{\gamma}_i$ ($= \text{Cnt}(\hat{\delta}_i)\theta_i \hat{N}_i$) is a c-formula of the form $a_1^0(\sigma_1) \land \cdots \land a_l^0(\sigma_l) \land b_1^{N_1}(\delta_1) \land \cdots \land b_r^{N_r}(\delta_r)$ where:

- $0 \leq l \leq |\Sigma^*_i|$
- $0 \leq r \leq 2|\Sigma^i|$
- $0 \leq N_k \leq \hat{N}_i$ for $k = 1, 2, \ldots, r$
- $\sigma_1, \ldots, \sigma_l$ are distinct elements of $\Sigma^*_i$
• The disjuncts of each \( \delta_k \) \((1 \leq k \leq r)\) are distinct elements of \( \Sigma_i^o \)

• \( \delta_1, \ldots, \delta_r \) are distinct disjunctions

Note that if \( l = r = 0 \), then the companion is \textit{true}.

The set \( \Gamma \) consists of all c-formulae of the form \( \gamma_1 \land \cdots \land \gamma_M \), where \( \gamma_i \) is a companion of \( \hat{\gamma}_i \) \((1 \leq i \leq M)\). We remove the \( \gamma_i \) that are the c-formula \textit{true}.

**Proposition F.2.1** Let \( E \) be the total number of edges in the patterns of the s-formulae \( \pi_{n_j} \alpha_j T_j \) that appear in \( \hat{\gamma}_i \). Let \( h \) be the maximal accumulated depth among those of the \( \hat{\gamma}_i \).

Then
\[
|\Gamma| \leq 2^E + 2^h M \prod_{i=1}^{M} (N_i + 1)^{2^h}.
\]

**Proof.** Consider an atom \( \hat{\gamma}_i \). Let \( E_i \) be the edge count of \( \gamma_i \) and \( h_i \) be the accumulated depth of \( \gamma_i \). Recall that \( E_i = |\Sigma_i^* \cup \Sigma_i^o| \) and \( h_i = |\Sigma_i^o| \). A companion of \( \hat{\gamma}_i \) is obtained by
(1) choosing a subset of \( \Sigma_i^* \), and (2) choosing some of the subsets of \( \Sigma_i^o \) and an operand of \{0, \ldots, N_i\} for each subset. The number of choices for (1) is at most \( 2^{E_i} \) while that for (2) is at most \( 2^{2^{h_i}} \cdot (N_i + 1)^{2^{h_i}} \). Therefore, the number of companions of \( \hat{\gamma}_i \) is at most \( 2^{E_i + 2^h (N_i + 1)^{2^h}} \). The proposition follows by taking the product for all the \( \hat{\gamma}_i \).

The following is a convenient terminology. Recall that a c-formula \( \gamma \) is a conjunction of atoms of the form \( b_1 (\sigma_{1,1} \lor \cdots \lor \sigma_{1,l_1}) \land \cdots \land b_k (\sigma_{k,1} \lor \cdots \lor \sigma_{k,l_k}) \) where \( \sigma_{i,j} = \pi_{n_{i,j}} \alpha_{i,j} T_{i,j} \).

We say that the atoms \( b_i \), the s-formulae \( \sigma_{i,j} \) and the patterns \( T_{i,j} \) are at the \textit{top-level} of \( \gamma \), whereas all the other atoms, s-formulae and patterns (that appear in \( \gamma \)) are \textit{nested}.

**F.3 First Case: Ordinary Node with at Most One Child**

In this section, we describe the computation of \( \Pr(\mathcal{P}_v \models \gamma) \), where \( \gamma \in \Gamma \), assuming that \( v \) is ordinary and has at most one child, denoted by \( w \). The same computation is used for the case where \( v \) has multiple children and the goal is to compute \( \Pr(\mathcal{P}_k \models \gamma) \) for \( k \leq 1 \).

We consider three cases.

\footnote{Note that this includes the choice of the empty conjunction (i.e., \textit{true}) as a companion of \( \hat{\gamma}_i \).}
Case 1a: \( v \) is childless. We evaluate \( \gamma \) over \( v \), which is an ordinary node in this case. Recall that in all the top-level patterns of \( \gamma \), the root is \( \star \) and has exactly one child (also, both the predicate and the attached c-formula of \( \star \) are true). Hence, all the top-level s-formulae evaluate to the empty set. So, in this case, an atom \( \text{Cnt}(\delta) = N \) is satisfied if and only if \( N = 0 \). Thus, we evaluate \( \gamma \) and if the result is true, then \( \Pr(\mathcal{P}^v \models \gamma) = 1 \); otherwise, it is 0.

Case 1b: \( w \) is distributional. Let \( W_1, \ldots, W_l \) be the subsets of the children of \( w \) for which a probability \( \tilde{P}(w, W) \) is specified. By the principle of total probability, the following holds.

\[
\Pr(\mathcal{P}^v \models \gamma) = \sum_{i=1}^{l} \Pr(\mathcal{P}^v \models \gamma \mid w \text{ chooses } W_i) \cdot \tilde{P}(w, W_i)
\]

Given that \( w \) chooses \( W_i \), the random variable \( \mathcal{P}^v \) actually corresponds to the following p-document. From \( \tilde{P}^v \), we remove \( w \) and all its children, except for those of \( W_i \) which become the new children of \( v \). Recall that \( \circ \) is actually the root of \( \tilde{P}^{W_i} \) for all \( 1 \leq i \leq l \). Hence, evaluating \( \gamma \) over \( \tilde{P}^v \) (given that \( w \) chooses \( W_i \)) is the same as evaluating it over \( \tilde{P}^{W_i} \), because all the top-level patterns of \( \gamma \) have \( \star \) as their roots (and both the predicate and the attached c-formula of \( \star \) are true). Therefore, \( \Pr(\mathcal{P}^v \models \gamma \mid w \text{ chooses } W_i) = \Pr(\mathcal{P}^{W_i} \models \gamma) \).

Furthermore, the right-side probability has already been computed when visiting \( w \).

Case 1c: \( w \) is ordinary. Let \( \gamma = a_0^1(\sigma_1) \land \cdots \land a_0^l(\sigma_l) \land b_1^{N_1}(\delta_1) \land \cdots \land b_q^{N_q}(\delta_q) \). The rough idea is as follows. First, we remove the roots (i.e., \( \star \)) from the top-level patterns of \( \gamma \). Recall that each of these patterns is a sub-branch, that is, the root has a single child. Hence, we obtain well-defined patterns after removing the roots. Let \( \xi \) be the resulting c-formula. Similarly to the previous case, we would like to use the equality \( \Pr(\mathcal{P}^v \models \gamma) = \Pr(\mathcal{P}^w \models \xi) \) and the already computed value of the right side. We expect this equality to hold because \( w \) is the only child of \( v \), the c-formula \( \xi \) is the result of removing the roots (namely, \( \star \)) from the top-level patterns of \( \gamma \), and both the node predicate and the attached c-formula of \( \star \) are true. But there are three problems. First, the equality does not hold if descendant edges emanate from the roots of some top-level patterns of \( \gamma \). Second, if \( \pi_n \alpha T \) is a top-level s-formula, such that \( n = \text{root}(T) \), then the projection is no longer well-defined once we
remove the root of $T$. Third, $\xi$ is not necessarily in $\Gamma$. We deal with the first two problems by changing $\gamma$ (before removing the roots) as follows.

