Finding a Minimal Tree Pattern Under Neighborhood Constraints

Benny Kimelfeld
IBM Research–Almaden
San Jose, CA 95120, USA
kimelfeld@us.ibm.com

Yehoshua Sagiv*
Dept. of Computer Science
The Hebrew University
Jerusalem 91094, Israel
sagiv@cs.huji.ac.il

ABSTRACT

Tools that automatically generate queries are useful when schemas are hard to understand due to size or complexity. Usually, these tools find minimal tree patterns that contain a given set (or bag) of labels. The labels could be, for example, XML tags or relation names. The only restriction is that, in a tree pattern, adjacent labels must be among some specified pairs. A more expressive framework is developed here, where a schema is a mapping of each label to a collection of bags of labels. A tree pattern conforms to the schema if for all nodes v, the bag comprising the labels of the neighbors is contained in one of the bags to which the label of v is mapped. The problem at hand is to find a minimal tree pattern that conforms to the schema and contains a given bag of labels. This problem is NP-hard even when using the simplest conceivable language for describing schemas. In practice, however, the set of labels is small, so efficiency is realized by means of an algorithm that is fixed-parameter tractable (FPT). Two languages for specifying schemas are discussed. In the first, one expresses pairwise mutual exclusions between labels. Though W[1]-hardness (hence, unlikeliness of an FPT algorithm) is shown, an FPT algorithm is described for the case where the mutual exclusions form a circular-arc graph (e.g., disjoint cliques). The second language is that of regular expressions, and for that another FPT algorithm is described.

Categories and Subject Descriptors: H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval—*query formulation, search process*; H.2[Database Management]: Heterogeneous Databases, Miscellaneous

General Terms: Algorithms, Theory

Keywords: Query extraction, minimal tree pattern, graph search

1. INTRODUCTION

Without a close familiarity with the schema of a given database, manual query formulation can be prohibitively laborious and time consuming. In the case of schemas that are very large and complex (for example, SAP schemas that may have ten thousand or more relations [12]), it may become practically impossible. Various paradigms tackle this problem, such as tools for schema exploration and semi-automatic query formulation [15], schema-free (or flexible) queries [5, 16], and keyword search over structured data [3, 11, 13, 22]. A standard module in the realization of these paradigms is an algorithm that finds small tree patterns that connect a given set of items (where an item can be the name of a relation, the value of an attribute, an XML tag, an XML CDATA value, and so on). However, none of the existing implementations of this module take into account even very simple constraints on the patterns, such as: a person has one gender, the identifier of a person is not shared by others, and so on. More complex constraints are imposed by XML schemas, where regular expressions qualify the possible sets of sub-elements. When ignoring these constraints in practical schemas, a huge number of patterns either make no sense to the end user, or do not lead to actual results in the overall application.

As an example, we discuss keyword search over structured data (e.g. relational databases, RDF triples, XML documents, etc.). The typical approach to this problem considers abstractly a *data graph*, where nodes represent objects (e.g., tuples or XML elements), and edges represent pairwise associations between the objects (e.g., foreign keys or element nesting). The keywords phrased by the user match some of the objects, and search results are subtrees of the data graph that cover all those keywords. A dominant factor in the quality of a result is its size (or weight), since smaller subtrees represent closer associations between the keywords. The computational goal is to find these small subtrees. For that, there are two main approaches. The *graph* approach (e.g., [3, 8, 13]) extracts the subtrees directly from the data, while ignoring the schema and the particular patterns of those subtrees. The *pattern* approach [11, 17, 22, 25], which is investigated in this paper, deploys two steps. In the first step, a set of candidate patterns is generated; in the second step, search results are obtained by evaluating the patterns as queries over the data graph (e.g., finding subtrees that are isomorphic to those patterns).

The pattern approach has several important advantages. The obvious one is the ability to use existing systems (e.g., RDBMS) that already have highly optimized query engines. Thus, the pattern approach can be implemented rather easily over such systems. Furthermore, as enunciated in [19], keyword search over data graphs is carried out across two dimensions. In one dimension, the user explores the different semantic connections (where each one is represented by a tree pattern). In the second dimension, the user looks for specific answers for each tree pattern that is of interest. The pattern approach facilitates the search across those two dimensions as follows. First, it can display early on results of diversified seman-

*Work supported by The German-Israeli Foundation for Scientific Research & Development (Grant 973–150.6/2007).

PODS'11, June 13–15, 2011, Athens, Greece.

tic connections (i.e., patterns), rather than inundate the user with many answers of just one tree pattern. Second, with the proper interface, it is fairly easy for a user to refine her query by phrasing conditions on the patterns that have already been displayed to her (e.g., filtering out some labels, adding selections, etc.) [1, 15].

Efficiency in producing a set of candidate patterns is crucial, since in all of the above applications, the patterns are generated in real time in response to a user query. The effectiveness of existing algorithms has been exhibited on tiny schemas (usually designed just for experiments). We believe that those algorithms are not likely to scale up to the sizes of real-world schemas. Specifically, most of the algorithms (e.g., [17, 18, 21, 25]) follow DISCOVER [11] by essentially producing all the possible *partial patterns* (i.e., trees that do not necessarily include the keywords) up to some size, or until sufficiently many *complete patterns* are produced. The complete patterns include all of the keywords (of the query), and they are evaluated against the database. This generation of the complete patterns is grossly inefficient, because the incremental construction can exponentially blowup the number of partial patterns. Moreover, many of the generated complete patterns may make no sense (and in particular produce empty results), simply because they violate basic constraints imposed on (or just happened to be satisfied by) the database.

To the best of our knowledge, the only algorithm that generates patterns with guaranteed efficiency is that of Talukdar et al. [22]. To illustrate this approach, consider Figure 1. The left side of the figure shows a *schema graph*. That graph has a node for each label (i.e., entity type). There is an edge between two labels if entities of those types could be connected by an edge in the underlying data graph. Given a query (which is a set of labels), the patterns generated by Talukdar et al. [22] are non-redundant subtrees (of the schema graph) that contain the labels of the query.[1] Now, suppose that we want to find patterns that connect `female` and `male` (given the schema graph of Figure 1). Intuitively, we would like to get the pattern in the bottom row at the right side of the figure. That pattern connects two persons, such that one is a male and the other is a female who is the wife of the male. But this pattern is not a subtree of the schema graph, because the label `person` is repeated. The only non-redundant subtree (of the schema graph) that contains the labels `female` and `male` is the one in the top row, namely, a person that is both a male and a female. Clearly, this pattern is useless; that is, it will never match actual results. Useful patterns necessarily include more than one person (i.e., a male and a female). But the algorithm of Talukdar et al. cannot find them, because it cannot generate patterns with repeated labels. Conceivably, this problem could be fixed by introducing repetitions in the schema graph (e.g., `person1`, `person2`, and so on). But then, we would get new useless patterns. For example, the pattern in the middle row refers to a male that is the *wife* of a female.

It should be noted that Talukdar et al. [22] employ the algorithm of Kimelfeld and Sagiv [13] for finding the k minimal non-redundant subtrees. In fact, the algorithm of [13] was designed for the aforementioned graph approach. As shown by the above example, using the algorithm of [13] for the pattern approach may provide an unsatisfactory solution. The algorithms we present in this paper are substantially different from those of [13].

To summarize thus far, we need an algorithm that can efficiently generate the patterns that will actually be evaluated against the database (in particular, we should not waste time on constructing partial patterns that are not even going to be evaluated). In addition, the algorithm should be able to generate patterns with repeated

[1]A subtree is *non-redundant* if it has no proper subtree that also contains all the labels of the query.

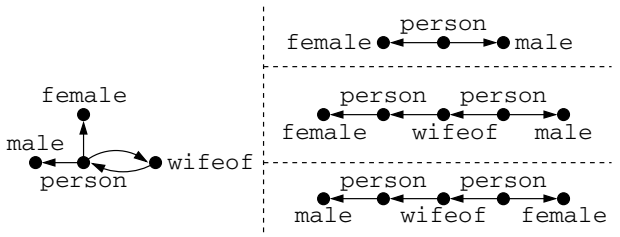

Figure 1: Possible relationships (left), and three patterns connecting `female` and `male` (right)

labels (or else we may miss important answers). And finally, it should avoid creating useless patterns that violate basic constraints that hold in the database.

Designing an algorithm that avoids useless patterns is not an easy task and it depends on the type of constraints that are used. In this paper, we do it for *neighborhood* constraints. As an example, such a constraint can state that a node labeled with `person` can have either a `male` neighbor or a `female` neighbor, but not both; furthermore, an incoming `wifeof` neighbor is allowed only if an outgoing `female` neighbor exists. Formally, a constraint[2] maps a label to a (possibly infinite) set comprising finite bags (i.e., multisets) of labels. A schema S is a collection of constraints $S(\sigma)$—one for each label σ, and it also specifies weights on the nodes and edges. For a node v of a pattern, where v is labeled with σ, the neighboring labels of v (including their multiplicities) must form a subbag of a bag in $S(\sigma)$. Given a bag Λ of labels derived from a user's query, the problem is to find a minimal Λ-*pattern*, which is a tree t such that the labels of t form a superbag of Λ, and every node of t satisfies the constraint for its label. In this paper, we give algorithms for finding minimal Λ-patterns. In practice, we would like to find the top-k Λ-patterns (rather than just the minimal one). We defer to future work the development of an algorithm for the task of finding the top-k Λ-patterns. However, we discuss this task in Section 6, and in particular show that even the definition of the top-k Λ-patterns is not obvious.

In practice, neighborhood constraints can be derived in several ways. A schema of some sort (if available) is an obvious source. Additionally, an analysis of the data itself may provide some constraints; for example, the data may show that a wife is always a female. Finally, in keyword search, the schema is typically augmented with keywords given by the user [11]. The connections between the keywords and the schema's entities may indicate some neighborhood constraints. For example, the keywords `IBM` and `SAP` are always connected to `company` (at most once), and no `company` is connected by edges to both `IBM` and `SAP`.

Naturally, the complexity of our problem depends on the language used for expressing the neighborhood constraints. However, this problem is NP-hard even in the case of the simplest conceivable languages (e.g., each $S(\sigma)$ is a finite set that is specified explicitly). But that hardness builds on the assumption that the given bag Λ of labels is large, whereas in practice the size of Λ is similar to that of the user's query; a user's query is typically very small, and in particular, significantly smaller than the schema. Hence, we focus on *parameterized complexity* [6, 9] with the size of Λ as the parameter. A problem is deemed tractable if it is *Fixed-Parameter Tractable* (*FPT*), which means that the running time is polynomial, except that the parameter can have an arbitrarily large effect on the constant of the polynomial (but not on the degree).

[2]As stated, this constraint is undirected, but there are also directed ones.

In Section 3, we present a generic algorithm that reduces the problem of finding a minimal Λ-pattern to a generalization of minimal set cover (which is defined in that section). The algorithm is generic in the sense that it does not depend on any particular language for specifying constraints. An immediate result of the algorithm is that if the size of Λ is fixed, then we can find a minimal Λ-pattern in polynomial time under a reasonable assumption about the constraint language (i.e., one can efficiently test whether a fixed-size bag is contained in some element of $S(\sigma)$, which is specified by the language). But this result means that $|\Lambda|$ affects the degree of the polynomial, rather than just the constant. More importantly, Section 4 shows that this reduction leads to FPT algorithms for two practical constraint languages.

In the first language, constraints are specified in terms of pairwise mutual exclusions between labels, such as "if `male` then not `female` and vice-versa." The formal specification of these constraints is by means of an undirected *mux* graph, where edges represent mutual exclusions. Though extremely simple, our problem is unlikely to be FPT under this language—we show that it is $W[1]$-hard [6, 9]. Nonetheless, we give an FPT algorithm for the important case where the mux graph consists of pairwise-disjoint cliques. We then generalize this FPT algorithm to interval graphs, and even to circular-arc graphs with multiplicity bounds (e.g., at most one `male` neighbor and at most two `parent` neighbors). Based on our algorithm for mux graphs, we present an FPT algorithm for the second language, namely, regular expressions. This language is essentially the core of DTDs,[3] where the content of an element having a specified tag (label) is determined by a regular expression over the possible sub-elements.

In summary, algorithms and systems that find minimal or even top-k patterns are abundant, but to the best of our knowledge, none of them has both nontrivial efficiency guarantees and support for nontrivial (and practical) constraints. This paper presents the first step towards top-k algorithms with these qualities by showing how to efficiently find a minimal pattern under practical neighborhood constraints. Interestingly, it is nontrivial to even define the meaning of *top-k patterns*, and we discuss this issue in Section 6. The problem of actually finding the top-k patterns is left for future work. We believe that our algorithms are going to be a primary building block when developing a solution to the top-k problem.

For clarity of presentation, we start with a model (Section 2) and algorithms (Sections 3 and 4) for undirected Λ-patterns. The directed case (e.g., Figure 1) is more involved, and is considered in Section 5. Finally, in Section 6, we discuss how to define the problem of finding the top-k Λ-patterns.

2. FORMAL SETTING

In this section, we describe our framework and the problem we study. We start with some preliminary definitions.

2.1 Bags, Labels, and Regular Expressions

Recall that a *bag* (or *multiset*) is a pair $b = (X, \mu_b)$, where X is a set and $\mu_b : X \to \mathbb{N}$ is a *multiplicity function* that maps every element $x \in X$ to its multiplicity $\mu_b(x)$ (which is a positive integer). Throughout the paper, we implicitly assume that the multiplicity function of a bag b is μ_b. By a slight abuse of notation, we may assume that μ_b is also defined on an element $z \notin X$, and then $\mu_b(z) = 0$. To distinguish a bag from a set, we use double braces instead of braces; for example, $\{0, 1, 1\}$ is a set of size 2 (and is equal to $\{0, 1\}$), whereas $b = \{\!\{0, 1, 1\}\!\}$ is a bag of size 3 with $\mu_b(0) = 1$ and $\mu_b(1) = 2$. The operator $\uplus$ denotes bag union. Recall that containment has a special meaning when applied to bags; specifically, given two bags $b = (X, \mu_b)$ and $b' = (X', \mu_{b'})$, we say that b is a *subbag* of b', denoted by $b \subseteq b'$, if $X \subseteq X'$ and $\mu_b(x) \leq \mu_{b'}(x)$ for all $x \in X$.

We assume an infinite set $\mathbf{\Sigma}$ of *labels*. The main players in this work are bags of labels. By $\mathcal{F}_{\mathbf{\Sigma}}$ we denote the set of all the finite bags of labels. For a subset B of $\mathcal{F}_{\mathbf{\Sigma}}$, we define the *containment closure* of B, denoted by $[\![B]\!]$, as the set:

$$[\![B]\!] \stackrel{\text{def}}{=} \bigcup_{b \in B} \{b' \in \mathcal{F}_{\mathbf{\Sigma}} \mid b' \subseteq b\}\,.$$

That is, $[\![B]\!]$ is the set consisting of all the bags of B, and all the subbags of the bags of B.

We use regular expressions over $\mathbf{\Sigma}$. Specifically, regular expressions are defined by the language

$$e := \quad \sigma \mid \epsilon \mid e* \mid e? \mid e\,e \mid e|e$$

where $\sigma \in \mathbf{\Sigma}$, and ϵ is the empty string. A regular expression e defines a language $\mathcal{L}(e)$, namely, the set of strings over $\mathbf{\Sigma}$ that match the expression e. We define the *bag language* $\mathcal{L}^{\mathrm{b}}(e)$ of e in the standard way (e.g., as in [2]), namely, $\mathcal{L}^{\mathrm{b}}(e)$ contains all the bags $b \in \mathcal{F}_{\mathbf{\Sigma}}$, such that there exists a string $x \in \mathcal{L}(e)$ where every label has the same multiplicity in x and b (that is, $b \in \mathcal{L}^{\mathrm{b}}(e)$ if b can be ordered to formulate a word in $\mathcal{L}(e)$).

2.2 Graphs, Schemas and Patterns

In this paper, we consider finite graphs with labeled nodes. To simplify the presentation, we focus on undirected graphs up to Section 5, where we extend our results to directed graphs. The set of nodes of the graph g is denoted by $\mathsf{V}(g)$, and the set of edges of g is denoted by $\mathsf{E}(g)$. Note that an edge of $\mathsf{E}(g)$ is a set $\{u, v\} \subseteq \mathsf{V}(g)$. For a node $v \in \mathsf{V}(g)$, the label of v is denoted by $\lambda^g(v)$. If U is a subset of $\mathsf{V}(g)$, then $\lambda^g(U)$ denotes the bag that is obtained from U by replacing each node u with its label $\lambda^g(u)$ (that is, the multiplicity of each label σ is $|\{u \in U \mid \lambda^g(u) = \sigma\}|$). Note that $\lambda^g(v) \in \mathbf{\Sigma}$ and $\lambda^g(U) \in \mathcal{F}_{\mathbf{\Sigma}}$. If all the labels of the nodes of g belong to a subset L of $\mathbf{\Sigma}$, then we say that g is a graph *over* L. For a node v of g, the set of neighbors of v is denoted by $\mathsf{nbr}^g(v)$. Usually, the graph g is clear form the context, and then we may write just $\lambda(v)$, $\lambda(U)$ and $\mathsf{nbr}(v)$ instead of $\lambda^g(v)$, $\lambda^g(U)$ and $\mathsf{nbr}^g(v)$, respectively. Also, a node labeled with σ is called a *σ-node*. A *tree* is a connected and acyclic graph.

A *graph schema* (or just *schema* for short) identifies a set of valid graphs by imposing restrictions thereon. In this paper, we consider restrictions that are applied to neighborhoods of nodes having a specified label; that is, a schema qualifies, for each label σ, the bags of labels that a σ-node can have for its neighbors. A more formal definition is the following. An *lb-constraint* (where "lb" stands for "label bags") is a nonempty[4] (and possibly infinite) subset of $\mathcal{F}_{\mathbf{\Sigma}}$. A schema S is associated with a finite set of labels, denoted by $dom(S)$, and it maps every label $\sigma \in dom(S)$ to an lb-constraint $S(\sigma)$. A graph g *conforms to* S, denoted by $g \models S$, if for all nodes $v \in \mathsf{V}(g)$ it holds that $\lambda(v) \in dom(S)$ and $\lambda(\mathsf{nbr}(v)) \in S(\lambda(v))$.

For a schema S, we denote by $[\![S]\!]$ the schema that is obtained from S by replacing every $S(\sigma)$ (where $\sigma \in dom(S)$) with its containment closure $[\![S(\sigma)]\!]$.

DEFINITION 2.1. Let S be a schema, and let Λ be a bag in $\mathcal{F}_{\mathbf{\Sigma}}$. A Λ-*pattern* (*under* S) is a tree t, such that $\Lambda \subseteq \lambda(\mathsf{V}(t))$ and $t \models [\![S]\!]$. □

[3] Actually, in DTDs, regular expressions are restricted to be one-unambiguous, but we do not require that.

[4] An lb-constraint may comprise just the empty bag.

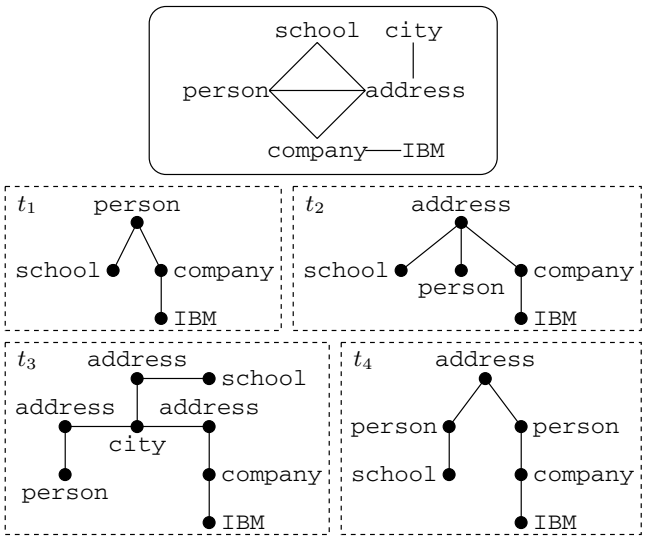

Figure 2: A neighboring relationship (top) and four trees

Note that for a Λ-pattern to exist, it is necessary (yet not sufficient) that every label in Λ belongs to $dom(S)$. Also note that in a Λ-pattern, each label σ of Λ occurs at least $\mu_\Lambda(\sigma)$ times.

EXAMPLE 2.2. Let S_1 be the schema such that $dom(S_1)$ consists of the six labels that appear at the top of Figure 2 (i.e., `school`, `person`, `IBM` and so on). In addition, for every label σ, the set $S_1(\sigma)$ comprises all the bags that have only neighbors of σ (in the graph at the top of Figure 2). For example, $S_1(\texttt{person})$ comprises all the bags that contain `school`, `company`, and `address`, each with some multiplicity.

Let Λ be the bag $\{\!\{\texttt{IBM}, \texttt{person}, \texttt{school}\}\!\}$. The four patterns t_1, t_2, t_3 and t_4 in Figure 2 are all Λ-patterns. The pattern t_1 corresponds to a person that is associated with both a school and a company (and that company is "IBM"). It may very well be the case that in the underlying data, a person that attends a school does not work in a company, and vice versa. Hence, the Λ-pattern t_1 has no occurrences in the data (that is, if t_1 is viewed as a query, then it has no results). As another example, the pattern t_2 has the same address for a school, a person, and a company; in most conceivable databases, there are no occurrences of t_2. In order to avoid such invalid patterns, we would like our schema to be more expressive. Hence, we define the schema S_2 to have the same domain as S_1, but now $S_2(\sigma)$ is $\mathcal{L}^{\text{b}}(e_\sigma)$, where:

$$\begin{aligned}
e_{\texttt{person}} &= (\texttt{school?} \mid \texttt{company*})\ \texttt{address}\\
e_{\texttt{school}} &= \texttt{person*}\ \texttt{address}\\
e_{\texttt{company}} &= \texttt{person*}\ \texttt{address}\ \texttt{IBM?}\\
e_{\texttt{address}} &= (\texttt{school?} \mid \texttt{person*} \mid \texttt{company?})\ \texttt{city}\\
e_{\texttt{city}} &= \texttt{address*}\\
e_{\texttt{IBM}} &= \texttt{company}
\end{aligned}$$

Neither t_1 nor t_2 conforms to $[\![S_2]\!]$; hence, they are not Λ-patterns under S_2. To see that $t_1 \not\models [\![S_2]\!]$, note that $\lambda(\mathsf{nbr}(v_p))$, where v_p is the `person`-node of t_1, does not belong to $[\![\mathcal{L}^{\text{b}}(e_{\texttt{person}})]\!]$. Similarly, $t_2 \not\models [\![S_2]\!]$ since the labels of the neighbors of its `address`-node do not form a bag in the set $[\![\mathcal{L}^{\text{b}}(e_{\texttt{address}})]\!]$.

It is easy to verify that both t_3 and t_4 are Λ-patterns under S_2, since both contain Λ and both conform to $[\![S_2]\!]$. Let Λ' be the same as Λ, except that `person` has multiplicity 2; that is, $\Lambda' = \{\!\{\texttt{IBM}, \texttt{person}, \texttt{person}, \texttt{school}\}\!\}$. Then t_4 is a Λ'-pattern, and t_3 is not, since t_4 has (at least) two `person`-nodes, and t_3 has only one `person`-node. □

2.3 Weighted Schemas and Minimal Patterns

When extracting patterns from schemas, some labels may be preferred over others, and some relationships could be viewed as stronger than others. For example, the label `city` may be deemed more important than `road`. Similarly, the `person-address` relationship could be viewed as more significant than `person-company`. To support such preferences, we augment a schema with weights for labels and edges. Formally, a *weight function* for a schema S is a mapping $W : dom(S) \cup (dom(S) \times dom(S)) \to [0, \infty)$ that is symmetric over $dom(S) \times dom(S)$ (which means that $W(\sigma, \tau) = W(\tau, \sigma)$ for all σ and τ in $dom(S)$).