Given an s-formula $\sigma = \pi_n \alpha(\star//T)$, we define $\sigma' = \pi_n \alpha(\star/T)$ and $\sigma'' = \pi_n \alpha(\star'//T)$. Clearly, $\sigma$ is equivalent to the union of $\sigma'$ and $\sigma''$.

We solve the first problem as follows. If $\gamma$ has an atom $a_0^j(\sigma_j)$, such that $\sigma_j = \pi_n \alpha(\star//T)$, then we replace $a_0^j(\sigma_j)$ with $\text{Cnt}(\sigma'/j) = 0 \land \text{Cnt}(\sigma''/j) = 0$. And if $\gamma$ has an atom $b_{N_j}^j(\delta_j)$, such that $\sigma = \pi_n \alpha(\star//T)$ is a disjunct of $\delta_j$, then we replace $\sigma$ with $\sigma' \lor \sigma''$. Note that after removing the roots from $\sigma'$ and $\sigma''$, the former has no occurrence of $\star$ (or $\star'$), whereas the latter has one—its root (which has a single child). The same holds for $\sigma'/j$ and $\sigma''/j$ that were used earlier.

Now, consider the second problem. If $\pi_n \alpha T$ is a top-level s-formula of $\gamma$, such that $n = \text{root}(T)$, then by the definition of a companion, it occurs in an atom $a_j^0(\sigma_j)$. Hence, we can replace $a_j^0(\pi_n \alpha T)$ with $a_j^0(\pi_n' \alpha T)$, where $n'$ is the child of $n$, because a document satisfies $a_j^0(\pi_n \alpha T)$ if and only if it satisfies $a_j^0(\pi_n' \alpha T)$ (due to the fact that the root $\star$ of $T$ is satisfied by the root of every document).

After modifying $\gamma$ in order to solve the first two problems, we remove the roots of all the top-level patterns. The result is a well-defined c-formula $\xi$.

Correctness of the process of obtaining $\xi$ from $\gamma$ is shown in the following proposition, and the proof is immediate from the above discussion.

**Proposition F.3.1** $\Pr(\mathcal{P}^v \models \gamma) = \Pr(\mathcal{P}^w \models \xi)$.

Observe that $\xi$ is not necessarily in $\Gamma$, so we cannot compute $\Pr(\mathcal{P}^w \models \xi)$ directly from the stored results of previous steps. In the next section, we compute $\Pr(\mathcal{P}^w \models \xi)$ by transforming it into an equal sum $\sum_{i=1}^{2^t} \pm \Pr(\mathcal{P}^w \models \gamma_i)$ where all the $\gamma_i$ are in $\Gamma$. Hence, all the probabilities of this sum have already been computed when $\tilde{\mathcal{P}}^w$ was processed.

---

2The sign $\pm$ means that some of the terms may be preceded by the minus operator.
F.4 Transformations

In this section, we describe transformations that are applied to $\Pr(\mathcal{P}w \models \xi)$. The c-formula $\xi$ is either one of the following. In the last step of the algorithm (namely, Line 13 of Figure F.1), $\xi$ is $\hat{\gamma}$ itself. During the execution of the algorithm, $\xi$ is obtained from some $\gamma \in \Gamma$ as described in the previous section.

By applying the transformations as described below, we convert $\Pr(\mathcal{P}w \models \xi)$ into an equal sum $\sum_{i=1}^{r} \pm \Pr(\mathcal{P}w \models \gamma_i)$, such that every $\gamma_i$ is in $\Gamma$. In the initial sum, $\Pr(\mathcal{P}w \models \xi)$ itself is the only term. A transformation replaces a term $\Pr(\mathcal{P}w \models \gamma_i)$ with some other terms (although loosely speaking, we sometimes say that the transformation is applied to $\gamma_i$ to replace it with some other c-formulae). Each transformation has a precondition. A transformation can be applied to $\gamma_i$ provided that $\gamma_i$ satisfies the precondition. The transformations are applied in the order in which they are presented. Each transformation is applied repeatedly until the precondition is not satisfied by any $\gamma_i$. When all the transformations terminate, every $\gamma_i$ is in $\Gamma$. In the next section, we prove the correctness and termination (in polynomial time) of the transformations; in particular, we show that applying the next transformation does not create new opportunities to satisfy the preconditions of the previous ones.

**TRANS. ExCR: Extracting c-formulae from roots.**

**PRECONDITION:** $\gamma_i$ is $\gamma \land a(\delta \lor \pi_n \alpha T)$ and $\alpha(\text{root}(T)) \neq \text{true}.$

This transformation applies when $\gamma_i$ has a top-level pattern $T$, such that the c-formula attached to the root of $T$ is not true. So, let $\gamma_0$ be an atom of $\alpha(\text{root}(T))$. The mapping $\alpha'$ is the same as $\alpha$, except that it maps $\text{root}(T)$ to the c-formula obtained by removing $\gamma_0$ from $\alpha(\text{root}(T))$. Then,

$$\Pr(\mathcal{P}w \models \gamma \land a(\delta \lor \pi_n \alpha T)) = \Pr(\mathcal{P}w \models \gamma \land a(\delta) \land \neg \gamma_0) +$$

$$+ \Pr(\mathcal{P}w \models \gamma \land a(\delta \lor \pi_n \alpha' T) \land \gamma_0).$$

In words, the original probability is the sum (of the probabilities) of two disjoint events. If
a random document $d$ rooted at $w$ does not satisfy $\gamma_0$, then the result of $\pi_n \alpha T$ is empty and we can\footnote{Note that it would be wrong just to remove $\gamma_0$ from the c-formula $\alpha(\text{root}(T))$.} completely remove it. And if $d$ satisfies $\gamma_0$ as a conjunct of the top-level c-formula, then there is no need to include $\gamma_0$ in the c-formula attached to the root of $T$.

The c-formula $\neg \gamma_0$ is obtained from the atom $\gamma_0$ by replacing the operator $\otimes$ with its complement $\bar{\theta}$.

**Trans. ENR:** Elimination of non-star roots.

**Precondition:** $\gamma_i$ has an atom $a(\delta \lor \pi_n \alpha T)$, such that $\text{root}(T) \neq \star$.

Firstly, note that at this point (namely, after applying the first transformation), the c-formula attached to the root of $T$ is true. If $w$ violates the node predicate of $\text{root}(T)$, then we remove $\pi_n \alpha T$. Otherwise, the root of $T$ is converted to $\star$, and if $n$ is the old root of $T$, then $\star$ replaces $n$.

**Trans. EMB:** Elimination of multiple branches.

**Precondition:** $\gamma_i$ is $\gamma \land a(\delta \lor \pi_n \alpha T)$ and $\text{root}(T)$ has more than one child.