A weight function W for S is naturally extended to any graph g over $dom(S)$ by defining $W(g)$ as follows.

$$W(g) \stackrel{\text{def}}{=} \sum_{v \in \mathsf{V}(g)} W(\lambda(v)) + \sum_{\{u,v\} \in \mathsf{E}(g)} W(\lambda(u), \lambda(v)) \qquad (1)$$

A *weighted schema* comprises a schema S and a weight function W for S, and is denoted by S^W.

We follow the convention that a lower weight means a higher preference. Thus, the goal is to find a minimal-weight pattern. Formally, we investigate the MINPATTERN problem, where the input consists of a weighted schema S^W and a bag Λ of labels, and the goal is to find a Λ-pattern t, such that $W(t) \leq W(t')$ for all Λ-patterns t'.

EXAMPLE 2.3. Consider S_2^W that is obtained from the schema S_2 of Example 2.2, and the weight function W, such that for all labels σ and τ it holds that $W(\sigma) = 1$ and $W(\sigma, \tau) = 0$ (hence, for a graph g we have $W(g) = |\mathsf{V}(g)|$). As in Example 2.2, let Λ be the bag $\{\!\{\texttt{IBM}, \texttt{person}, \texttt{school}\}\!\}$. Recall from Example 2.2 that, among the four trees of Figure 2, only t_3 and t_4 are Λ-patterns under S_2^W (since the other two do not conform to $[\![S_2]\!]$). Since $W(t_3) = 8$ and $W(t_4) = 6$, we get that t_3 is not a minimal pattern. It can be shown that t_4 is actually a minimal Λ-pattern under S_2^W.

Now, let W' be the weight function that is similar to W, except that $W'(\texttt{person}, \texttt{address}) = 5$. Then, we have that $W'(t_3) = 13$ and $W'(t_4) = 16$, and hence, t_4 is not a minimal Λ-pattern under $S_2^{W'}$. □

2.4 Complexity Basics

Recall that a schema S maps every label of $dom(S)$ to an lb-constraint (i.e., a nonempty and possibly infinite set of finite bags of labels). As expected, the complexity of MINPATTERN depends on how the lb-constraints are specified. An *lb-specification* is a finite object (e.g., a formal expression, a graph, etc.) that compactly represents an lb-constraint. Given an lb-specification s, we use $\mathcal{F}_{\mathbf{\Sigma}}(s)$ to denote the subset of $\mathcal{F}_{\mathbf{\Sigma}}$ that is represented by s.

In Section 4, we study two types of lb-specifications; one allows to express *mutual exclusions* among labels, and the other employs regular expressions. For now, consider an extremely restricted type, where an lb-specification is simply a bag $s \in \mathcal{F}_{\mathbf{\Sigma}}$ with $\mathcal{F}_{\mathbf{\Sigma}}(s) = \{s\}$. One can easily show (e.g., by a reduction from minimal set cover) that even for this restricted type of lb-specifications, the problem MINPATTERN is already NP-hard (moreover, approximation within any constant factor is also NP-hard).

However, in practical scenarios, the input bag Λ corresponds to a user query, and so, is typically very small. Thus, our main yardstick for efficiency is *fixed-parameter tractability* [6, 9], where $|\Lambda|$ is the parameter. Formally, an algorithm for solving MINPATTERN is

Fixed-Parameter Tractable (abbr. FPT) if its running time is bounded by a function of the form $f(|\Lambda|) \cdot p(|S^W|)$, where $f(k)$ is a computable function (e.g., 2^k), $p(m)$ is a polynomial, and $|S^W|$ is the size of the representation of S^W. (We usually do not formally define $|S^W|$, but rather assume a simple encoding.) A weaker yardstick for efficiency, in the spirit of *data complexity* [24], is *polynomial time* under the assumption that $|\Lambda|$ is *fixed*.

Later, we will show that MINPATTERN is FPT for types of lb-specifications that are far more expressive than the restricted one mentioned earlier. Our first step is to present a generic algorithm that is independent of the language used for lb-specifications.

3. A GENERIC ALGORITHM

We now describe a generic algorithm for solving MINPATTERN. The running time depends on the language for lb-specifications that is employed, but the algorithm itself does not use any specific details of that language. This algorithm reduces MINPATTERN to a generalization of the set-cover problem, which is defined next.

3.1 Labeled Bag Cover

Minimum labeled bag cover is a generalization of the minimum (weighted) set-cover problem. The former differs from the latter in three ways. First, labeled bags are used instead of sets. Second, a labeled bag can be used more than once in a cover. Third, the bag of labels of a cover has to be contained in some bag of a given lb-constraint. Next, we give the formal definition.

Let $b_1 = (X_1, \mu_1)$ and $b_2 = (X_2, \mu_2)$ be two bags. Recall that $b_1 \uplus b_2$ is the *multiset sum* (or bag union) of b_1 and b_2. That is, $b_1 \uplus b_2$ is the bag $b = (X, \mu)$, such that $X = X_1 \cup X_2$ and $\mu(x) = \mu_1(x) + \mu_2(x)$ for each $x \in X$ (recall that $\mu_i(x) = 0$ if $x \notin X_i$).

The minimum labeled bag-cover problem, which we denote by MINLBCOVER, is the following. The input consists of a bag Γ, an lb-constraint $\mathcal{B}$, and a set $\mathcal{C}$ of triples $\Delta = (b, \tau, w)$, where b is a bag, $\tau \in \mathbf{\Sigma}$ is the *label* of Δ, and $w \in [0, \infty)$ is the *weight* of Δ. A *legal cover* (*of* Γ *by* $\mathcal{C}$) is a bag $\{\!\{\Delta_1, \ldots, \Delta_n\}\!\}$, where each Δ_i is a triple (b_i, τ_i, w_i) in $\mathcal{C}$, such that $\Gamma \subseteq \biguplus_{i=1}^n b_i$ and $\{\!\{\tau_1, \ldots, \tau_n\}\!\} \in [\![\mathcal{B}]\!]$; the *weight* of the legal cover is the sum $\sum_{i=1}^n w_i$. As in the ordinary set-cover problem, the goal is to find a legal cover that has a minimal weight.

3.2 The Algorithm

Our algorithm, called FindMinPattern, reduces MINPATTERN to MINLBCOVER. It operates in the spirit of existing algorithms for finding Steiner trees [7, 14]. We first describe some notation and data structures.

The algorithm FindMinPattern gets as input a weighted schema S^W and a bag Λ of labels, and returns a Λ-pattern that has a minimal weight. In the remainder of this section, we fix the input S^W and Λ.

We use the following notation. Let $\mathcal{B}$ be an lb-constraint, and let $\varsigma \in \mathbf{\Sigma}$ be a label. By $\mathcal{B}_{-\varsigma}$ we denote the set that comprises all the bags $b \in \mathcal{F}_{\mathbf{\Sigma}}$, such that $b \uplus \{\!\{\varsigma\}\!\} \in \mathcal{B}$. We use a special label $\star$, such that $\star \notin dom(S)$ and $\star \notin \Lambda$. The label $\star$ has the following special role. For all lb-constraints $\mathcal{B}$, it holds that $\mathcal{B}_{-\star}$ is equal to $\mathcal{B}$. We say that $\mathcal{B}$ *mentions* ς if ς belongs to some bag of $\mathcal{B}$. Note that $\mathcal{B}_{-\varsigma}$ is nonempty (hence, an lb-constraint) if and only if $\varsigma = \star$ or $\mathcal{B}$ mentions ς. Also note that $[\![\mathcal{B}_{-\varsigma}]\!] \subseteq [\![\mathcal{B}]\!]$.

EXAMPLE 3.1. Consider the lb-constraint $\mathcal{B} = \mathcal{L}^{\mathrm{b}}(e_{\texttt{person}})$ of Example 2.2. If ς is the label `company`, then $\mathcal{B}_{-\varsigma}$ is the lb-constraint $\mathcal{L}^{\mathrm{b}}(\texttt{company}^* \texttt{address})$. If ς is the label `school`, then $\mathcal{B}_{-\varsigma}$ contains a single bag: $\{\!\{\texttt{address}\}\!\}$. □

The algorithm uses two data structures. One is an array $\mathcal{T}$. An index of $\mathcal{T}$ is a triple $\langle \sigma, b, \varsigma \rangle$, such that $\sigma \in dom(S)$, b is a subbag of Λ, and ς is either $\star$ or a label mentioned by $S(\sigma)$. Throughout the execution, $\mathcal{T}[\langle \sigma, b, \varsigma \rangle]$ is either $\perp$ (null), or a b-pattern t that has a node v, such that $\lambda(v) = \sigma$ and $\lambda(\mathsf{nbr}(v)) \in [\![S(\sigma)_{-\varsigma}]\!]$. In other words, node v of t has the following properties. First, v is a σ-node. Second, if $\varsigma = \star$, then v satisfies the lb-constraint $[\![S(\sigma)]\!]$ (i.e., $\lambda(\mathsf{nbr}(v)) \in [\![S(\sigma)]\!]$); otherwise (i.e., $\varsigma \neq \star$), v will continue to satisfy $[\![S(\sigma)]\!]$ even if we add a new ς-node as a neighbor of v.

The b-patterns that are stored in $\mathcal{T}$ are created by increasing weight (more precisely, a tree is created from smaller ones). For that we use the priority queue $\mathcal{Q}$ that stores the indexes of $\mathcal{T}$, namely, triples of the form $\langle \sigma, b, \varsigma \rangle$ as described above. The priority of $\langle \sigma, b, \varsigma \rangle$ in $\mathcal{Q}$ is determined by the weight of the b-pattern stored in $\mathcal{T}[\langle \sigma, b, \varsigma \rangle]$ (i.e., $W(\mathcal{T}[\langle \sigma, b, \varsigma \rangle])$). A higher weight means a lower priority. We define $W(\perp) = \infty$, so a triple $\langle \sigma, b, \varsigma \rangle$ has the lowest priority if $\mathcal{T}[\langle \sigma, b, \varsigma \rangle] = \perp$.

During the execution, when $\langle \sigma, b, \varsigma \rangle$ is removed from the top of $\mathcal{Q}$, it is the case that $\mathcal{T}[\langle \sigma, b, \varsigma \rangle]$ has a minimal weight among all the b-patterns having a node v with the above property (i.e., $\lambda(v) = \sigma$ and $\lambda(\mathsf{nbr}(v)) \in [\![S(\sigma)_{-\varsigma}]\!]$), unless no such b-patterns exist and then $\mathcal{T}[\langle \sigma, b, \varsigma \rangle] = \perp$.

The pseudo code of the algorithm FindMinPattern is given in Figure 3. The main procedure is shown at the top, and there are also three subroutines. We first describe the subroutine Initialize() that is called in line 1 of the main procedure. In that subroutine, line 1 initializes $\mathcal{Q}$ to the empty priority queue. The loop of line 2 iterates over all triples $\langle \sigma, b, \varsigma \rangle$ that could be indexes of $\mathcal{T}$. Line 3 checks that the triple $\langle \sigma, b, \varsigma \rangle$ is indeed a valid index, as defined earlier. In lines 4–6, $\mathcal{T}[\langle \sigma, b, \varsigma \rangle]$ is initialized to $\perp$, unless b is the singleton $\{\!\{\sigma\}\!\}$ and then $\mathcal{T}[\langle \sigma, b, \varsigma \rangle]$ is set to a tree that consists of a single node labeled with σ. Line 7 inserts into $\mathcal{Q}$ the triple $\langle \sigma, b, \varsigma \rangle$ with the priority $W(\mathcal{T}[\langle \sigma, b, \varsigma \rangle])$.

After initialization, the main procedure executes the loop of line 2. Line 3 removes the top triple $\langle \sigma', b, \sigma \rangle$ from $\mathcal{Q}$. Next, three cases are considered. First, in lines 4–5, the algorithm terminates in failure if $\mathcal{T}[\langle \sigma', b, \sigma \rangle] = \perp$. Namely, a Λ-pattern does not exist in this case. The second case is when $b = \Lambda$ and $\mathcal{T}[\langle \sigma', b, \sigma \rangle] \neq \perp$. The former is the test of line 6 and the latter must hold if that line is reached. In this case, line 7 returns $\mathcal{T}[\langle \sigma', b, \sigma \rangle]$ and the algorithm terminates. The third case is when line 8 is reached (which happens only if $\mathcal{T}[\langle \sigma', b, \sigma \rangle] \neq \perp$ and $b \neq \Lambda$) and the test of that line is true (i.e., $\sigma \neq \star$). In this case, we need to update all the entries of $\mathcal{T}$ that could potentially incorporate $\mathcal{T}[\langle \sigma', b, \sigma \rangle]$ as a subtree. This is done in lines 9–10 by calling the subroutine Update(σ, ς) for every label ς, such that for some $\hat{b}$ the triple $\langle \sigma, \hat{b}, \varsigma \rangle$ is an index of $\mathcal{T}$.

For every subbag b of Λ, the subroutine Update(σ, ς) tries to find a new (and better) b-pattern t for $\mathcal{T}[\langle \sigma, b, \varsigma \rangle]$. The main idea is to create a new σ-node v and connect it to some existing trees $t_1, \ldots, t_m$ as follows. Each tree t_i should have a τ_i-node u_i, such that $\lambda(\mathsf{nbr}^{t_i}(u_i)) \in [\![S(\tau_i)_{-\sigma}]\!]$. Thus, we can connect v to every u_i without causing the u_i to violate conformity to the schema S. Of course, we also need to verify that, in the new tree t, node v satisfies $\lambda(\mathsf{nbr}^t(v)) \in [\![S(\sigma)_{-\varsigma}]\!]$ and $b \subseteq \lambda(\mathsf{V}(t))$.

The search for t is done by a reduction to the following instance I of MINLBCOVER. The set $\mathcal{C}$ consists of all triples (b', τ, w), such that $S(\sigma)$ mentions τ, $\mathcal{T}[\langle \tau, b', \sigma \rangle] \neq \perp$ (in particular, $S(\tau)$ should mention σ so that the index $\langle \tau, b', \sigma \rangle$ is well defined), and $w = W(\mathcal{T}[\langle \tau, b', \sigma \rangle])$. The lb-constraint (of the instance I) is $S(\sigma)_{-\varsigma}$ and the bag of labels is b. The triples of a legal cover correspond to entries of $\mathcal{T}$ that form the trees $t_1, \ldots, t_m$ mentioned earlier.

Lines 1–5 of Update(σ, ς) construct the set $\mathcal{C}$. Line 6 simultaneously solves the MINLBCOVER problem for all subbags b of Λ.

Algorithm FindMinPattern(S^W, Λ)

```
1:  Initialize()
2:  while Q ≠ ∅ do
3:      ⟨σ', b, σ⟩ ← Q.pop()
4:      if T[⟨σ', b, σ⟩] = ⊥ then
5:          fail (no Λ-pattern exists)
6:      if b = Λ then
7:          return T[⟨σ', b, σ⟩]
8:      if σ ≠ ⋆ then
9:          for all ς ∈ dom(S) ∪ {⋆} s.t. S(σ)_{-ς} ≠ ∅ do
10:             Update(σ, ς)
```

Subroutine Initialize()

```
1:  Q ← empty priority queue
2:  for all σ ∈ dom(S), ς ∈ dom(S) ∪ {⋆}, and b ⊆ Λ do
3:      if S(σ)_{-ς} ≠ ∅ then
4:          T[⟨σ, b, ς⟩] ← ⊥
5:          if b = {{σ}} then
6:              T[⟨σ, b, ς⟩] ← a tree having a single σ-node
7:          Q.insert(⟨σ, b, ς⟩, W(T[⟨σ, b, ς⟩]))
```

Subroutine Update(σ, ς)

```
1:  C ← ∅
2:  for all labels τ mentioned by S(σ) do
3:      for all subbags b' ⊆ Λ do
4:          if S(τ) mentions σ and T[⟨τ, b', σ⟩] ≠ ⊥ then
5:              C ← C ∪ {(b', τ, W(T[⟨τ, b', σ⟩]))}
6:  MinCovers ← FindMinCovers(Λ, S(σ)_{-ς}, C)
7:  for all subbags b ⊆ Λ do
8:      if MinCovers[b − {{σ}}] ≠ ⊥ then
9:          T[⟨σ, b, ς⟩] ← Assemble(σ, MinCovers[b − {{σ}}])
10:         Q.update(⟨σ, b, ς⟩, W(T[⟨σ, b, ς⟩]))
```

Subroutine Assemble$(\sigma, \mathcal{C}')$

```
1:  t ← a tree with a single σ-node v
2:  for all (b', τ, w) ∈ C' do
3:      t_{b'} ← T[⟨τ, b', σ⟩]
4:      let u be a τ-node of t_{b'} s.t. nbr(u) ∈ [[S(τ)]]_{-σ}
5:      t ← t ∪ t_{b'} ∪ {v, u}
```

Figure 3: Finding a minimal Λ-pattern

Hence, Λ (rather than some b) is the first argument of the call to FindMinCovers in line 6. That call returns the array $MinCovers$ that has an index b for every subbag b of Λ. $MinCovers[b]$ is a minimal legal cover of b (if there is no cover, then $MinCovers[b] = \perp$). From each $MinCovers[b]$, we need to construct a b-pattern t and assign it to $\mathcal{T}[\langle\sigma, b, \varsigma\rangle]$. Line 7 iterates over all subbags b of Λ. Line 8 tests that[5] $b - \{\!\{\sigma\}\!\}$ has a cover. If so, line 9 calls Assemble$(\sigma, \mathcal{C}')$, where $\mathcal{C}'$ is $MinCovers[b - \{\!\{\sigma\}\!\}]$, in order to construct t, as described earlier. Note that we need a cover of $b - \{\!\{\sigma\}\!\}$, because the label σ of v could be in b. Line 10 updates the priority of $\langle\sigma, b, \varsigma\rangle$ in $\mathcal{Q}$ with the weight of the new t.

Observe that in the subroutine Assemble$(\sigma, \mathcal{C}')$, each execution of line 3 uses a new variable for $t_{b'}$. Thus, this subroutine constructs a tree even if the same triple appears more than once in $\mathcal{C}'$.

3.3 Correctness and Complexity

The description in the previous section shows that the subroutine Assemble constructs b-patterns that are valid entries of $\mathcal{T}$, as defined earlier. The crux of showing correctness of FindMinPattern is the following lemma. The proof is similar to the one given in [14] for an algorithm that finds minimal Steiner trees.

LEMMA 3.2. *In an execution of* FindMinPattern(S^W, Λ)*, the triples $\langle\sigma', b, \sigma\rangle$ are removed from $\mathcal{Q}$ in an order of increasing $W(\mathcal{T}[\langle\sigma', b, \sigma\rangle])$. Furthermore, whenever $\langle\sigma', b, \sigma\rangle$ is removed from $\mathcal{Q}$ in line 3:*

1. *If $\sigma \neq \star$, then $\mathcal{T}[\langle\sigma', b, \sigma\rangle]$ is a minimal b-pattern having a σ'-node v with $\lambda(\mathsf{nbr}(v)) \in [\![S(\sigma)_{-\varsigma}]\!]$. (And $\mathcal{T}[\langle\sigma', b, \sigma\rangle] = \perp$ if no such b-pattern exists.)*
2. *If $\sigma = \star$, then $\mathcal{T}[\langle\sigma', b, \sigma\rangle]$ is a minimal b-pattern having a σ'-node. (And $\mathcal{T}[\langle\sigma', b, \sigma\rangle] = \perp$ if no such b-pattern exists.)*

As a result of Lemma 3.2, we get the following theorem, which states the correctness of the algorithm. This theorem also states a dependency of the running time on the input, and on the running time of FindMinCovers (which we consider in the next section).

THEOREM 3.3. FindMinPattern(S^W, Λ) *returns a minimal Λ-pattern, or fails if none exists. The number of operations and function calls is polynomial in $2^{|\Lambda|} \cdot |dom(S)|$.*

Recall that MINLBCOVER gets as input a bag Γ, an lb-constraint $\mathcal{B}$, and a set $\mathcal{C}$ of triples. When we consider the parameterized complexity of MINLBCOVER, the parameter is $|\Gamma|$. Theorem 3.3 implies the following corollary that relates the parameterized complexity of MINPATTERN to that of MINLBCOVER under a language of lb-specifications.

COROLLARY 3.4. *Under a language of lb-specifications, the following holds. If* MINLBCOVER *is FPT, then* MINPATTERN *is FPT.*

Let k be a natural number. The definition of the problem *k-bounded* MINPATTERN is the same as MINPATTERN, except that we make the assumption that $|\Lambda| \leq k$ (but the size of the schema S can be unbounded). For a language of lb-specifications, *containment checking* is the problem of testing whether $b \in [\![\mathcal{B}]\!]$, when given a bag $b \in \mathcal{F}_{\mathbf{\Sigma}}$ and an lb-constraint $\mathcal{B}$ represented by an lb-specification. A procedure for containment checking is sufficient for solving MINLBCOVER (but not necessarily most efficiently). The problem *k-bounded* containment checking is the same as containment checking, except that $|b| \leq k$ is assumed. The following corollary of Theorem 3.3 relates k-bounded MINPATTERN to k-bounded containment checking.

COROLLARY 3.5. *Let k be a fixed natural number. For a language of lb-specifications, if k-bounded containment checking is in polynomial time then so is k-bounded* MINPATTERN.

Regarding Corollaries 3.4 and 3.5, it may seem that to guarantee the running times, the language of lb-specifications should support an efficient construction of $\mathcal{B}_{-\varsigma}$ when $\mathcal{B}$ and ς are given. However, this is not really necessary, because we can use $S(\sigma)$ instead of $S(\sigma)_{-\varsigma}$ in the call to FindMinCovers by incorporating the following changes. We add to $\mathcal{C}$ the triple $(\{\rho\}, \varsigma, 1)$, where ρ is a fresh new label, and compute covers that contain ρ.

[5] Note that this is bag difference, that is, if $\sigma \in b$ then $b - \{\!\{\sigma\}\!\}$ is obtained from b by decrementing the multiplicity of σ by 1, and otherwise, $b - \{\!\{\sigma\}\!\}$ is just b.

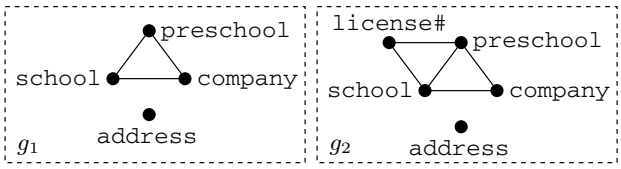

Figure 4: Mux graphs g_1 and g_2

4. LB-SPECIFICATIONS

We now consider languages for lb-specifications, and study the complexity of MINLBCOVER under them. We note that there are many (practically motivated) such languages, and this section by no means covers all of them; rather, our goal is to study the complexity of MINPATTERN for a few basic languages. We start with a language that allows one to specify mutual exclusions between pairs of labels.

4.1 Mutual-Exclusion Graphs

Recall that an lb-constraint is a nonempty subset $\mathcal{B}$ of $\mathcal{F}_{\mathbf{\Sigma}}$. A *mutual-exclusion graph* (abbr. *mux graph*) defines an lb-constraint by specifying the set of allowed labels, along with pairwise mutual exclusions among them. Our goal in studying the class of mux-graph specifications is threefold. First, this class captures the basic constraint of mutual exclusion, which is practically useful. Second, as we will show, this class enables us to exhibit the fact that fine details may make the difference between tractability (FPT) and intractability. Third, the algorithm we devise here will be used in the next section for handling regular expressions.