Let $T_b$ be a branch of $T$ with the following property. If $n \neq \text{root}(T)$, then $T_b$ does not contain $n$; otherwise, $T_b$ is an arbitrary branch. Let $T_{\bar{b}}$ be obtained from $T$ by removing all the nodes of $T_b$, except for the root. Let $\sigma_b = \pi_{\text{root}(T)} \alpha T_b$. Then,

$$
\Pr(\mathcal{P}^w \models \gamma \land a(\delta \lor \pi_n \alpha T)) = \Pr(\mathcal{P}^w \models \gamma \land a(\delta) \land \text{CNT}(\sigma_b) = 0) + \\
+ \Pr(\mathcal{P}^w \models \gamma \land a(\delta \lor \pi_n \alpha T_{\bar{b}}) \land \text{CNT}(\sigma_b) > 0).
$$

In words, if for a random document rooted at $w$, the result of $\sigma_b$ is empty, then we can completely remove $\pi_n \alpha T$; otherwise, the branch $T_b$ can be removed from $T$.

We replace the second term in the above sum with

$$
\Pr(\mathcal{P}^w \models \gamma \land a(\delta \lor \pi_n \alpha T_{\bar{b}})) - \Pr(\mathcal{P}^w \models \gamma \land a(\delta \lor \pi_n \alpha T_{\bar{b}}) \land \text{CNT}(\sigma_b) = 0)
$$

since $\text{CNT}(\sigma_b) = 0$ is “closer” to the definition of $\Gamma$.\footnote{Note that it would be wrong just to remove $\gamma_0$ from the c-formula $\alpha(\text{root}(T))$.}
TRANS. **ENE:** *Elimination of non-equality operators.*

**Precondition:** $\gamma_i$ is $\gamma \land \text{Cnt}(\delta) \otimes N$ and $\otimes$ is not $\neq$.

Let $Y = \Pr(\mathcal{P}^w | \ = \gamma \land \text{Cnt}(\delta) \otimes N)$. We replace $\text{Cnt}(\delta) \otimes N$ with atoms of the form $\text{Cnt}(\delta) = N'$, where $0 \leq N' \leq N$. For example, if $\otimes$ is $\neq$, then

$$Y = \Pr(\mathcal{P}^w \models \gamma) - \Pr(\mathcal{P}^w \models \gamma \land \text{Cnt}(\delta) = N).$$

If $\theta$ is $\leq$, then

$$Y = \sum_{i=0}^{N} \Pr(\mathcal{P}^w \models \gamma \land \text{Cnt}(\delta) = i).$$

Similarly, if $\otimes$ is $<$, then the sum ends at $N - 1$. In the case of $\geq$,

$$Y = \Pr(\mathcal{P}^w \models \gamma) - \sum_{i=0}^{N-1} \Pr(\mathcal{P}^w \models \gamma \land \text{Cnt}(\delta) = i).$$

Finally, if $\otimes$ is $>$, then the above sum ends at $N$.

TRANS. **EPR:** *Elimination of projection on roots.*

**Precondition:** $\gamma_i$ is $\gamma \land \text{Cnt}(\sigma_1 \lor \cdots \lor \sigma_q) = N$ where either $q > 1$ or $N > 0$, and there is a $\sigma_j = \pi_n \alpha T$ $(1 \leq j \leq q)$ such that $n = \text{root}(T)$.

Without loss of generality, suppose that in $\sigma_1, \ldots, \sigma_k$ $(1 \leq k \leq q)$ the projection is on the root, while in the other $\sigma_j$ it is not. We define the atoms $a_N$ and $a_{N-1}$ and the conjunction $\gamma_-$ as follows.

$$a_N = (\text{Cnt}(\sigma_{k+1} \lor \cdots \lor \sigma_q) = N)$$
$$a_{N-1} = (\text{Cnt}(\sigma_{k+1} \lor \cdots \lor \sigma_q) = N - 1)$$
$$\gamma_- = (\text{Cnt}(\sigma_1) = 0 \land \cdots \land \text{Cnt}(\sigma_k) = 0)$$
Then,

\[
\Pr(\mathcal{P}^w \models \gamma \land \text{Cnt}(\sigma_1 \lor \cdots \lor \sigma_q) = N) = \\
= \Pr(\mathcal{P}^w \models \gamma \land a_N \land \neg \gamma_\neg) + \Pr(\mathcal{P}^w \models \gamma \land a_{N-1} \land \neg \gamma_\neg) = \\
= \Pr(\mathcal{P}^w \models \gamma \land a_N \land \neg \gamma_\neg) + \Pr(\mathcal{P}^w \models \gamma \land a_{N-1}) - \Pr(\mathcal{P}^w \models \gamma \land a_{N-1} \land \neg \gamma_\neg).
\]

In words, if for a random document rooted at \(w\), the result of \(\sigma_1 \lor \cdots \lor \sigma_k\) is empty, then \(a_N\) should hold; otherwise, that result consists exactly of \(w\), hence \(a_{N-1}\) should hold.

**Trans. ETC:** Elimination of trivial c-formulae.

**Precondition:** \(\gamma_i\) has an atom \(a\) that is either \(\text{Cnt}() \otimes N\), or \(\text{Cnt}(\pi \otimes T) \otimes N\) where \(\ast\) is the only node of \(T\).

Note that \(\text{Cnt}(\pi \otimes T) = 1\) and \(\text{Cnt}() = 0\). So, we evaluate the atom \(a\). If the result is **true**, then we remove this atom from \(\gamma_i\). Otherwise, \(\gamma_i\) evaluates to **false**; hence, we completely remove it.

**Trans. ERD:** Elimination of repeated disjunctions and disjuncts.

**Precondition:** \(\gamma_i\) has either two atoms of the form \(\text{Cnt}(\delta) = N_1\) and \(\text{Cnt}(\delta) = N_2\) (i.e., the same \(\delta\) in both), or an atom \(a(\sigma_1 \lor \cdots \lor \sigma_l)\) such that not all of \(\sigma_1, \ldots, \sigma_l\) are distinct.

In the first case, if \(N_1 \neq N_2\), then \(\gamma_i\) evaluates to **false** and can be removed, otherwise one of the two atoms is removed from \(\gamma_i\). In the second case, we remove duplicates from \(\sigma_1 \lor \cdots \lor \sigma_l\).

**F.4.1 Correctness and Efficiency**

A transformation is applied to (the probability of) \(\gamma_i\). To show termination, we define below the **potential** of each transformation with respect to \(\gamma_i\). It is easy to see that the potential decreases in each of the new c-formulae that replace \(\gamma_i\). So, every transformation by itself terminates when it is applied repeatedly until no \(\gamma_i\) satisfies the precondition. **Transform** is the process of applying the transformations one-by-one in the order of their appearance in
the previous section. We show below that the application of the next transformation does not create new opportunities for applying any of the previous ones.

The potential of ExCR (the first transformation) is the total number of atoms in the c-formulae that are attached to roots of patterns (including nested ones).

The transformation ENR changes roots to $\star$ in some of the top-level patterns and removes the s-formulae containing the others. So, it clearly terminates and does not create new opportunities for applying the previous transformation.

The potential of EMB is defined as follows. We take the number of branches minus one in each top-level pattern and then sum over all those patterns. EMB does not create new opportunities to apply either one of the previous two transformations, because it does not change existing roots, and new patterns have the same roots as old ones (namely, $\star$).

For each one of ENE and EPR, the potential is the number of atoms that satisfy the precondition. Neither ENE nor EPR changes (or introduces new) s-formulae. Furthermore, EPR does not introduce inequalities. So, neither ENE nor EPR creates new opportunities to apply any of the previous transformations.

Finally, ETC and ERD only delete existing atoms and s-formulae.

**Proposition F.4.1** Transform terminates after a polynomial number (in $P$ and the numerical specification of $\hat{\gamma}$) of applications of the transformations.

**Proof.** For a c-formula $\gamma'$, let $|\gamma'|$ denote the size of $\gamma'$, namely, the number of nodes (of patterns, including nested ones) and symbols used by $\gamma'$. Note that the size of $\gamma'$ excludes the numerical specification for the operands. Let $N$ be the maximal operand given in the numerical specification. Note that $m_{\Gamma}$ depends only on $\gamma$, and not on the numerical specification. Recall our (not limiting) assumption that $N \leq |v^{ord}(\hat{\mathcal{P}})|$.

It can be easily verified from the definition of the potentials that the total sum of potentials for $\gamma_i = \xi$ (i.e., the original c-formula) is at most $|\xi|$. All the transformations create new c-formulae, such that the size of each one is linear in the size of $\gamma_i$. Each transformation, except ENE, replaces $\gamma_i$ with at most three c-formulae. The number of new c-formulae introduced by ENE is at most $N$, because every $N_i$ that appears in $\hat{\gamma}$ (or
introduced by either ENE or EPR) satisfies $N_i \leq N$. Thus, since transformations result in c-formulae with lower potentials, the algorithm must terminate after $O(N^{1/\xi})$ applications of transformations.

The next proposition shows the correctness of the transformations. Its proof follows from the discussion in the previous sections and the fact that if a $\gamma_i$ (that is obtained when Transform is applied to $\xi$) does not satisfy any precondition, then $\gamma_i \in \Gamma$.

**Proposition F.4.2** Let $\sum_{r_i=1}^r \pm \Pr(P^w | = \gamma_i)$ be the result of Transform. The following hold.

- $\Pr(P^w | = \xi) = \sum_{i=1}^r \pm \Pr(P^w | = \gamma_i)$.
- $\gamma_i \in \Gamma$ for all $1 \leq i \leq r$.

**F.5 Second Case: Ordinary Node with Multiple Children**

We now show how to compute $\Pr(P^v_k | = \gamma)$ for $\gamma \in \Gamma$, given that $v$ is ordinary and $k > 1$. Recall that at this point, the algorithm Eval($\hat{\gamma}$) of Figure F.1 has already computed $\Pr(P^v_{k-1} | = \gamma')$ for all $\gamma' \in \Gamma$. Let $P'$ denote the p-document that is obtained from $P^v_k$ by removing $w_1, \ldots, w_{k-1}$ (and their descendants), so that $w_k$ remains as the only child of $v$.

Let $\gamma$ be $\text{Cnt}(\delta_1) = N_1 \land \cdots \land \text{Cnt}(\delta_q) = N_q$. If $\gamma'$ is of the form $\text{Cnt}(\delta_i) = N_i' \land \cdots \land \text{Cnt}(\delta_q) = N_q'$, where $0 \leq N_i' \leq N_i$ $(1 \leq i \leq q)$, then we say that $\gamma'$ is covered by $\gamma$. Let $\gamma'$ be covered by $\gamma$. The complement of $\gamma'$ w.r.t. $\gamma$, denoted by $\bar{\gamma}'$, is the c-formula $\text{Cnt}(\delta_1) = \bar{N}_1' \land \cdots \land \text{Cnt}(\delta_q) = \bar{N}_q'$, where $\bar{N}_i' = N_i - N_i'$ $(1 \leq i \leq q)$. We use $\text{Cvrd}(\gamma)$ to denote the set of all the c-formulae that are covered by $\gamma$. Note that $\text{Cvrd}(\gamma)$ is closed under complementation and $\text{Cvrd}(\gamma_1) \subseteq \Gamma$.

**Proposition F.5.1** $\Pr(P^v_k | = \gamma)$ is equal to

$$\sum_{\gamma' \in \text{Cvrd}(\gamma)} \Pr(P' | = \gamma') \cdot \Pr(P^v_{k-1} | = \bar{\gamma}')$$
Proof. We divide the random variable $\mathcal{P}_k^v$ (which corresponds to $\tilde{D}_k^v$) into $D'$ and $D_{k-1}^v$. The random variable $D'$ corresponds to $\mathcal{P}'$ (where $w_k$ is the only child of $v$), while $D_{k-1}^v$ corresponds to $\tilde{D}_{k-1}^v$. Note that $v$ is the only node that is common to $D'$ and $D_{k-1}^v$. Moreover, $D'$ and $D_{k-1}^v$ are probabilistically independent. We write $\gamma$ as a conjunction $\gamma_0 \land \gamma_1$, such that the atoms of $\gamma_0$ are of the form $\text{CNT}(\sigma_1 \lor \cdots \lor \sigma_k) = 0$, while those of $\gamma_1$ are $\text{CNT}(\sigma_1 \lor \cdots \lor \sigma_k) = N$ where $N > 0$.

By the definition of $\Gamma$, in every top-level pattern of $\gamma$, the root has one child. Therefore, a match of a top-level pattern in $\mathcal{P}_k^v$ must be completely embedded in either $D'$ or $D_{k-1}^v$. It thus follows that $\mathcal{P}_k^v \models \gamma_0$ holds if and only if both $D' \models \gamma_0$ and $D_{k-1}^v \models \gamma_0$ do. Note that a c-formula $\text{CNT}(\delta_i) = 0$ is identical to its correspondent in the complement.

Now, consider a conjunct $\text{CNT}(\sigma_1 \lor \cdots \lor \sigma_i) = N$ of $\gamma_1$. By the definition of $\Gamma$ and the fact that $N > 0$, the projection in each $\sigma_i$ is not on the root. Therefore, the sets of nodes of $\cup_i \sigma_i(D')$ and $\cup_i \sigma_i(D_{k-1}^v)$ are disjoint. Consequently, $\mathcal{P}_k^v \models \text{CNT}(\sigma_1 \lor \cdots \lor \sigma_i) = N$ holds if and only if both $D' \models \text{CNT}(\sigma_1 \lor \cdots \lor \sigma_i) = N'$ and $D_{k-1}^v \models \text{CNT}(\sigma_1 \lor \cdots \lor \sigma_i) = \bar{N}'$ hold for some $N'$ and $\bar{N}'$ such that $N' + \bar{N}' = N$.