Formally, a mux graph g has nodes that are uniquely labeled (i.e., for all nodes u and v of g, if $u \neq v$ then $\lambda(u) \neq \lambda(v)$).[6] The lb-constraint $\mathcal{F}_{\mathbf{\Sigma}}(g)$ that is defined by the mux graph g comprises all the bags $b \in \mathcal{F}_{\mathbf{\Sigma}}$, such that all the labels of b belong to $\lambda(g)$, and for all edges $\{u, v\}$ of g, at most one of $\lambda(u)$ and $\lambda(v)$ is in b. In other words, $b \in \mathcal{F}_{\mathbf{\Sigma}}(g)$ if the set of labels that occur in b corresponds to an independent set of g. Note that $\mathcal{F}_{\mathbf{\Sigma}}(g)$ is closed under containment, which means that $[\![\mathcal{F}_{\mathbf{\Sigma}}(g)]\!] = \mathcal{F}_{\mathbf{\Sigma}}(g)$. Later we will generalize mux graphs by allowing some restrictions on the multiplicity of labels.

EXAMPLE 4.1. The two mux graphs g_1 and g_2 shown in Figure 4 could be used for specifying $S(\texttt{person})$. The lb-constraint $\mathcal{F}_{\mathbf{\Sigma}}(g_1)$ consists of all the bags that include `address` and one of `school`, `preschool` and `company`, with any multiplicity (including 0). The lb-constraint $\mathcal{F}_{\mathbf{\Sigma}}(g_2)$ is the same as $\mathcal{F}_{\mathbf{\Sigma}}(g_1)$, except that the label `#license` can be included provided that both `school` and `preschool` are absent (i.e., students in schools are not permitted to drive). □

For a mux graph g and a bag b, containment checking (i.e., testing whether $b \in [\![\mathcal{F}_{\mathbf{\Sigma}}(g)]\!]$) is trivial. Hence, Corollary 3.5 implies that MINPATTERN is solvable in polynomial time if lb-specifications are given by mux graphs and $|\Lambda|$ is fixed. Next, we develop FPT algorithms for MINPATTERN that apply to some classes of mux graphs. Recall from Corollary 3.4 that it is enough to find FPT algorithms for MINLBCOVER.

4.1.1 Disjoint Cliques

We first present an FPT algorithm for MINLBCOVER in the special case of mux graphs consisting of *disjoint cliques*, that is, graphs such that every connected component is a clique. In Figure 4, for example, g_1 comprises two disjoint cliques, but g_2 is not a disjoint-clique graph. Later, we will show that the algorithm we present here can be applied to a much broader class of mux graphs (that includes g_2).

[6] A mux graph could be defined as a graph over labels; we use the given definition for uniformity of presentation.

Algorithm MuxMinCovers$(\Gamma, g, \mathcal{C})$

```
1: let v_1, ..., v_n be a friendly order of V(g)
2: for all subbags b ⊆ Γ do
3:     M_0[b] ← ⊥
4:     for i = 1 to n do
5:         N_i[b] ← ⊥
6: M_0[∅] ← ∅
7: for i = 1 to n do
8:     for all subbags b ⊆ Γ by increasing |b| do
9:         for all Δ = (b_Δ, τ, w) ∈ C where τ = λ(v_i) do
10:            for all subbags b' ⊆ b such that b ⊆ b' ∪ b_Δ do
11:                let j < i be the maximal s.t. {v_j, v_i} ∉ E(g), or 0
                   if {v_j', v_i} ∈ E(g) for all 1 ≤ j' < i
12:                let C_i = N_i[b] ⊎ {{Δ}} and C_j = M_j[b'] ⊎ {{Δ}}
13:                N_i[b] ← min(N_i[b], C_i, C_j)
14:        M_i[b] ← min(M_{i-1}[b], N_i[b])
15: return M_n
```

Figure 5: Algorithm MuxMinCovers **for finding minimal covers**

The pseudo code of the algorithm, called MuxMinCovers, is given in Figure 5. The input consists of a bag Γ, a disjoint-clique mux graph g, and a set $\mathcal{C}$ of triples $\Delta = (b, \tau, w)$ (where b is a bag, τ is a label, and w is a weight).

Since g consists of disjoint cliques, there is a *friendly order* $v_1, \ldots, v_n$ of its nodes, which means that every clique constitutes a consecutive subsequence (i.e., substring) of $v_1, \ldots, v_n$.

For each i in $\{1, \ldots, n\}$, the algorithm maintains an array $\mathcal{M}_i$ and an array $\mathcal{N}_i$. In addition, there is another array $\mathcal{M}_0$. For each of those arrays, an index is a subbag b of Γ. So, if $\Gamma = (X, \mu_\Gamma)$, then the size of each array is $\prod_{\sigma \in X}(\mu_\Gamma(\sigma) + 1)$.

Before we proceed, we need some notation. For a number $i \in \{0, \ldots, n\}$ and a subbag b of Γ, a legal cover $\mathcal{C}'$ of b is called an *i-cover* of b if for every triple (b_Δ, τ, w) in $\mathcal{C}'$, the label τ is one of $\lambda(v_1), \ldots, \lambda(v_i)$. In particular, note that only the empty bag has a 0-cover (and this cover is the empty bag). Also note that every legal cover is an n-cover, and every i-cover is a j-cover for $j \geq i$. For $i \in \{1, \ldots, n\}$, the bag $\mathcal{C}'$ is called a *strict i-cover* of b if $\mathcal{C}'$ is an i-cover of b and contains one or more occurrences of $\lambda(v_i)$ (i.e., $\mathcal{C}'$ contains a triple (b_Δ, τ, w) such that $\tau = \lambda(v_i)$). In other words, an i-cover is strict if it is not an $(i-1)$-cover.

When the algorithm terminates, each $\mathcal{M}_i[b]$ contains a minimal i-cover of b, and each $\mathcal{N}_i[b]$ contains a minimal strict i-cover of b. In line 15, the algorithm returns the array $\mathcal{M}_n$. Lines 2–6 initialize all the $\mathcal{M}_i[b]$ and $\mathcal{N}_i[b]$ to $\bot$, except for $\mathcal{M}_o[\emptyset]$ which is initialized to $\emptyset$. Note that the weight of $\emptyset$ is 0, and $\bot$ has an infinite weight.

The main part of the algorithm is the dynamic programming in lines 7–14. The loop of line 7 iterates over all $i \in \{1, \ldots, n\}$ in increasing order. In the ith iteration, lines 8–14 set the arrays $\mathcal{M}_i$ and $\mathcal{N}_i$ to their final values as follows. Line 8 iterates over all subbags b of Γ by increasing size. Lines 9–13 set the value of $\mathcal{N}_i[b]$, and line 14 sets the value of $\mathcal{M}_i[b]$. Lines 11–13 are executed for all triples $\Delta = (b_\Delta, \tau, w)$ of $\mathcal{C}$ with $\tau = \lambda(v_i)$, and all the subbags b' of b, such that $b \subseteq b' \uplus b_\Delta$; these lines attempt to improve the

strict i-cover of b found so far, by adding Δ to an i-cover $\mathcal{C}'$ of b'. The algorithm considers two such $\mathcal{C}'$ where the addition of Δ does not violate the lb-constraint specified by g. The first (denoted $\mathcal{C}_i$ in the algorithm) is a strict i-cover of b', and then $\triangle$ can be added $\mathcal{C}'$ because $\mathcal{C}'$ already has a triple labeled with $\tau = \lambda(v_i)$. The second (denoted $\mathcal{C}_j$ in the algorithm) is a j-cover of b', where j is the maximal index (subject to $j < i$) such that either $j = 0$ or v_j and v_i are in different cliques (here we use the fact that $v_1 \ldots, v_n$ is a friendly order). Line 11 finds j, and line 13 sets $\mathcal{N}_i[b]$ to the strict i-cover that has the minimum weight among the current $\mathcal{N}_i[b]$ and the two $\mathcal{C}' \uplus \{\!\{\triangle\}\!\}$ mentioned earlier. Note that $\bot \uplus \Delta = \bot$. Finally, line 14 sets $\mathcal{M}_i[b]$ to the cover that has the minimum weight among $\mathcal{M}_{i-1}[b]$ and $\mathcal{N}_i[b]$.

Correctness of the algorithm follows from the next lemma. The proof is by a rather straightforward (double) induction on i and $|b|$.

LEMMA 4.2. *Let b be a subbag of Γ. At the end of the ith iteration of the loop of line 7 of* MuxMinCovers$(\Gamma, g, \mathcal{C})$*,*

1. *$\mathcal{N}_i[b]$ is a minimal strict i-cover of b (or $\bot$ if none exist).*
2. *$\mathcal{M}_i[b]$ is a minimal i-cover of b (or $\bot$ if none exist).*

The following theorem states the correctness and running time of the algorithm. Correctness is an immediate corollary of Lemma 4.2, and the running-time analysis is straightforwardly obtained from the pseudo code of Figure 5.

THEOREM 4.3. *Let g be a disjoint-clique graph, and $\mathcal{M}$ be the array returned by* MuxMinCovers$(\Gamma, g, \mathcal{C},)$*. For all $b \subseteq \Gamma$, the entry $\mathcal{M}[b]$ is a minimal cover of b (or $\bot$ if none exist). The running time is polynomial in the input size and 2^{Γ}.*

4.1.2 Beyond Disjoint Cliques

We now extend, beyond disjoint cliques, the class of mux graphs that MuxMinCovers can handle. First, we generalize the notion of a *friendly order* over the nodes of a graph.

DEFINITION 4.4. Let g be a graph. A *friendly order* of g is a total (linear) order $\prec$ on $\mathsf{V}(g)$, such that for all nodes $w \in \mathsf{V}(g)$, the preceding neighbors of w succeed the preceding non-neighbors of w; that is, for all $u, v, w \in \mathsf{V}(g)$:

$$u \prec v \prec w \wedge \{u, w\} \in \mathsf{E}(g) \quad \Rightarrow \quad \{v, w\} \in \mathsf{E}(g)\,. \quad \square$$

Put differently, $v_1, \ldots, v_n$ is a friendly order of g if for all i, node v_i and all its preceding neighbors form a suffix of $v_1, \ldots, v_i$. We use the term "friendly" because every node is placed right after its (already placed) neighbors.

As an example, the following is a friendly order of the graph g_2 of Figure 4 (nodes are represented by their labels).

`#license, school, preschool, company, address`

As other examples, observe that a graph g has a friendly order if g is a clique, a star, or a path. Of course, these cases do not cover all the graphs with a friendly order (as observed in Figure 4). Note that g has a friendly order if (and only if) every connected component of g has a friendly order. Finally, not every graph has a friendly order; for instance, it can easily be shown that a cycle of four or more nodes does not have any friendly order.

Obviously, the algorithm MuxMinCovers is correct whenever we have a friendly order (according to Definition 4.4), and not just in the case of disjoint cliques. We would like to identify when a mux graph has a friendly order. Interestingly, friendly orders are tightly related to *interval graphs*. We say that g is an interval graph if there is a function I that maps every node $v \in \mathsf{V}(g)$ to a closed interval $I(v)$ on the real line, such that for every two nodes $v, w \in \mathsf{V}(g)$ it holds that $\{v, w\} \in \mathsf{E}(g)$ if and only if $I(v) \cap I(w) \neq \emptyset$. If g is an interval graph, then I is called a *realizer*. Examples of interval graphs are cliques and *caterpillar trees* (which generalize paths and stars). Booth and Lueker [4] showed that determining whether g is an interval graph, and finding a realizer if so, are in polynomial time, and a more recent algorithm by Habib et al. [10] performs these tasks in linear time in g. This is important, because finding a realizer is *the same problem* as finding a friendly order.

PROPOSITION 4.5. *A graph g has a friendly order if and only if it is an interval graph. Furthermore, a friendly order can be obtained in polynomial time from a realizer.*

We conclude that if interval mux graphs are used for expressing lb-specifications, then both MINPATTERN and MINLBCOVER are tractable. Next, we further push this tractability to the class of *circular-arc* graphs.

We say that a graph g is a circular-arc graph if every node v of g can be mapped to an arc $a(v)$ on a circle, such that two arcs $a(u)$ and $a(v)$ intersect if and only if u and v are adjacent (in other words, it is the same definition as that of an interval graph, except that now arcs are used instead of intervals). Observe that circular-arc graphs are a proper generalization of interval graphs. For example, every cycle is a circular-arc graph, but most of the cycles are not interval graphs. Tucker [23] showed that recognition of circular-arc graphs is in polynomial time, and more recently McConnell [20] gave a linear-time recognition algorithm.