Consequently, $\mathcal{P}_k^v \models \gamma$ holds if and only if for some $\gamma' \in \text{Cvrd}(\gamma)$, both $D' \models \gamma'$ and $D_{k-1}^v \models \bar{\gamma'}$ hold. Observe that two distinct c-formula $\gamma'_1$ and $\gamma'_2$ are contradictory (that is, their satisfactions by a random document are disjoint events). So, we conclude that $\Pr(\mathcal{P}_k^v \models \gamma)$ is equal to

$$\sum_{\gamma' \in \text{Cvrd}(\gamma)} \Pr(D' \models \gamma' \land D_{k-1}^v \models \bar{\gamma'}) = \sum_{\gamma' \in \text{Cvrd}(\gamma)} \Pr(D' \models \gamma') \cdot \Pr(D_{k-1}^v \models \bar{\gamma'}) .$$

The last equality follows from the independence of $D'$ and $D_{k-1}^v$. To finish the proof, observe that $D'$ and $D_{k-1}^v$ have the same distributions as $\mathcal{P}'$ and $\tilde{D}_{k-1}^v$, respectively. \hfill \Box

By Proposition F.5.1, it suffices to compute for all $\gamma' \in \text{Cvrd}(\gamma)$, the probabilities $\Pr(\mathcal{P}' \models \gamma')$ and $\Pr(\tilde{D}_{k-1}^v \models \bar{\gamma'})$. The first value is computed as in the previous sections, because $v$ has only one child in $\mathcal{P}'$. The second value has been previously computed.
F.6 Third Case: Distributional Node

The last case that is left to consider is when $v$ is distributional, namely, the goal is to compute the probability $\Pr(\mathcal{P}_k^W = \gamma)$ for some $\gamma \in \Gamma$, subset $W$ of the children of $v$ (for which a probability is specified by $v$), and a number $k$ ($0 \leq k \leq |W|$). Recall that the root of $\tilde{\mathcal{P}}_k^W$ is $\circ$ (which is an ordinary node). Hence, in this case, we use one of the computations described in Sections F.3 and F.5, depending on the number $k$ and, if $k = 1$, the type of the first node of $W$. Note that in those computations, the node $\circ$ is used instead of $v$.

F.7 The Final Computation and Total Execution Time

In the algorithm $\text{Eval}(\hat{\gamma})(\tilde{\mathcal{P}})$ of Figure F.1, the final step is Line 13 that computes $\Pr(\mathcal{P} = \hat{\gamma})$. We do it by applying $\text{Transform}$ to $\Pr(\mathcal{P} = \hat{\gamma})$ (as described in Section F.4) in order to get a sum $\sum_{i=1}^{r} \pm \Pr(\mathcal{P} = \gamma_i)$, such that all the $\gamma_i$ are in $\Gamma$. All the probabilities of this sum have already been computed when $\tilde{\mathcal{P}}$ itself was processed.

Finally, we analyze the running time of the algorithm $\text{Eval}(\hat{\gamma})(\tilde{\mathcal{P}})$ and show that it is bounded by a polynomial in the input (i.e., $\tilde{\mathcal{P}}$ and the numerical specification). For simplicity of presentation, we assume that each of the arithmetic operations (e.g., addition and multiplication of probabilities) takes a constant time.

Let $|\tilde{\mathcal{P}}|$ denote the size of the representation of $|\tilde{\mathcal{P}}|$, namely, the number of nodes of $\tilde{\mathcal{P}}$, and the total sum of the cardinalities of the sets $W$ for which a probability $\tilde{\mathcal{P}}(v, W)$ is specified. By the pseudo code of Figure F.1, after constructing the set $\Gamma$ the algorithm iterates over all the p-documents $\tilde{\mathcal{P}}_k^v$ (in a bottom-up order) and for each $\tilde{\mathcal{P}}_k^v$, it evaluates all the queries of $\Gamma$. Observe that the number of p-documents $\tilde{\mathcal{P}}_k^v$ is at most $|\tilde{\mathcal{P}}|$, and recall that an upper bound on $|\Gamma|$ is given in Proposition F.2.1. From the description of computing $\Pr(\mathcal{P}_k^v = \gamma)$ in Case 1b ($v$ has a single distributional node $w$) it follows that the total cost of handling this case (over all $v$ and $\gamma$) is $O(|\tilde{\mathcal{P}}| \cdot |\Gamma|)$. Therefore, the running time of the algorithm is bounded by $O(|\tilde{\mathcal{P}}| \cdot |\Gamma| \cdot F)$, where $F$ is any upper bound on the cost of computing $\Pr(\mathcal{P}_k^v = \gamma)$ in each of the cases other than 1b. Next, we compute such a bound $F$. 

43
Recall from the proof of Proposition F.4.1 that $|\gamma'|$ denotes the size of $\gamma'$, and that this size excludes the numerical specification for the operands. Let $m_\Gamma$ denote the maximal size (i.e., the number of nodes and symbols) of a c-formula in $\Gamma$, and let $N$ be the maximal operand given in the numerical specification. Note that $m_\Gamma$ depends only on the fixed $\hat{\gamma}$, and not on the numerical specification. Also, recall that $N \leq |\hat{\mathcal{P}}|$. We consider the different cases of computing $\Pr(\mathcal{P}_k^v = \gamma)$. In Case 1a ($v$ is childless), it follows immediately from the description that the execution cost is $O(m_\Gamma)$. In Case 1c, where $k = 1$, the execution time is dominated by the transformations described in Section F.4, and the proof of Proposition F.4.1 shows that the execution cost is then $O(Nm_\Gamma)$. In the case of Section F.5 ($k > 1$), we need to compute the sum of Proposition F.5.1. This requires evaluating the queries of $\Gamma$ over $\hat{\mathcal{P}}'$ (i.e., Case 1b or 1c) and applying an iteration over $Cord(\gamma)$. Thus, the execution cost is again upper bounded by $O(Nm_\Gamma)$. Finally, the case of Section F.6 ($v$ is distributional) is executed similarly to the previous cases. It follow that we can use the bound $F = O(Nm_\Gamma)$.
Appendix G

Sampling PXDBs: Proof of Correctness

In this section, we prove Theorem 5.6.2, namely, the correctness of the sampling algorithm \text{Sample(\mathcal{C})}. We start with some notation.

G.1 Notation and Conventions

We use $\mathcal{B}$ to denote the set $\{\text{true}, \text{false}\}$. Consider a vector $\vec{I} \in \mathcal{B}^m$. The $i$th element of $\vec{I}$ is denoted by $\vec{I}[i]$, where $1 \leq i \leq m$. For $0 \leq j \leq m$, the vector $\vec{I}[\leq j]$ is the prefix of $\vec{I}$ of length $j$. Note that if $j = 0$, then $\vec{I}[\leq j]$ is the empty vector.

We first give an alternative (yet equivalent) description of generating a random document from a p-document $\mathcal{P}$. Instead of randomly choosing children top-down (thus obtaining a random r-subtree $\mathcal{P}^\Sigma$), we use the following random process that generates a forest $\mathcal{P}^f$. We randomly choose a set of children for every distributional node $v$. For each unchosen child $w$ of $v$, the edge from $v$ to $w$ is removed. Finally, $\mathcal{P}^\Sigma$ is the subtree of $\mathcal{P}^f$ that contains root($\mathcal{P}$), and a document is obtained from $\mathcal{P}^\Sigma$ in the original way, namely, by removing the distributional nodes. Clearly, this process results in the same distribution over random documents as the original one.

Recall that given a p-document $\mathcal{P}$, the algorithm iterates over the set of pairs $\mathcal{W}(\mathcal{P}) =$
\{(v_1, W_1), \ldots, (v_m, W_m)\}. We use this representation also for describing the above process of generating a forest \(P^f\). In particular, the random variable \(\vec{P}\) is a vector of \(B^m\), such that \(\vec{P}[i] = \text{true}\) if and only if \(v_i\) chooses \(W_i\) as its set of children. So, every p-document \(\vec{P}\) is associated with two random variables: one is the above \(\vec{P}\), and the other is \(P\) that represents the random document generated by the above process (which is equivalent to the top-down process described earlier). The random variables \(\vec{P}[i] (1 \leq i \leq m)\) and \(\vec{P}_{\leq i} (0 \leq i \leq m)\) are obtained from \(\vec{P}\) as described above.

Consider the algorithm \texttt{Sample}(\mathcal{C}). The variable \(b_i\) does not get a value if the test of Line 5 is \texttt{true}. So, in this case, we define \(b_i = \text{true}\) if \(\vec{P}_{i-1}(v_i, W_i) = 1\), and \(b_i = \text{false}\) if \(\vec{P}_{i-1}(v_i, W_i) = 0\). The random variable \(\vec{b}\) is the vector in which the \(i\)th element is the value of \(b_i\) when the (random) execution of \texttt{Sample}(\mathcal{C})(\vec{P}) terminates. Note that \(\vec{b} = \vec{P}_m\), that is, \(\vec{b}\) corresponds to the forest \(P^f\) that is used for generating the result of the algorithm. Again as described above, the random variables \(\vec{b}[i] (1 \leq i \leq m)\) and \(\vec{b}_{\leq i} (0 \leq i \leq m)\) are obtained from \(\vec{b}\).

\section*{G.2 Analysis of a Specific Execution}

In this section, we consider a specific (random) execution of \texttt{Sample}(\mathcal{C})(\vec{P}). This execution generates the specific p-documents \(\vec{P}_0, \ldots, \vec{P}_m\) (where \(\vec{P}_0 = \vec{P}\)) and the numbers \(q_0, \ldots, q_m, p_1, \ldots, p_m\). Furthermore, the vector \(\vec{b}\) is instantiated (and is not considered here as a random variable).

We start with the following proposition that shows the correctness of \texttt{Normalize}. Its proof follows immediately from the definitions of a p-document (and its associated random variables) and the subroutine \texttt{Normalize}. Note that the event \(\vec{P}_{i-1}[i]\) is the same as \(\vec{P}_{i-1}[i] = \text{true}\).

\textbf{Proposition G.2.1} Consider a specific execution of \texttt{Sample}(\mathcal{C})(\vec{P}). For all \(\vec{c} \in B^m\) and \(i = 1, 2, \ldots, m\), the following hold.
\begin{itemize}
\item If $\mathcal{P}_{i-1}(v_i, W_i) > 0$ and $\mathcal{P}' = \text{Normalize}(\mathcal{P}_{i-1}, v_i \rightarrow W_i)$, then
\[ \Pr(\mathcal{P}' = \mathcal{c}) = \Pr(\mathcal{P}_{i-1} = \mathcal{c} | \mathcal{P}_{i-1}[i]). \]
\item If $\mathcal{P}_{i-1}(v_i, W_i) < 1$ and $\mathcal{P}' = \text{Normalize}(\mathcal{P}_{i-1}, v_i \not\rightarrow W_i)$, then
\[ \Pr(\mathcal{P}' = \mathcal{c}) = \Pr(\mathcal{P}_{i-1} = \mathcal{c} | \neg \mathcal{P}_{i-1}[i]). \]
\end{itemize}

The next lemma shows some basic properties of the values that are computed during the execution of the algorithm. In particular, it shows that there is no division by zero in Line 12.

**Lemma G.2.2** Consider a specific execution of $\text{Sample}(\mathcal{C})(\hat{\mathcal{P}})$. For all $\mathcal{c} \in \mathbb{B}^m$ and $0 \leq i \leq m$, the following hold.

1. If $i > 0$, then $p_i = \Pr(\mathcal{P}_{i-1}[i] | \mathcal{P}_{i-1} \models \mathcal{C}).$
2. $q_i = \Pr(\mathcal{P}_i \models \mathcal{C}) > 0.$

**Proof.** We prove the two parts simultaneously by induction on $i$. The basis of the induction (i.e., $i = 0$) is vacuously true for Part 1, and it holds for Part 2 because of Line 3 and the assumption (about the input) that $\hat{\mathcal{P}}$ is consistent with $\mathcal{C}$.

For the inductive step, let $i > 0$. We first prove Part 1. By the definition of $\hat{\mathcal{P}}$,
\[ \Pr(\mathcal{P}_{i-1}[i]) = \mathcal{P}_{i-1}(v_i, W_i). \]

If $\mathcal{P}_{i-1}(v_i, W_i) = 0$, the claim holds because Line 6 assigns 0 to $p_i$. So, suppose that $\Pr(\mathcal{P}_{i-1}[i]) > 0$. By this assumption and Part 2 of the induction hypothesis, we can apply Bayes' theorem to get the equation
\[ \Pr(\mathcal{P}_{i-1}[i] | \mathcal{P}_{i-1} \models \mathcal{C}) = \frac{\Pr(\mathcal{P}_{i-1} \models \mathcal{C} | \mathcal{P}_{i-1}[i]) \Pr(\mathcal{P}_{i-1}[i])}{\Pr(\mathcal{P}_{i-1} \models \mathcal{C})}. \]

\[ ^1 \text{There is no division by zero in either Line 19 or the subroutine Normalize because the potentially problematic cases are handled by the test of Line 5.} \]
If $\tilde{p}_{i-1}(v_i, W_i) = 1$, then $\Pr(\tilde{p}_{i-1} \models C | \tilde{p}_{i-1}[i]) = \Pr(\tilde{p}_{i-1} \models C)$. So, the claim holds because Line 6 assigns 1 to $p_i$.

Now, suppose that $0 < \tilde{p}_{i-1}(v_i, W_i) < 1$. Denote $\text{Normalize}(\tilde{p}_{i-1}, v_i \rightarrow W_i)$ by $\tilde{p}'$. By Proposition [G.2.1] and the fact that $\mathcal{P}$ and $\tilde{p}$ describe the same distribution over random documents,

$$\Pr(\tilde{p}_{i-1} \models C | \tilde{p}_{i-1}[i]) = \Pr(\tilde{p}' \models C).$$

Part 2 of the induction hypothesis implies that $q_{i-1} = \Pr(\tilde{p}_{i-1} \models C)$. Therefore, $p_i$ gets the correct value in Line 12, due to the assignments in Lines 11 and 12.

We now prove the inductive step for Part 2. First, we show that $q_{i-1} = \Pr(\tilde{p}_{i-1} \models C)$. If $\tilde{p}_{i-1}(v_i, W_i)$ is either 0 or 1, then the claim holds because of the induction hypothesis and Lines 7–8.