Now, consider the problem MINLBCOVER with an lb-constraint that is specified by a circular-arc mux graph. We can efficiently reduce this problem to MINLBCOVER with an interval mux graph. Therefore, it follows that both MINPATTERN and MINLBCOVER are FPT if lb-specifications are given by circular-arc mux graphs.

Next, we consider another (tractable) extension. Mux graphs lack the ability to express constraints on the multiplicity of nodes having a given label. As a simple example, the mux graph g_2 in Figure 4 allows any multiplicity of `#license`. As another example, the *Mondial* schema[7] does not allow a `border` element to have more than two `country` children. We extend mux graphs so that they can specify an upper bound on the number of occurrences of a label, as follows. A *multiplicity-bounded* mux graph, denoted as $\overline{\text{mux}}$ graph, is a pair (g, β), where g is a mux graph and $\beta : \mathsf{V}(g) \to \mathbb{N} \cup \{\infty\}$ is a function. The lb-constraint $\mathcal{F}_{\mathbf{\Sigma}}((g, \beta))$ comprises all the bags $b \in \mathcal{F}_{\mathbf{\Sigma}}(g)$, such that for each node v of g it holds that $\mu_b(\lambda(v)) \leq \beta(v)$.

The algorithm MuxMinCovers (Figure 5) can be easily extended to (efficiently) support multiplicity bounds. In a nutshell, instead of the arrays $\mathcal{M}_i$ and $\mathcal{N}_i$, we compute the arrays $\mathcal{M}_i^j$ and $\mathcal{N}_i^j$, respectively, for $j \leq \beta(v_i)$. Upon termination, each $\mathcal{M}_i^j[b]$ (resp., $\mathcal{N}_i^j$) contains a minimal i-cover (resp., minimal strict i-cover) of b with at most j occurrences of $\lambda(v_i)$. An observation needed here is that $\beta(v_i) \leq |\Lambda|$ can be assumed. The above is also applicable when circular-arc graphs are used.

We conclude this section with the following theorem, which states the tractability of the problems MINLBCOVER and MINPATTERN when lb-specifications belong to the class of circular-arc $\overline{\text{mux}}$ graphs.

THEOREM 4.6. *If lb-specifications are circular-arc $\overline{\text{mux}}$ graphs, then both* MINLBCOVER *and* MINPATTERN *are FPT.*

[7] `http://www.dbis.informatik.uni-goettingen.de/Mondial/`

Next, we address the question of whether Theorem 4.6 can be extended to the whole class of mux graphs (even regardless of multiplicity bounds). Next, we give a negative answer to this question (under standard complexity assumptions). Specifically, the following theorem shows that when general mux graphs are allowed, both MINPATTERN and MINLBCOVER are hard for the complexity class $W[1]$, which is believed not to be contained in FPT [6, 9]. The proof is by a reduction from the problem of determining whether a graph has an independent set of size k, with k being the parameter.

THEOREM 4.7. *Suppose that lb-constraints are specified by mux graphs. The following decision problems are $W[1]$-hard with the parameters $|\Lambda|$ and $|\Gamma|$, respectively.*

1. *Given a schema S and a bag Λ of labels, is there a Λ-pattern with $2|\Lambda|+1$ or fewer nodes?*
2. *Given an input $(\Gamma, g, \mathcal{C})$ for* MINLBCOVER*, is there any legal cover of Γ?*

Thus, under standard complexity assumptions, Theorem 4.6 does not extend to general mux graphs (i.e., without the restriction to circular-arc graphs).

4.2 Regular Expressions

In this section, we consider lb-specifications that are given as regular expressions, similarly to Example 2.2. As in that example, we define $\mathcal{F}_{\mathbf{\Sigma}}(e)$ to be the set $\mathcal{L}^{\mathrm{b}}(e)$. The following theorem states the tractability of MINLBCOVER and MINPATTERN when regular expressions are used for specifying lb-constraints.

THEOREM 4.8. *If lb-specifications are regular expressions, then both* MINLBCOVER *and* MINPATTERN *are FPT.*

Next, we prove Theorem 4.8 by presenting an FPT algorithm for MINLBCOVER (recall Corollary 3.4).

4.2.1 The Algorithm

The input of our algorithm is $(\Gamma, e, \mathcal{C})$, where Γ is a bag, e is a regular expression specifying the lb-constraint $[\![\mathcal{L}^{\mathrm{b}}(e)]\!]$, and $\mathcal{C}$ is a set of triples. The algorithm applies a recursion on the structure of e. In each recursive step, we consider a sub-expression e' of e, and construct an array $\mathcal{M}$, which is similar to the arrays $\mathcal{M}_i$ built in MuxMinCovers. That is, an index of $\mathcal{M}$ is a subbag b of Γ, and when the step terminates, each $\mathcal{M}[b]$ is a minimal legal cover with respect to $(b, e', \mathcal{C})$. The construction of $\mathcal{M}$ is by a reduction to MINLBCOVER under $\overline{\text{mux}}$-graph specifications with multiplicity bounds, where each $\overline{\text{mux}}$ graph is an extremely simple interval graph. Actually, we could give a more direct algorithm that does not use a reduction to $\overline{\text{mux}}$ graphs; however, using this reduction shortens the presentation (by avoiding the repetition of arguments from the previous section).

We now describe the algorithm in detail. Recall that a regular expression is defined by the following language.

$$e := \quad \sigma \mid \epsilon \mid e* \mid e? \mid e\,e \mid e|e$$

So, we consider the six options of composing regular expressions. The case where $e = \sigma$ or $e = \epsilon$ is straightforward. For $e = e_1?$, we simply ignore the question mark and solve the problem for e_1. So, three cases remain.

Case 1: $e = e_1*$**.** Suppose that $\mathcal{M}_1$ is the result of the algorithm for e_1. Let σ_1 be a label, let g be the mux graph consisting of a single σ_1-node, and let $\mathcal{C}'$ be the set of triples $\Delta_b = (b, \sigma_1, \mathrm{weight}(\mathcal{M}_1[b]))$ for all bags $b \subseteq \Gamma$. The next step is to execute $\mathsf{MuxMinCovers}(\Gamma, g, \mathcal{C}')$. Let $\mathcal{M}'$ be the result. The array $\mathcal{M}$ that we return is the following. For each $b' \subseteq \Gamma$, we set $\mathcal{M}[b']$ to $\biguplus_{\Delta_b \in \mathcal{M}'[b']} \mathcal{M}_1[b]$.

Case 2: $e = e_1\,e_2$**.** Suppose that $\mathcal{M}_1$ and $\mathcal{M}_2$ are the results of the algorithm for e_1 and e_2, respectively. Let (g, β) be a $\overline{\text{mux}}$ graph, such that g has exactly two nodes v_1 and v_2 labeled with σ_1 and σ_2, respectively, and $\beta(v_1) = \beta(v_2) = 1$. There are no edges in g. Let $\mathcal{C}'$ be the set of triples $\Delta_b = (b, \sigma_i, \mathrm{weight}(\mathcal{M}_i[b]))$ for all bags $b \subseteq \Gamma$ and $i \in \{1, 2\}$. We execute $\mathsf{MuxMinCovers}(\Gamma, (g, \beta), \mathcal{C}')$, and suppose that $\mathcal{M}_0$ is the result. Due to β, the cover $\mathcal{M}_0[b]$ is either a singleton or a pair (except for $b = \emptyset$ where $\mathcal{M}_0[b]$ can be empty). We define $\mathcal{M}[b]$, where $b \subseteq \Gamma$, as follows. If $\mathcal{M}_0[b]$ is a singleton $\Delta_b = (b, \sigma_i, \mathrm{weight}(\mathcal{M}_i[b]))$, then we set $\mathcal{M}[b]$ to $\mathcal{M}_i[b]$. Otherwise, $\mathcal{M}_0(b)$ contains two triples Δ_{b_1} and Δ_{b_2} with the labels σ_1 and σ_2, respectively, such that $b = b_1 \uplus b_2$. Then we set $\mathcal{M}[b]$ to $\mathcal{M}_1[b_1] \uplus \mathcal{M}_2[b_2]$.

Case 3: $e = e1 \mid e_2$**.** This case is handled exactly like the previous case, except for the following change. In the $\overline{\text{mux}}$ graph (g, β), an edge connects the two nodes of g. Note that in this case, $\mathcal{M}_0[b]$ is always a singleton (or the empty set).

This completes the description of the algorithm, and by that, the proof of Theorem 4.8.

5. DIRECTED PATTERNS

Often, direction of edges are important, and even crucial. As an example, in the tree t_3 of Figure 7, only directions can determine whether the top company sells to the bottom one or vice-versa. Directions are also needed for handling format restrictions. For example, in an XML document, each node (except the root) has one parent. To distinguish between a parent and a child, directions are needed. In this section, we extend our framework and complexity results to a directed model.

5.1 Problem Definition

We first modify the basic definitions. Recall that an lb-constraint was defined as a nonempty subset of $\mathcal{F}_{\mathbf{\Sigma}}$. A *directed lb-constraint* is a nonempty subset of $\mathcal{F}_{\mathbf{\Sigma}} \times \mathcal{F}_{\mathbf{\Sigma}}$. The *containment closure* of a directed lb-constraint $\mathcal{B}$, denoted by $[\![\mathcal{B}]\!]$, is the set of all the pairs $(b'_{\mathsf{in}}, b'_{\mathsf{out}})$, such that there is some $(b_{\mathsf{in}}, b_{\mathsf{out}}) \in \mathcal{B}$ that satisfies $b'_{\mathsf{in}} \subseteq b_{\mathsf{in}}$ and $b'_{\mathsf{out}} \subseteq b_{\mathsf{out}}$.

A *directed schema* S maps each label $\sigma \in dom(S)$ to a directed lb-constraint $S(\sigma)$. Given a σ-node v, the directed lb-constraint $S(\sigma)$ imposes constraints on the labels of $\mathsf{nbr}^g_{\mathsf{in}}(v)$ and $\mathsf{nbr}^g_{\mathsf{out}}(v)$, which are the sets of incoming and outgoing neighbors of v, respectively. Formally, a directed graph g *conforms to* S, denoted by $g \models S$, if for all nodes $v \in \mathsf{V}(g)$ it holds that $\lambda(v) \in dom(S)$ and $(\lambda(\mathsf{nbr}^g_{\mathsf{in}}(v), \lambda(\mathsf{nbr}^g_{\mathsf{out}}(v)) \in S(\lambda(v))$. As earlier, $[\![S]\!]$ denotes the directed schema that is obtained from S by replacing every $S(\sigma)$ with its closure $[\![S(\sigma)]\!]$. We say that t is a *directed tree* if t is a tree when directions are ignored. (Towards the end of this section, we consider another popular notion of directed trees.) For a directed schema S and a bag $\Lambda \in \mathcal{F}_{\mathbf{\Sigma}}$, a *directed Λ-pattern* (*under* S) is defined similarly to a Λ-pattern. Namely, it is a directed tree t, such that $\Lambda \subseteq \lambda(\mathsf{V}(t))$ and $t \models [\![S]\!]$.

EXAMPLE 5.1. We define three directed schemas S_1, S_2 and S_3, each having the domain consisting of all the labels that appear in Figure 6. Let $\Lambda = \{\!\{\texttt{company}, \texttt{coke}, \texttt{pizza}\}\!\}$ and $\Lambda' = \{\!\{\texttt{company}, \texttt{company}, \texttt{company}\}\!\}$.