So, suppose that $0 < \tilde{p}_{i-1}(v_i, W_i) < 1$. If $b_i = \text{true}$, then Part 2 holds because of Lines 11, 15 and 16. Next, we assume that $b_i = \text{false}$. The first equality below follows from Proposition [G.2.1] and Line 18, the second is elementary and the third uses the equality $\Pr(\tilde{p}_{i-1}[i]) = \tilde{p}_{i-1}(v_i, W_i)$.

$$\Pr(\tilde{p}_{i-1} \models C | \tilde{p}_{i-1}[i]) = \frac{\Pr(\tilde{p}_{i-1} \models C \land \tilde{p}_{i-1}[i])}{\Pr(\tilde{p}_{i-1}[i])} = \frac{\Pr(\tilde{p}_{i-1} \models C) - \Pr(\tilde{p}_{i-1} \models C \land \tilde{p}_{i-1}[i])}{1 - \tilde{p}_{i-1}(v_i, W_i)}$$

The induction hypothesis states that $q_{i-1} = \Pr(\tilde{p}_{i-1} \models C)$. So, to conclude that Line 19 assigns $\Pr(\tilde{p}_{i} \models C)$ to $q_i$, we show the following.

Let $\tilde{p}' = \text{Normalize}(\tilde{p}_{i-1}, v_i \rightarrow W_i)$. The first equality below is elementary and the second follows from Proposition [G.2.1] and $\Pr(\tilde{p}_{i-1}[i]) = \tilde{p}_{i-1}(v_i, W_i)$.

$$\Pr(\tilde{p}_{i-1} \models C \land \tilde{p}_{i-1}[i]) = \Pr(\tilde{p}_{i-1}[i]) \cdot \Pr(\tilde{p}_{i-1} \models C | \tilde{p}_{i-1}[i]) = \tilde{p}_{i-1}(v_i, W_i) \cdot \Pr(\tilde{p}' \models C)$$

We end with the proof that $\Pr(\tilde{p}_{i} \models C) > 0$. If $\tilde{p}_{i-1}(v_i, W_i)$ is either 0 or 1, then
the claim holds because of the induction hypothesis and Lines 7–8. So, suppose that $0 < \tilde{P}_{i-1}(v_i, W_i) < 1$. We assume that $b_i = \text{true}$ (the proof for $b_i = \text{false}$ is similar). The first equality below is by Proposition G.2.1, Line 10 and Line 15; the second is by Bayes’ theorem.

$$
\Pr(P_i \models C) = \Pr(P_{i-1} \models C \mid \tilde{P}_{i-1}[i]) = \frac{\Pr(\tilde{P}_{i-1}[i] \mid P_{i-1} \models C) \cdot \Pr(P_{i-1} \models C)}{\Pr(\tilde{P}_{i-1}[i])}
$$

$\Pr(P_{i-1} \models C) > 0$ follows from Part 2 of the induction hypothesis. By Part 1 of that hypothesis, $p_i = \Pr(\tilde{P}_{i-1}[i] \mid P_{i-1} \models C)$, and $p_i > 0$ must hold because $b_i$ gets true in Line 13.

The next lemma shows that the prior probability distribution of $\tilde{P}_i$ is the same as the posterior distribution of $\tilde{P}$ given the first $i$ choices of the execution. Note that the first part of the lemma is needed to ensure that the second part is well defined.

**Lemma G.2.3** Consider a specific execution of Sample$(C)(\tilde{P})$. For all $\vec{c} \in B^m$ and $0 \leq i \leq m$, the following hold.

1. $\Pr(\tilde{P}_{\leq i} = \vec{b}_{\leq i}) > 0$.

2. $\Pr(\tilde{P}_i = \vec{c}) = \Pr(\tilde{P} = \vec{c} \mid \tilde{P}_{\leq i} = \vec{b}_{\leq i})$.

**Proof.** The proof is by induction on $i$. For $i = 0$, Parts 1 and 2 hold because two empty vectors are always equal and $\tilde{P}_0 = \tilde{P}$, respectively.

For the inductive step, let $i > 0$. Firstly, we prove Part 1. By the definition of conditional probability,

$$
\Pr(\tilde{P}_{\leq i} = \vec{b}_{\leq i}) = \Pr(\tilde{P}_{\leq i-1} = \vec{b}_{\leq i-1}) \cdot \Pr(\tilde{P}[i] = \vec{b}[i] \mid \tilde{P}_{\leq i-1} = \vec{b}_{\leq i-1})
$$

By Part 1 of the induction hypothesis, the first probability on the right side is positive. To show that the second is positive, we substitute $\vec{b}$ for $\vec{c}$ in Part 2 of the induction hypothesis,

$$
\Pr(\tilde{P}[i] = \vec{b}[i] \mid \tilde{P}_{\leq i-1} = \vec{b}_{\leq i-1}) = \Pr(\tilde{P}_{i-1}[i] = \vec{b}[i]).
$$
Due to the test of Line 5, if \( \vec{b}[i] = \text{true} \), then \( \tilde{P}_{i-1}(v_i, W_i) > 0 \); otherwise, \( \tilde{P}_{i-1}(v_i, W_i) < 1 \). Hence, the right side of the above equation is nonzero, because \( \Pr(\tilde{P}_{i-1}[i]) = \tilde{P}_{i-1}(v_i, W_i) \) and \( \Pr(\neg\tilde{P}_{i-1}[i]) = 1 - \tilde{P}_{i-1}(v_i, W_i) \).

To prove Part 2, we assume that \( b[i] = \text{true} \) (the proof for \( b[i] = \text{false} \) is similar). In this case, \( \tilde{P}_i = \text{Normalize}(\tilde{P}_{i-1}, v_i \rightarrow W_i) \) and, as shown above, \( \Pr(\tilde{P}_{i-1}[i]) > 0 \). The first equality below is by Proposition G.2.1, the third by Part 2 of the induction hypothesis, and the others are elementary.

\[
\Pr(\vec{b}[\leq i] = \vec{c}[\leq i]) = \Pr(\tilde{P}_{\leq i}[\leq i] = \vec{c} \cap \tilde{P}_{i-1}[i]) = \frac{\Pr(\tilde{P}_{\leq i}[\leq i] = \vec{c} \cap \tilde{P}_{i-1}[i])}{\Pr(\tilde{P}_{i-1}[i])} = \\
\frac{\Pr(\tilde{P} = \vec{c} \cap \tilde{P}[i] \mid \tilde{P}_{\leq i-1}[\leq i] = \vec{b}[\leq i-1])}{\Pr(\tilde{P}[i] \mid \tilde{P}_{\leq i-1}[\leq i] = \vec{b}[\leq i-1])} = \Pr(\tilde{P} = \vec{c} \mid \tilde{P}[i] \cap \tilde{P}_{\leq i-1}[\leq i] = \vec{b}[\leq i-1]) = \\
\Pr(\tilde{P} = \vec{c} \mid \tilde{P}[\leq i] = \vec{b}[\leq i])
\]

Note that in the second line of the above equation, we apply Part 2 of the induction hypothesis once to the numerator and a second time to the denominator, thereby getting the third line. These applications are correct because the hypothesis holds for all \( \vec{c} \in B^m \), hence Part 2 is valid for every event and not just equality to a specific \( \vec{c} \).

\[\square\]

### G.2.1 Proving Theorem 5.6.2

We need the following lemma to complete the proof that the algorithm \( \text{Sample}('C')(\tilde{P}) \) is correct.

**Lemma G.2.4** During an execution of \( \text{Sample}('C')(\tilde{P}) \), the following holds for all \( 0 \leq i \leq m \) and \( \vec{c} \in B^m \).

\[
\Pr(\vec{b}[\leq i] = \vec{c}[\leq i]) = \Pr(\tilde{P}[\leq i] = \vec{c}[\leq i] \mid \mathcal{P} \models 'C')
\]

**Proof.** To prove the lemma, we consider a fixed (yet arbitrary) vector \( \vec{c} \) of \( B^m \). Note that the equality \( \vec{b}[\leq i] = \vec{c}[\leq i] \) is satisfied by every execution of the algorithm, such that in the first \( i \) iterations, \( b_1, \ldots, b_i \) get the values \( \vec{c}[1], \ldots, \vec{c}[i] \), respectively.
The proof is by induction on $i$. The basis (i.e., $i = 0$) is trivial because two empty vectors are always equal, hence both probabilities are 1. For the inductive step, we assume that $i > 0$.

If $\Pr(\vec{b}_{\leq i-1} = \vec{c}_{\leq i-1}) = 0$, then by the induction hypothesis, $\Pr(\vec{P}_{\leq i-1} = \vec{c}_{\leq i-1} | \mathcal{P} \models \mathcal{C}) = 0$. Both probabilities remain 0 when $i - 1$ is replaced with $i$, because if no execution satisfies $\vec{b}_{\leq i-1} = \vec{c}_{\leq i-1}$, then the same is certainly true for $i$.

So, suppose that $\Pr(\vec{b}_{\leq i-1} = \vec{c}_{\leq i-1}) > 0$. By the definition of conditional probability,

$$\Pr(\vec{b}_{\leq i} = \vec{c}_{\leq i}) = \Pr(\vec{b}_{\leq i-1} = \vec{c}_{\leq i-1}) \cdot \Pr(\vec{b}[i] = \vec{c}[i] | \vec{b}_{\leq i-1} = \vec{c}_{\leq i-1}).$$  \hspace{1cm} (G.1)

By the induction hypothesis,

$$\Pr(\vec{b}_{\leq i-1} = \vec{c}_{\leq i-1}) = \Pr(\vec{P}_{\leq i-1} = \vec{c}_{\leq i-1} | \mathcal{P} \models \mathcal{C}).$$  \hspace{1cm} (G.2)

The second term in the right side of Equation (G.1) is the probability that the variable $b_i$ gets the same truth value as $\vec{c}[i]$ in an execution of $\text{Sample}(\mathcal{C})(\vec{P})$, such that $\vec{b}_{\leq i-1} = \vec{c}_{\leq i-1}$. So, we consider such an execution.

Firstly, we assume that $\vec{c}[i] = \text{true}$. Line 14 of the algorithm assigns true to $b_i$ with probability $p_i$. Hence, $p_i$ is equal to the second probability in the right side of Equation (G.1). The first equality below is by Part 1 of Lemma G.2.2 in the second equality, we use Part 2 of Lemma G.2.2 the third equality is by Lemma G.2.3 and the other two are elementary.

$$p_i = \Pr(\vec{P}_{i-1}[i] | \mathcal{P}_{i-1} \models \mathcal{C}) = \frac{\Pr(\vec{P}_{i-1}[i] \land \mathcal{P}_{i-1} \models \mathcal{C})}{\Pr(\mathcal{P}_{i-1} \models \mathcal{C})} = \frac{\Pr(\vec{P}[i] \land \mathcal{P} \models \mathcal{C} | \vec{P}_{\leq i-1} = \vec{c}_{\leq i-1})}{\Pr(\mathcal{P} \models \mathcal{C} | \vec{P}_{\leq i-1} = \vec{c}_{\leq i-1})} = \Pr(\vec{P}_{\leq i-1} = \vec{c}_{\leq i-1} \land \mathcal{P} \models \mathcal{C}).$$  \hspace{1cm} (G.3)

We should note the following. In Lemma G.2.3 $\vec{b}$ describes the outcome of a specific execution of the algorithm, but only the result of the first $i$ iterations is relevant to the
actual claim made in that lemma. When deriving the third equality above, Lemma $G.2.3$ is applied for $i - 1$ and the result of the first $i - 1$ iterations is now described by $\vec{c}_{[\leq i-1]}$. So, $\vec{c}_{[\leq i-1]}$ plays the role of $\vec{b}_{[\leq i-1]}$ when we apply that lemma here.

In the right side of Equation (G.1), we replace the first and second probabilities by using Equations (G.2) and (G.3), respectively. We conclude the following.

$$\Pr\left(\vec{b}_{[\leq i]} = \vec{c}_{[\leq i]} \right) =$$

$$= \Pr\left(\vec{h}_{\leq i-1} = \vec{c}_{[\leq i-1]} \mid \mathcal{P} \models C\right) \cdot \Pr\left(\vec{h}[i] \mid \vec{h}_{[\leq i-1]} = \vec{c}_{[\leq i-1]} \land \mathcal{P} \models C\right) =$$

$$= \Pr\left(\vec{h}[i] \land \vec{h}_{[\leq i-1]} = \vec{c}_{[\leq i-1]} \mid \mathcal{P} \models C\right)$$

Since $\vec{c}[i] = \text{true}$, the last probability above is equal to $\Pr(\vec{h}_{[\leq i]} = \vec{c}_{[\leq i]} \mid \mathcal{P} \models C)$, as claimed.

Now, suppose that $\vec{c}[i] = \text{false}$ and let $\vec{c}$ be obtained from $\vec{c}$ by replacing $\vec{c}[i]$ with $\text{true}$. The second equality below is by the induction hypothesis and the proof of the previous case.

$$\Pr(\vec{b}_{[\leq i]} = \vec{c}_{[\leq i]}) = \Pr\left(\vec{b}_{[\leq i-1]} = \vec{c}_{[\leq i-1]} \right) - \Pr\left(\vec{b}_{[\leq i]} = \vec{c}_{[\leq i]} \right) =$$

$$= \Pr\left(\vec{h}_{[\leq i-1]} = \vec{c}_{[\leq i-1]} \mid \mathcal{P} \models C\right) - \Pr\left(\vec{h}_{[\leq i]} = \vec{c}_{[\leq i]} \mid \mathcal{P} \models C\right) =$$

$$= \Pr(\vec{h}_{[\leq i]} = \vec{c}_{[\leq i]} \mid \mathcal{P} \models C)$$

Thus, the proof is complete. $\square$

The following is a corollary of Lemma $G.2.4$ when $i = m$. Hence, Theorem 5.6.2 is proven.

**Corollary G.2.5** *Upon termination of $\text{Sample}(\exists C)(\mathcal{P})$, the following holds for all $\vec{c} \in B^m$.*

$$\Pr(\vec{b} = \vec{c}) = \Pr(\vec{h} = \vec{c} \mid \mathcal{P} \models C)$$