The schema S_1 is defined by the graph g of Figure 6 as follows. For a label $\sigma \in dom(S_1)$, the directed lb-constraint $S_1(\sigma)$ is the pair $\mathcal{B}_{\mathsf{in}} \times \mathcal{B}_{\mathsf{out}}$, such that $(b_{\mathsf{in}}, b_{\mathsf{out}}) \in \mathcal{B}_{\mathsf{in}} \times \mathcal{B}_{\mathsf{out}}$ if for all labels

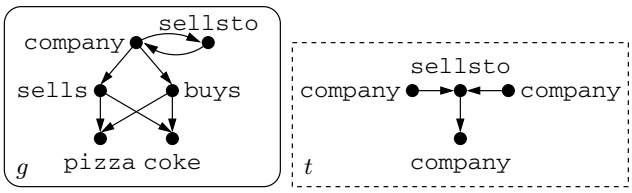

Figure 6: A graph g defining S_1 and a directed Λ'-pattern t for $\Lambda' = \{\!\{\texttt{company}, \texttt{company}, \texttt{company}\}\!\}$

τ_{in} and τ_{out} in b_{in} and b_{out}, respectively, the (directed) edges $(\tau_{\mathsf{in}}, \sigma)$ and $(\sigma, \tau_{\mathsf{out}})$ are in g. As an example, a `sells`-node can have any number of incoming `company`-nodes, and any number of outgoing `pizza`-nodes and `coke`-nodes. Under S_1, each of the three trees of Figure 7 is a directed Λ-pattern. In addition, the tree t of Figure 6 is a directed Λ'-pattern.

The schema S_2 is defined by regular expressions as follows. The meaning of the subscripts in and out is explained below.

$$\begin{aligned} e_{\texttt{company}} &= \texttt{sellsto}_{\mathsf{out}}{*}\ \texttt{sellsto}_{\mathsf{in}}{*}\ \texttt{sells}_{\mathsf{out}}?\ \texttt{buys}_{\mathsf{out}}? \\ e_{\texttt{sellsto}} &= \texttt{company}_{\mathsf{in}}\ \texttt{company}_{\mathsf{out}} \\ e_{\texttt{sells}} &= e_{\texttt{buys}} = \texttt{company}_{\mathsf{in}}{*}\ (\texttt{pizza}_{\mathsf{out}} \mid \texttt{coke}_{\mathsf{out}})? \\ e_{\texttt{pizza}} &= e_{\texttt{coke}} = \texttt{buys}_{\mathsf{in}}{*}\ \texttt{sells}_{\mathsf{in}}{*} \end{aligned} \quad (2)$$

Each e_σ specifies a directed lb-constraint $S_2(\sigma)$ as follows. For every bag b matching e_σ, construct a pair $(b_{\mathsf{in}}, b_{\mathsf{out}}) \in S_2(\sigma)$ such that b_x, where $x \in \{\mathsf{in}, \mathsf{out}\}$, is obtained by selecting all elements of b with the subscript x and then removing x. For example, the tree t of Figure 6 is not a directed Λ'-pattern under S_2, since $e_{\texttt{sellsto}}$ says that a `sellsto`-node cannot have more than one incoming `company`-node. As another example, the tree t_1 of Figure 7 is also not a directed Λ-pattern (under S_2), because $e_{\texttt{sells}}$ states that a `sells`-node cannot have both `pizza` and `coke` outgoing nodes. However, both t_2 and t_3 are directed Λ-patterns under S_2.

The schema S_3 is similar to S_2, except that a company cannot be both a buyer and a seller. Hence, $e_{\texttt{company}}$ is:

$$(\texttt{sellsto}_{\mathsf{out}}{*}\ \texttt{sells}_{\mathsf{out}}?) \mid (\texttt{sellsto}_{\mathsf{in}}{*}\ \texttt{buys}_{\mathsf{out}}?) \quad (3)$$

Note that S_3 is more restrictive than S_2, and hence, t is not a directed Λ'-pattern and t_1 is not a directed Λ-pattern. Also, now t_2 is not a directed Λ-pattern, since the `company`-node has outgoing edges to both a `sells`-node and a `buys`-node. But t_3 is a directed Λ-pattern. In particular, note that the top `company`-node has outgoing edges to a `sells`-node and a `sellsto`-node, and the bottom `company` has an outgoing `buys`-node and an incoming `sellsto`-node. □

A *weight function* W for a directed schema S is defined similarly

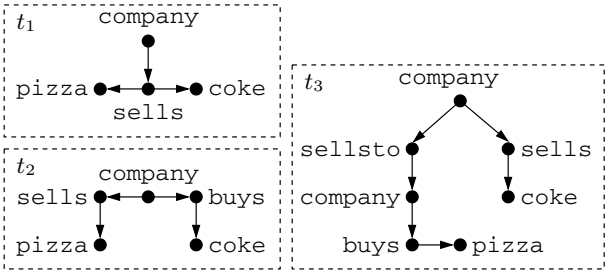

Figure 7: Three directed Λ-patterns t_1, t_2 and t_3, for $\Lambda = \{\!\{\texttt{company}, \texttt{coke}, \texttt{pizza}\}\!\}$

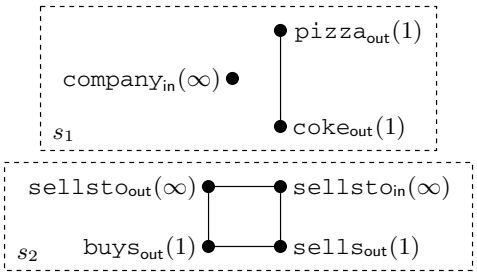

Figure 8: $\overline{\text{mux}}$ graphs over $L_{\mathsf{in}} \cup L_{\mathsf{out}}$

to the undirected case (Section 2.3), except that W does not have to be symmetric over $dom(S) \times dom(S)$. Given a weighted directed schema S^W and a directed graph g, the weight $W(g)$ of g is defined as in (1).

EXAMPLE 5.2. Consider the schemas S_1, S_2 and S_3 of Example 5.1, and the directed Λ-patterns t_1, t_2 and t_3 of Figure 7, where $\Lambda = \{\!\{\texttt{company}, \texttt{coke}, \texttt{pizza}\}\!\}$. Let W assign weight 1 to all nodes and edges. It can be verified that under S_1, the tree t_1 is a minimal directed Λ-pattern. Similarly, t_2 is minimal under S_2, and t_3 is minimal under S_3. □

Next, we consider the problem MINDIPATTERN, which is the directed version of MINPATTERN and is naturally defined as follows. The input consists of a weighted directed schema S^W and a bag Λ of labels, and the goal is to find a directed Λ-pattern t that has the minimum weight. Again, for this problem to be well defined, we need to describe languages for lb-specifications.

5.2 Directed lb-Specifications

For directed lb-specifications, we combine the undirected ones (e.g., those of Section 4) with the following transformation, which is essentially how S_2 and S_3 are specified in Example 5.1. We generate two disjoint sets of labels L_{in} and L_{out} by replacing every $\sigma \in dom(S)$ with the new labels σ_{in} and σ_{out}, respectively. At first, we define an lb-constraint $\mathcal{B}$ over $L_{\mathsf{in}} \cup L_{\mathsf{out}}$ by means of some undirected lb-specification. Then, $\mathcal{B}$ is conceptually translated into a directed lb-constraint by splitting each bag $b \in \mathcal{B}$ into a pair $(b_{\mathsf{in}}, b_{\mathsf{out}})$: the bag b_x, where $x \in \{\mathsf{in}, \mathsf{out}\}$, is obtained by selecting all the elements of b with the subscript x and then removing x.

We can now define two types of directed lb-specifications. The first consists of the $\overline{\text{mux}}$ graphs (g, β), such that $\lambda(\mathsf{V}(g))$ (i.e., the set of labels that appear in g) is a subset of $L_{\mathsf{in}} \cup L_{\mathsf{out}}$. The second type uses regular expressions over $L_{\mathsf{in}} \cup L_{\mathsf{out}}$.

Example 5.1 uses directed lb-specifications of the second type. The $\overline{\text{mux}}$ graphs $s_1 = (g_1, \beta_1)$ and $s_2 = (g_2, \beta_2)$ of Figure 8 are directed lb-specifications of the second type. Note that $\beta(v)$ is written in parentheses on the right of $\lambda(v)$. The (undirected) lb-constraints $\mathcal{F}_{\mathbf{\Sigma}}(s_1)$ and $\mathcal{F}_{\mathbf{\Sigma}}(s_2)$ are equal to $\mathcal{L}^{\mathrm{b}}(e_{\texttt{sells}})$ and $\mathcal{L}^{\mathrm{b}}(e_{\texttt{company}})$ of Equations (2) and (3), respectively. Also, note that s_1 is an interval (hence, circular-arc) $\overline{\text{mux}}$ graph, whereas s_2 is a circular-arc, but not interval $\overline{\text{mux}}$ graph.

5.3 Tractability

In order to extend our results to directed lb-specifications, we need only to adapt the generic algorithm FindMinPattern of Figure 3, which is fairly straightforward. Thus, we get the next theorem that extends Corollary 3.5 and Theorems 4.6 and 4.8.

Note that k-bounded MINDIPATTERN is defined similarly to k-bounded MINPATTERN (i.e., $|\Lambda| \leq k$ is assumed). Containment

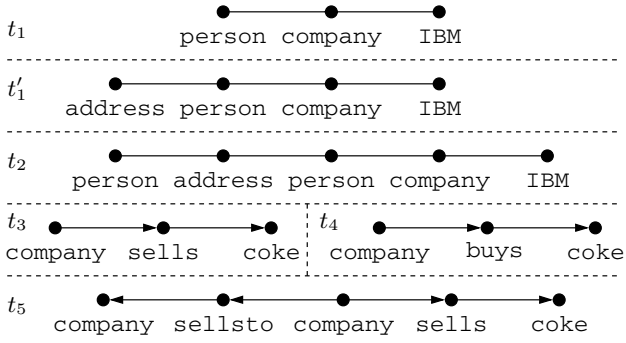

Figure 9: Redundant and (weakly) non-redundant patterns

checking for directed lb-constraints is the problem of testing whether $(b_{\mathsf{in}}, b_{\mathsf{out}}) \in [\![\mathcal{B}]\!]$, when given a pair $(b_{\mathsf{in}}, b_{\mathsf{out}}) \in \mathcal{F}_{\mathbf{\Sigma}} \times \mathcal{F}_{\mathbf{\Sigma}}$ and a directed lb-constraint $\mathcal{B}$ (represented by a directed lb-specification). In the *k-bounded* version, $|b_{\mathsf{in}}| \leq k$ and $|b_{\mathsf{out}}| \leq k$ are assumed.

THEOREM 5.3. *The following hold.*

1. *Let k be a fixed natural number. Under a language of directed lb-specifications, if k-bounded containment checking is in polynomial time, then so is k-bounded* MINDIPATTERN.

2. *If lb-specifications are given as circular-arc $\overline{mux}$ graphs or as regular expressions, then* MINDIPATTERN *is FPT.*

Sometimes a stronger definition of a Λ-pattern t is used, whereby t is required to be *rooted*, that is, have a node r such that every other node of t is reachable from r via a directed path in t. We can adapt FindMinPattern also to this stronger definition, and hence the above theorem remains valid in this case as well. Details are omitted due to a lack of space.

6. DEFINING THE TOP-K PATTERNS

In this section, we discuss how to define the problem of finding *k-minimal* (or *top-k*) patterns. When dealing with non-minimal patterns, we need a suitable notion of non-redundancy. We describe such a notion and use it to formally define the top-k Λ-patterns. Intuitively, the top-k Λ-patterns are required to be non-redundant and have the k-minimal weights (among all the non-redundant Λ-patterns). Developing an algorithm for finding the top-k Λ-patterns is left for future work.

Suppose that S is a schema or a directed schema, and consider a bag $\Lambda \in \mathcal{F}_{\mathbf{\Sigma}}$. By Definition 2.1, a Λ-pattern is a tree t, such that $\Lambda \subseteq \lambda(\mathsf{V}(t))$ and $t \models [\![S]\!]$. For instance, under the schema S_2 of Example 2.2 and for $\Lambda = \{\!\{\texttt{person}, \texttt{IBM}\}\!\}$, each of the trees t_1, t'_1 and t_2 of Figure 9 is a Λ-pattern. Recall that Λ-patterns are automatically generated queries, as in the schema-free [16] or flexible [5] paradigms. In that sense, t'_1 is useless, because it does not convey any information about the relationship between a person and IBM, beyond what is already known from t_1. This is a severe problem when top-k patterns are desired, because many patterns among those with the k-minimal weights could be likewise useless. For example, the patterns for $\{\!\{\texttt{person}, \texttt{IBM}\}\!\}$ with the k-minimal weights might be t_1 and another $k-1$ patterns obtained from t_1 by adding irrelevant nodes (like the address-node of t'_1).

The common approach to excluding useless patterns is by requiring non-redundancy [3, 5, 11, 13]. Formally, a Λ-pattern t is *redundant* if t has a proper subtree $t' \neq t$, such that t' is also a Λ-pattern; otherwise, t is *non-redundant*. Put differently, a Λ-pattern t is non-redundant if the removal of any leaf[8] from t violates $\Lambda \subseteq \lambda(\mathsf{V}(t))$. As an example, consider again Figure 9. For S_2 of Example 2.2 and $\Lambda = \{\!\{\texttt{person}, \texttt{IBM}\}\!\}$, the tree t_1 is non-redundant, but the tree t'_1 is redundant because even if the address leaf is removed, the result is still a Λ-pattern.

The above definition of non-redundancy has been applied when the goal is to find actual answers (i.e., subtrees of the underlying data graph), rather than patterns that are queries extracted from the schema. In that context, non-redundancy is relative to a set of nodes, rather than a bag of labels, of the tree. This subtle difference renders non-redundancy overly restrictive. As an example, the tree t_2 of Figure 9 represents a meaningful relationship, namely, some person and an IBM employee live in the same address. Although that relationship is not obtained from any subtree of t_2, the above definition deems t_2 a redundant Λ-pattern (when $\Lambda = \{\!\{\texttt{person}, \texttt{IBM}\}\!\}$), because t_1 is (isomorphic to) a proper subtree of t_2. So next, we propose a relaxation of non-redundancy.

We view t_2 as a meaningful Λ-pattern, because we can use the leftmost and rightmost nodes of t_2 as the output corresponding to $\Lambda = \{\!\{\texttt{person}, \texttt{IBM}\}\!\}$). The middle person-node serves only as part of the connection between the output nodes, and it just happens to have a label from Λ. Building on this intuition, we define *weak non-redundancy* as follows. We fix a label $\star \in \mathbf{\Sigma}$, and assume that $\star$ is never used in either a schema or Λ. An *output designation* of a Λ-pattern t is a tree t' that is obtained as follows. We choose $|\Lambda|$ nodes of t whose labels form the same bag as Λ and *designate* them as the *output*; we then replace the label of every non-output node with $\star$. Now, consider a (directed) schema S and a bag $\Lambda \in \mathcal{F}_{\mathbf{\Sigma}}$. A (directed) Λ-pattern t is *weakly non-redundant* if there is an output designation t' of t, such that t' is non-redundant (i.e., removing any leaf from t' causes a violation of $\Lambda \subseteq \lambda(\mathsf{V}(t'))$).

An easy observation is that a Λ-pattern t is weakly non-redundant if and only if no label has a higher multiplicity among the leaves of t than in Λ; that is, $b \subseteq \Lambda$, where b is the bag comprising the labels of the leaves. For example, given $\Lambda = \{\!\{\texttt{person}, \texttt{IBM}\}\!\}$, the tree t_2 of Figure 9 is weakly non-redundant, but t'_1 is not (because address is not in Λ). For $\Lambda = \{\!\{\texttt{company}, \texttt{coke}\}\!\}$, the tree t_5 of Figure 9 is weakly non-redundant. Finally, if Λ is the bag $\{\!\{\texttt{company}, \texttt{company}, \texttt{company}\}\!\}$, then the tree t of Figure 6 is weakly non-redundant. But that tree is weakly redundant for either $\{\!\{\texttt{company}\}\!\}$ or $\{\!\{\texttt{company}, \texttt{company}\}\!\}$, because company occurs three times among the leaves of t.

Observe that a minimal-weight Λ-pattern can be easily made non-redundant by repeatedly removing redundant leaves. So, the algorithms of the previous sections actually find Λ-patterns that are both minimal and (weakly) non-redundant.

COMMENT 6.1. Interestingly, some natural types of constraints require a suitable notion of non-redundancy even when looking just for a minimal Λ-pattern. One such case is that of *through* constraints. As an intuitive example, consider the schema of Figure 2, and let $\Lambda = \{\!\{\texttt{person}, \texttt{school}, \texttt{city}\}\!\}$. Suppose that we are interested in a Λ-pattern, such that the connection between person and city is through school. Then we impose a through constraint on school, which means that school should be on the path connecting person and city (and in particular cannot be a leaf). Such a Λ-pattern conveys, for example, the information that the person attends the school which is located in the city, as opposed to a Λ-pattern stating that both the person and the school are in the same city. The requirement that a Λ-pattern satisfies some through constraints is defined using the notion of weak non-

[8] A *leaf* is a node with exactly one incident edge.

redundancy, as follows. Suppose that $\Lambda = \Lambda_1 \uplus \Lambda_2$, where a through constraint is imposed on each label of Λ_2 but on none of Λ_1. A Λ-pattern satisfies the through constraints if it is weakly non-redundant with respect to Λ_1 (i.e., we ignore the labels of Λ_2 in the definition of weak non-redundancy). □

Next, we define top-k patterns. Consider a (directed) weighted schema S^W and a bag $\Lambda \in \mathcal{F}_{\mathbf{\Sigma}}$. Let k be a natural number. We say that T is a *top-k set* of (directed) Λ-patterns if $|T| = k$ and all of the following hold.

1. All the Λ-patterns of T are weakly non-redundant.
2. No two distinct members of T are isomorphic.
3. For all (directed) Λ-patterns t', if t' is weakly non-redundant, then either t' is isomorphic to some $t \in T$, or no $t \in T$ has a weight greater than $W(t')$.

If there are only $k' < k$ weakly non-redundant and pairwise non-isomorphic Λ-patterns, then a top-k set is a top-k' set.

As an example, consider the trees of Figure 9, and suppose that all nodes and edges have weight 1. For the schema S_2 of Example 2.2 and $\Lambda = \{\!\{\texttt{person}, \texttt{IBM}\}\!\}$, a top-2 set is $T = \{t_1, t_2\}$. For the schema S_2 of Example 5.1 and $\Lambda = \{\!\{\texttt{company}, \texttt{coke}\}\!\}$, a top-3 set is $T = \{t_3, t_4, t_5\}$.

As mentioned in the beginning of this section, the development of algorithms for finding a top-k set of (directed) Λ-patterns (where the input consists of a weighted schema S^W, a bag $\Lambda \in \mathcal{F}_{\mathbf{\Sigma}}$, and a number k) is a challenge that we defer to future work.

7. CONCLUSIONS

We gave a generic reduction of MINPATTERN to MINLBCOVER, which is independent of a particular language for expressing lb-specifications. As a corollary, k-bounded MINPATTERN is in polynomial time under the (reasonable) assumption that so is k-bounded containment. More importantly, we developed FPT algorithms for MINLBCOVER under the language of circular-arc $\overline{\text{mux}}$ graphs (recall that for general mux graphs the problem is $W[1]$-hard), and under the language of regular expressions. We also extended these results to the directed model. We believe that the results presented here are the basis for algorithms that find top-k Λ-patterns. But actually designing those algorithms is left for future work.

The formal setting we have used has labeled nodes, but no labeled edges. This is suitable for XML, but not for RDF. It is easy to model labeled edges by introducing a node in the "middle" of each edge. We call such a node a *connector*. For example, the label `wifeof` in Figure 1 is actually a connector (because it represents a relationship rather than an entity). If a connector appears in a Λ-pattern, then it should not be a leaf (or else we would not know which is the second entity in that relationship). Finding a minimal Λ-pattern under this restriction can be reduced to the problem of finding a minimal Λ-pattern under through constraints. It is fairly easy to adapt our generic algorithm to the latter (the details will be given in the full version).

The language of regular expressions captures the constraints imposed by DTDs. To model *XML Schemas* and *RDF Schemas* (and thereby exploit their expressiveness to the maximal extent), additional types of constraints are needed. Furthermore, an ontology should be taken into account in the case of RDF. Dealing with these extensions is an important task for future work.

We believe that the languages we proposed are sufficiently expressive to be useful in practice; in particular, these algorithms can substantially enhance the practicality of the pattern approach in keyword search on data graphs. We emphasize that in practice, lb-specifications can be derived either from DTDs or other schemas, or by an offline analysis of the underlying database.

Acknowledgments

We thank C. Seshadhri for fruitful discussions, and are grateful to Virginia Vassilevska Williams and Ryan Williams for observing Proposition 4.5.

8. REFERENCES

[1] H. Achiezra, K. Golenberg, B. Kimelfeld, and Y. Sagiv. Exploratory keyword search on data graphs. In *SIGMOD Conference*, pages 1163–1166. ACM, 2010.
[2] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *ICDT*, pages 296–313. Springer, 1999.
[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, pages 431–440, 2002.
[4] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. Syst. Sci.*, 13(3):335–379, 1976.
[5] S. Cohen, Y. Kanza, B. Kimelfeld, and Y. Sagiv. Interconnection semantics for keyword search in XML. In *CIKM*, pages 389–396. ACM, 2005.
[6] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.
[7] S. Dreyfus and R. Wagner. The Steiner problem in graphs. *Networks*, 1:195–207, 1972.
[8] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD Conference*, pages 927–940. ACM, 2008.
[9] M. Grohe and J. Flum. *Parameterized Complexity Theory*. Theoretical Computer Science. Springer, 2006.
[10] M. Habib, R. M. McConnell, C. Paul, and L. Viennot. Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theor. Comput. Sci.*, 234(1-2):59–84, 2000.
[11] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *VLDB*, pages 670–681. Morgan Kaufmann, 2002.
[12] A. Kemper, D. Kossmann, and B. Zeller. Performance tuning for SAP R/3. *IEEE Data Eng. Bull.*, 22(2):32–39, 1999.
[13] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *PODS*, pages 173–182. ACM, 2006.
[14] B. Kimelfeld and Y. Sagiv. New algorithms for computing Steiner trees for a fixed number of terminals. Accessible from the first author's home page (`http://www.cs.huji.ac.il/~bennyk`), 2006.
[15] B. Kimelfeld, Y. Sagiv, and G. Weber. ExQueX: exploring and querying XML documents. In *SIGMOD Conference*, pages 1103–1106. ACM, 2009.
[16] Y. Li, C. Yu, and H. V. Jagadish. Schema-free XQuery. In *VLDB*, pages 72–83. Morgan Kaufmann, 2004.
[17] Y. Luo, W. Wang, and X. Lin. SPARK: A keyword search engine on relational databases. In *ICDE*, pages 1552–1555. IEEE, 2008.
[18] A. Markowetz, Y. Yang, and D. Papadias. Keyword search over relational tables and streams. *ACM Trans. Database Syst.*, 34(3), 2009.
[19] Y. Mass, M. Ramanath, Y. Sagiv, and G. Weikum. IQ: The case for iterative querying for knowledge. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 38–44. www.crdrdb.org, 2011.
[20] McConnell. Linear-time recognition of circular-arc graphs. *Algorithmica*, 37(2):93–147, 2003.
[21] L. Qin, J. X. Yu, and L. Chang. Keyword search in databases: the power of RDBMS. In *SIGMOD Conference*, pages 681–694. ACM, 2009.
[22] P. P. Talukdar, M. Jacob, M. S. Mehmood, K. Crammer, Z. G. Ives, F. Pereira, and S. Guha. Learning to create data-integrating queries. *PVLDB*, 1(1):785–796, 2008.
[23] Tucker. An efficient test for circular-arc graphs. *SIAM J. Comput.*, 9(1):1–24, 1980.
[24] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 137–146. ACM, 1982.
[25] G. Zenz, X. Zhou, E. Minack, W. Siberski, and W. Nejdl. From keywords to semantic queries - incremental query construction on the semantic Web. *J. Web Sem.*, 7(3):166–176, 2009.